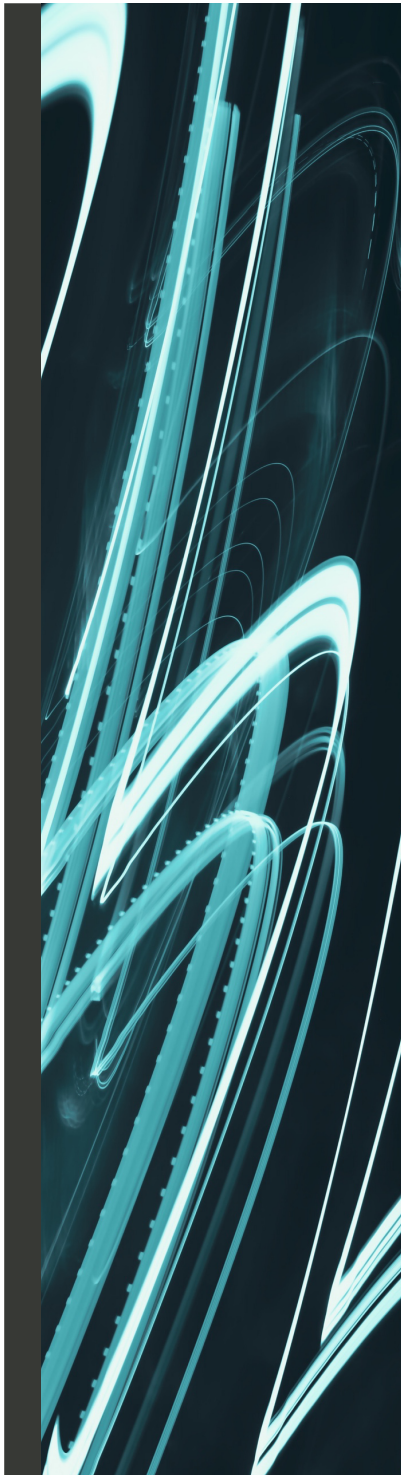
responsiv
simple · effective · distinctive

WHITEPAPER

Designing Services for Digital-SOA Architectures



Preface

Integration of enterprise systems has been a challenge for many years. Initially systems operated as islands of automation, connected only as an exception. Integrated systems are now a primary requirement and Service Oriented Architecture (SOA) is a preferred style. SOA promises re-use, location independence, and the benefits of mediated integration. Determination of scope and behaviour that should be provided by an "SOA interface" has regularly undermined the expected value from SOA.

This paper looks at the challenges around setting the scope for an enterprise interface and suggests the basis for how decisions about interface design should be considered for enterprise architectures. Many of considerations included are appropriate to all integration architectures and not limited to the SOA style.

Intended Audience

This paper is intended to support executive and leadership teams in consumer-focused companies to understand what Digital-SOA and Digital economy really mean, and how they must be addressed.

Disclaimer

Observations and recommendations documented in this white paper are based on our opinions, experience, and research. They are as objective and representative as we can reasonably be, however Responsiv makes no representation as to accuracy or fitness for purpose. Once you have chosen a course of action, it should be thoroughly evaluated to ensure its fitness for purpose.

About the Author

Richard Whyte is an accomplished IT architect with a proven ability to innovate and focus on customer requirements to deliver simple, effective solutions. He has demonstrated thought leadership based on a breadth of technical and project experience spanning Investment Banking, Retail, Manufacturing, and Aerospace, delivering sustainable technology to some of the world's largest companies.

Richard has a degree in statistics and computing and a master's in business administration. He is a Fellow of the British Computer Society (FBCS), Chartered Engineer (CEng), Chartered IT Professional (CITP), and Fellow of the Institute of Engineering and Technology (FIET).

About Responsiv

Responsiv Solutions is a UK based company that specialises in delivering business integration across the enterprise, including API management, Business Process Automation, and Digital-SOA platforms. We work across many industries, including Retail, Financial Services, and Government.

Responsiv can provide fully commissioned solutions that include all the software and professional services needed to deliver an integration platform to support your business plan and grow with your business.

Table of Contents

responsiv
simple · effective · distinctive
WHITEPAPER

Introduction

Architecture is about structure and the way in which buildings and computer systems express themselves to observers, and how they function as a cohesive unit rather than a set of disparate parts. This is demonstrated by the observation that business functionality, for example the need for an accounting system, is not fundamentally changed by choice of architecture style. The primary reason for a well-defined enterprise architecture is to allow the business to gain value from the structure of the overall system.

The Service Oriented Architecture (SOA) style has been the dominant choice for enterprise technologists for over ten years. It is perceived as a great enabler for business agility and to impose a useful structure on large scale IT environments where the need for separation (facilitate agility and parallel work-streams), encapsulation (replacement, upgrades, and manage complexity), and isolation (control, security, and manage change impact) are high priorities. The problem is that firms have struggled to realize the full benefits of the style. One reason for this is the difficulty in designing sustainable SOA Service interfaces.

EAI interfaces: Built from the perspective of a provider

For example

> *Register Gas Meter reading in progress*
>
> *Read System X for Customer profile to get Gas meter type and location*
>
> *Ask System Y for the appropriate way to read meter*
>
> *Initiate meter reading*

In the Enterprise-Application-Integration (EAI) world an application exposes an application-programming interface (API) for others to use; what is included in the interface and its structure is informed by capabilities of the specific application and its internal design. It is independent from the environment. The consumer and environment must adapt to the presence of the application and its API, protocols, and specific meaning assigned to data values. APIs of this nature are generally designed for building distributed applications using procedural computer languages. When the implementation changes the API may change too.

SOA Interfaces: Built from the perspective of a consumer

For example

> *Perform Gas meter reading, including deciding the most appropriate reading technique for the type of meter and its location.*

SOA interfaces are designed from the perspective of a consumer rather than provider. They are intended to hide the underlying topology, data, and system landscape to deliver a consumer agnostic interface. SOA interfaces are intended for use by business processes style applications as business activities[1] rather than procedural subroutines. When the implementation changes the SOA Interface should remain consistent. Often SOA interfaces consume data and function from more than one implementation (source system) in order to present a complete "function" seen from the context of a specific organisation. This is one reason why vendors struggle to provide SOA interfaces that can be used without change by different organisations.

[1] SOA Interfaces and their implementations are often collectively referred to as "Business Services". This paper refers to SOA interfaces to ignore the service implementation and focus on the interface alone.

SOA is not just about Integration

> SOA Harbinger of change: However perceived, SOA challenges the enterprise with a different approach, without which it is unlikely that SOA will prosper any better than previous attempts at EA.
>
> Whyte, 2003

A "Service Oriented" architecture is oriented about the concept of a unit of function known as a "service", which encapsulates a specialised capability that is exposed through a defined interface. The term "Service" refers to both interface and implementation, however internal details of a service are of no architectural concern so long as they are properly encapsulated and accessible only through the defined interface.

An Interface definition is considered to include the Service Level Agreement, which specifies function, performance, and reliability. Implications of total encapsulation include:

- SOA Services may not share implementation except through the published interface
- SOA Services are completely represented by their interface
- SOA Interfaces are designed from the consumer perspective and completely represent a service domain

The result is an architecture that focuses all attention onto Service interfaces, making them a critical part of the structure of the system. They are the key to agility, re-use, version control, release management, ownership, monitoring, technology selection, and system vitality.

Location of the Service Interface

Positioning Service Interfaces away from their implementation, for example on a "Service Bus" delivers an implementation agnostic control point whose availability and locational characteristics are separated from the underlying implementations; the separation delivers a raft of opportunities to the platform that are otherwise lost. Specific benefits include:

- A Single interface may expose functionality from several physically separate components that are considered parts of the "service implementation"
- Interfaces add capabilities to packaged solutions that are otherwise owned and managed by third parties
- Creates potential for central control enforced by a trusted point in the architecture that is independent of development teams working on consumer and/or provider applications and systems

Designing an SOA Service Interface

Design Goals (business)

The design goal for any business architecture is to provide a platform capable of responding to new requirements needed to support business strategy and to support existing business operations. Efficiency of response to new requirements can be measured using the concept of time to value and ability to efficiently support business operations can be measured using a combination of cost per transaction and failure/availability metrics.

Design Goals (Technology)

The Service Bus, of which the Service interfaces are an intrinsic part, must deliver its structural support to the architecture for the whole to function. While the business organisation demands functionality, they also implicitly require efficiency from the IT organisation. Techniques employed by the IT organisation to achieve these demands include containment of the effect and impact of failures, reduction of maintenance overhead caused by the effects of change rippling across the architecture, and by avoidance of unnecessary effort.

Efficient project delivery is fundamentally undermined by project dependencies; a project with dependencies out of its control will need to synchronise its activity to allow those external elements to be delivered. When one project must synchronise with another the effect is to provide excuses for non-delivery and significant loss of project momentum. In many cases the synchronisation of project work can drain quickly contingency from across a programme.

Design Derailment Factors

An Enterprise Service Bus (ESB) and its hosted interfaces is deployed to contain and manage the effects of failure and change, to remove or soften cross project dependencies, and to establish a trusted intermediary that can translate and mediate between technologies and systems with different ways of communicating and delivering their function.

Fundamentally SOA is about interfaces and interfaces are about scope.

Interface derailment factors that undermine business	*Structural factors that undermine the architecture*
• *Proliferation of versions and variants* • *Delivery of overlapping functionality* • *Inconsistent behaviour*	• Lack of containment of change • Circumventing structure • Excessive interactions between consumer and interfaces
• *Inconsistent application of standards* • *Inappropriate granularity and fixed function in interfaces* • *Alignment of interfaces to existing APIs*	• Excessive work forced on provider systems • Excessive dependencies forced on provider systems • Exposure of implementation, e.g. surrogate keys

Figure 1; Business and structural derailment factors

Elements of Interface Scope

A Service Interface consists of one or more operations that provide access to specific functionality and data elements. Relational database, JSON, and other technologies demonstrate that the minimal set of operations required to provide for all data manipulation needs is Create, Read, Update, and Delete. The scope must hide implementation from consumers, and consumers and their context from the implementation.

The form of an interface can be considered against four dimensions determined by the needs of a consumer:

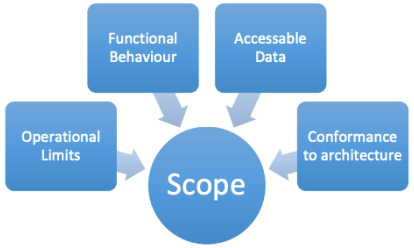


Figure 2; Dimensions of interface scope

An interface cannot rely on being called before or after another action has completed, and should not behave differently depending on knowledge that is external to the implementation and not explicitly part of the interface. This means that behaviour must be explicitly linked to elements passed in the interface, and that all data moved between parties should do so through the interface.

- **Operational Limits** indicate the situations when use of this interface is appropriate and is generally detailed in a Service Level Agreement (SLA). Elements of an SLA should address aspects of the implementation as experienced through the interface. Refer to the appendix for details.
- **Functional behaviour** refers to the actions that are available through the interface, for example triggering a notification or performing some task. It also refers to how errors and exceptions are managed, how selectors are specified and used, and whether a request to read repeating data, results in a complete set, fixed number of records returned, or a cursor style interaction. This paper will refer to this dimension as the "depth" of the interface.
- **Accessible data** is the dimension generally associated with "scope" and the dimension discussed in this paper. It details the fields that can be accessed through the interface and should include the format and validation rules surrounding each data element. This paper will refer to this dimension as the "breadth" of the interface.
- **Conformance to architecture** is a dimension that may not be of high interest to most consumers of an interface but is important to designers and architects and potentially of major importance to those in the support and operational departments. It effectively details the compromises made during implementation of the service and its interface. For example, if all dependencies should be focused on the published interface but a package has a shortcut because of its co-location or performance needs, then this will affect the way the system is managed.

Layered Interfaces

Service interfaces are almost always layered across existing or purpose built (technical) interfaces. These technical interfaces come with their own scope and requirements for how they can be called. Two significant side effects of this are the need to cross reference keys from one interface to another, and the potential for "structure clashes".

Cross-referencing keys is needed when the identifier for an entity in one system is not used to identify the same entity or associated entities in another system. An example is a customer record identified by "customer number" in one system and the same customer's transactions identified in a different system using "account number". To access both parts of the information it is necessary to cross-reference between account and customer. This is a functional concern because the data cannot be effectively used without the join.

Structure clashes (M.A.Jackson, 1975) are more difficult to explain. There are three types of structure clash recognised in this context:

- **Ordering** clashes occur when the data provided to a program is in a different order from that required. The program is forced to wait for the complete set to then sort, or to cache data and identify the logical next record when it arrives. Either situation causes inefficient use of memory and CPU.
- While this clash cannot be completely removed the proposal is that assertions are used to force an order for response to the consumer. This places the load on the ESB or source system, which is more capable of dealing with the resource requirements than a mobile or small device.
- **Boundary** clashes occur when the boundary of one structure, for example the scope of an existing (technical) interface, does not match that of the new SOA Service, in this case data is retrieved and not required or multiple interfaces are called to accommodate the new scope; data may become inaccessible.
- Boundary clashes become more abundant when the scope of interfaces is small relative to the overall dataset. By allowing consumers to specify their own scope we cause some clashes and remove others but generally focus on the optimal compromise. The ability to analyse actual demand allows new interfaces to be built to further reduce the problem.
- **Threading** clashes occur when two threads of execution are operating on the same data such that they interfere with each other. The result can be anything from data corruption, deadlock, or hidden waiting for resources to become available.
- Forcing all consumers to access a set of resources in a specific order and using cooperating synchronisation techniques can reduce wait times and avoid deadlocks as well as provide a point to gather evidence to be used to resolve any clashes.

Determination of Interface Granularity

In the introduction we presented considerations that influence the optimal granularity of an interface in a given situation. Concentrating on the typical enterprise environment the granularity decision focuses on the breadth of data that should be accessible through an interface. The outcome of this decision will impact performance, integrity, complexity, and cost of ownership.

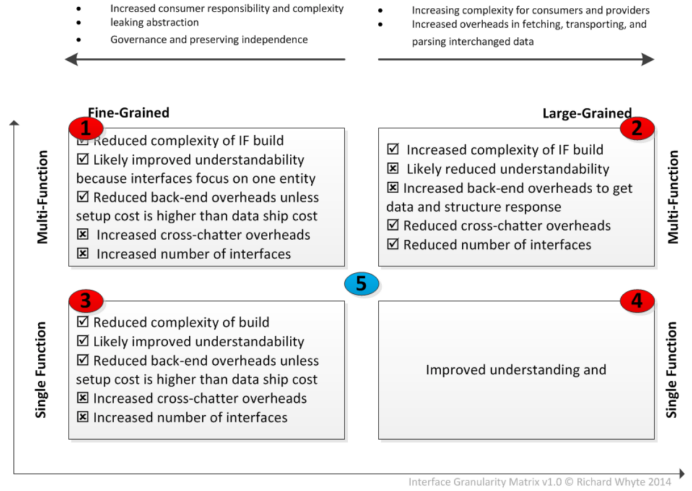


Figure 3; Interface granularity matrix

The evidence suggests that an interface provides optimal overall efficiency when its scope is closely matched to the needs of a consumer [5]. Exceptions to this rule exist but are considered to be edge cases that do not represent the problem being addressed.

This represents a point of optimal balance of complexity and runtime cost (latency, availability, CPU, memory, locks) across all participants from back-end provider to consumer. As an interface specification moves away from this point it cannot avoid imposing complexity or cost onto either the consumer or the implementation.

The Importance of Scope

Consumers call interfaces to achieve their own purposes; two consumers may have different purposes but a shared reliance on elements of enterprise data or function. This creates multiple points of demand for access to data and function, each with subtle variations or extensions to a common requirement.

- *Request [A] Get details of this customer's account transactions; include all transactions*
- *Request [B] Get details of this customer's account transactions; include only standing orders*
- *Request [C] Get details of this customer's account transactions; include last 6 days, exclude closed accounts*

This example can be represented as a Venn diagram (Figure 4). The superset of all requirements from the three requests could be accommodated by a single interface, or by combinations of interfaces as shown in the table.

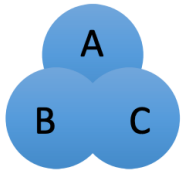


Figure 4; Venn diagram of interface scope

The option to create a superset interface will reduce the number of interfaces, however all requests will exert additional work on the service implementation to provide information that is not required and work on the consumer to remove or reformat information that it requires.

Options to combine the needs of any two or more consumers will result in the same problem to varying degrees and will also lead to more interfaces being built. Arbitrary setting the scope of an interface to include [A+B] and another with [C] leads to consumers that require [A+C] calling both interfaces and again removing unwanted data.

Brief summary of compromises forced by interface scope decisions:

Interface Scope	*Decision Name*	*IF–Proliferation*[2]	*BE- Load*	*Reliability*	*Cross-talk*	*Description and cost of compromise*
[A], [B], [C]	Full separation	High	Low	High	High	Any overlap in function required will cause duplication in the interfaces and confusion for consumers as to which interface, they should use. Increases test, development, design, and impact analysis costs. Future maintenance is likely to lead to increased overlap of function and further confusion.
[A-B], [C]	Arbitrary consolidation	Med	Med	Med	Med	Improves the efficiency of those consumers that only need [C] but those that need [A-C], [A]. [B], [B-C] are all compromised. Divergence in the needs of [A] consumers will affect consumers that really only want [B] data.
[A-B-C]	Full consolidation	Low	High	Low	Low	Ultimate high load on implementation systems with all the

[2] IF-Proliferation is the number of interfaces required, BE-Load is the back end or implementation load, Reliability is the mathematically predicted availability of the interface (see later in this paper), and cross-talk refers to the potential for consumers to need to call multiple interfaces to achieve their goals.

Optimal Granularity for One Purpose is Inefficient for Other Purposes

The conundrum created by this situation is that what is optimal for a single consumer is unlikely to be optimal for another. The re-use of the "optimal" interface means that in practice it will be optimal for very few consumers.

Interfaces that are extended for new purposes extend their ability to be reused for different purposes at the cost of moving their point of optimal efficiency away from the primary consumer (the one for which the interface was first designed.

It appears that whatever scope is chosen it will be more appropriate for some uses than others, and that if optimal efficiency is required for all purposes, then each must have a bespoke interface. If the goal is a set of reusable interfaces, then a static interface model cannot achieve an efficient outcome.

Design Considerations

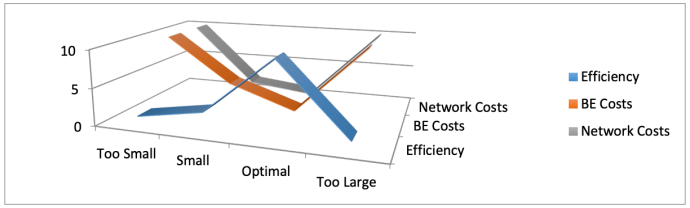


Figure 5; Relative measures strengths and weaknesses of large and fine-grained services (indicative only)

Notes

[1] Interfaces that are too small and specialised lead to chatty behaviour between consumers and the service but may impose a reduced load on back-end (provider) systems. Often the consumer must perform their own aggregation of several calls to achieve their objective, for example populating a page or drop-list. Fine-grained interfaces tend to require the consumer to understand more about the implementation and its operation than larger interfaces. The need for aggregation can be alleviated by increasing the breadth (number of fields supported) of the interface, however taking this too far can lead to complex, sparse, structures in the interface. Interfaces that are backed my multiple implementations will suffer increased latency and reduced availability caused by having to visit multiple systems.

[2] Transactional integrity relies on the ability for a transaction coordinator to coordinate transactions across their entire breadth. Fine-grained interfaces can segment the transaction across functionally arbitrary lines. The result is a service interface that cannot be coordinated and that has significant recovery complexity.

[3] Interface complexity is driven in the case of fine-grained services by the propensity to have interfaces that overlap with subtly different behaviour and the need for consumers to coordinate across interfaces. Interfaces that are too large suffer from the interface itself becoming complex to provide and to consume.

[4] Cost of ownership is increased for interfaces that are too fine-grained because they leak the implementation and tend to prevent changes to the underlying implementation from being transparent. The added complexity of consumption and overlapping nature of the interfaces can significantly increase testing effort and confuse programmers as they build consuming applications.

[5] Leakage refers to the exposure of internal implementation details through the interface or by way of forcing ordering on interface calls. Fine-grained interfaces tend to suffer most from this problem, but it can apply to any interface. One detail of implementation is of course the scope of capability provided by the underlying systems, which can change when the system is upgraded or replaced.

Delivering Optimal and Efficient Interfaces

Second-guessing the needs of future developers is fraught with danger. SOA interfaces should therefore avoid fixing the service boundaries until the last moment, perhaps when the consumer is deployed or right to the time it makes the first call to the interface.

Declarative Interfaces

A declarative interface is capable of adapting it behaviour and form to the situation of its invocation, for example the interface may detect that the user is on a low bandwidth and reduce the quality of a video, or may understand that the user wants fast response even if the data quality is not 100%.

An interface that allows consumers to indicate runtime scope and responds accordingly may be more complex to build than a non-adaptive interface but delivers an efficient interface to the majority of consumers. Consumer complexity, number of calls, and data-shipping overheads can be managed and reduced.

Using the Interface as a point of separation the adaptive interface can be allowed to adapt on the consumer side or the provider side independently of one-another. For example, delivering adaptability to the consumer but with no optimisation of provision through to optimisation of both.

Implementation Optimisation

If a static (non-declarative) interface makes N=3 fields available from N=3 different provider systems, each with the same availability $N_a = 90\%$ then the interface availability becomes $N_a^N = 90\% * 90\% * 90\% = 72\%$. Similarly, if the latency of one provider system is 3 seconds compared to 2 seconds for the other systems, then in an asynchronous environment the best latency for a static interface is 3 seconds. In a serial (one after another) call regime the latency will be slightly over the sum of the individual latencies.

In an adaptive interface, various optimisations are possible to deliver savings to the overall platform and improve availability.[3]

For example, if the consumer only needs one field from the interface, then only get the one field; avoid the other costs. Avoiding this effort provides significant benefits to the overall platform by reducing the work to assemble a response, removing the effort to find and format the data, and avoiding the potential for an interface to be unavailable because of an underlying system when that system is not actually required.

Increased Opportunity

If the declarative interface allows a subset of fields to be requested, then the harm caused by having a large breadth of attributes to choose from is reduced. If the fields available are all connected to a single large entity, then management of the entity can be fully encapsulated, hiding boundaries between back-end systems, removing cross-reference complexity and avoiding cross system chatter.

From an SOA perspective this is now starting to move toward the goal of excellent isolation, encapsulation of implementation, and context insensitivity. Changes to implementations and details of order of access, locking strategy, and availability can be dealt with for all consumers and in a manner that protects the majority from the effects of change.

Refer to the appendix for additional design tactics.

Declarative Interfaces

Declarative interfaces allow a consumer to ask for something to be done, the outcome, rather than being concerned about the way the outcome is achieved. They can change their request and behaviour to suit the

[3] See SLA appendix

moment to benefit from reduced data management and lower bandwidths as well as providing the interface more latitude to optimise its latency and improve availability.

Structured Query Language (SQL) and all self-describing grammars do this already. Rather than defining a static interface they allow the request to be self-describing. Secondary advantages quickly become apparent; for example the ability to perform analytics based on fields rather than interfaces; understanding the fields used together, what is removed for security reasons most often, those using specific data, and the performance characteristics of different calls. In marketing terms the difference in resolution of this data is the same as the difference between counting baskets of goods and counting individual products.

Optimal granularity (breadth) – every time

- The primary justification for this declarative approach is to allow the consumer to select granularity (breadth) and function (depth) according to their needs at the very last moment. The number of requests can be reduced compared to a vanilla fine-grained approach, and availability and load placed on provider systems can be reduced compared to the large-grained approach.

Fact based hard optimisers

- The ESB can choose to get everything anyway and release to the client what was asked for, or can gradually build optimisations to get only what is required. Any optimisation in the shared part of the architecture benefits all consumers.

Explicit behaviour

- Behaviour is determined by a combination of default behaviour and assertions. Default behaviour should conform to the principles of simplification of the interface and minimal cost.

Assertions

Assertions modify the behaviour of an interface to better suit the requirements of a given consumer. It operates by moving the behaviour away from the documented default behaviour, which should be the simplest and least cost behaviour. An example of this might be that the default response for a request for payment-transactions is to return no more than ten transactions, however an assertion may increase this to return twenty payment-transactions.

Declarative interfaces can remain deterministic

Declarative programming allows instructions to be given to describe what you want to do, and not how you want to do it. It is left up to the machine to figure out the how. Imperative programming focuses on the details of how something is achieved. (J.W.Lloyd, 1994)

When a programmer uses SQL, a business rules set, or late binding C++ or Java the details of execution are hidden and fundamentally declarative. In fact, most calls to closed subroutines requests an activity to be completed while leaving details of how it is achieved to code in the routine. This concept is a natural extension of the desire stated in the SOA model to fully encapsulate and “hide” the implementation of a service.

A database request for fields A, B, C may be satisfied from cache, from index only, from a materialised-view, or from a base table. The caller has no idea and no real interest as long as the service agreement is not breached. The addition or removal of a field, change to database configuration, current database load, and many other factors may cause the database to change its access path at any time.

Regardless of the path chosen the mechanism is physically accessing the same physical data or data tagged as directly equivalent. This ensures that results of an operation are deterministic given that underlying data does not change.

If system of record (SOR) accuracy is not required, then the mechanism has the option to access older copies at its convenience. Clearly this behaviour can affect quality of response and is controlled using "*assertions.*"

Performance and contention characteristics

Performance and contention characteristics may change depending on the access path used and system conditions, for example a path may be excellent when the system is under low load but very poor when demand is high. The mechanism allows APIs (their adapters) to be categorised by their availability and performance characteristics as well as being prioritised within a category based on their relative cost.

Use-Cases for Late Binding SOA Interfaces

If an interface is to choose between multiple sources of data, it will require a mechanism to determine when a particular source is used. This model addresses the following cases:

1. Multiple interfaces to the same data
- *All interfaces see the same exact data value for a given attribute but include different combinations of attributes.*
- A company has multiple interfaces that optimise consumer effort in different applications. The interfaces overlap with regards the data they access. An example is when application 1 requires fields A, B, C and Application 2 requires A, B, C, D, E. Two interfaces exist to allow both to operate, either because of a second version and an inability to upgrade previous consumers, or an explicit decision to have two interfaces.

2. Multiple interfaces to the same data at different levels of quality
- *Interfaces see different copies of the same data such that the accuracy and timeliness may vary across interfaces.*
- Data is sourced from one place designated "system of record" but copies are available for read-only that are accurate enough for most purposes.

3. Multiple identical interfaces to different data
- Multiple system instances host the same data partitioned by brand or account type or another routable attribute. The mechanism must distinguish to determine where to look for the answer.

Example

A request for account-details specifies the account and the fields that should be returned.

ServiceName (Read acc-sort=12-13-14,acc-num=12345678) [acc-name, acc-owner, acc-type]

1. Security determines whether the user is allowed to perform the operation (Read) on the fields being requested. Confidential fields are removed from the selection criteria and not retrieved from the source systems.
2. Values specified in filters (12-13-14) can be resolved to a brand, for example Bank-A, which is considered to be an interface category. Interfaces in the candidate-list that do not include this brand can also be removed.
3. The SSI "master-list" of all APIs it understands is reduced to form a list of candidate APIs on the basis that they contain at least one of the fields we require and the keys we have.
4. Finally, the chosen interfaces are called as listed in the candidate-list and the results correlated and returned.

Reference Design

The role of an ESB SOA Interface is to isolate consumers to changes to location, topology, protocol, semantic, and other aspects of one or more provider system(s). It is also to provide the enterprise with an interface suitable for use in a Business Process.

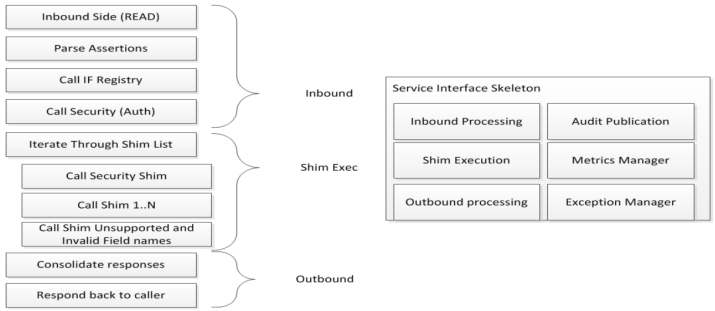


Figure 6; Logical interface schematic

Inbound Processing

Accepts the payload (header, body, and assertions) and parses the representation into an internal format that is optimised for processing. It calls the Interface Registry (IF Registry) to validate field names exist, and that they have implementations (data sources) attached, and to calculate the optimal calling plan for shim activation.

The security service is called to ensure that the caller has permission to access this service and to perform the operation on the fields requested. Fields that are not allowed are removed from the call in a read, and for a write the user is informed that the call cannot be completed.

The interface registry is called with a request that includes the security-allowed-fields-mask to provide the list of interfaces required to satisfy the request. The list of shims returned will always include the security (Restriction) shim as the first entry.

Shim Iteration

An adapter layer known as a "shim" is constructed to individually wrap each existing interface. Each shim contains logic to transform data between the existing interface (API) and the canonical model used by the ESB. The scope of the existing API is extended, possibly by calling a cross-reference API, to include a set of integration keys that can be used to join data from this shim to that of others and to the security system.

The late binding mechanism relies on content routing to select a sub-set of shims that can understand the requested key, and that can provide the data; for example a sort-code might be used to select a particular system that stores the range of accounts with that sort-code.

The sub-set generated by content routing is then reduced further to remove all interfaces that do not contain fields that are being requested, and then further to minimise the set of interfaces that will be physically called.

This final list of shim-names is used to retrieve and execute shims from the shim library.

Each shim is called passing the service header and assertions. It executes and returns its results in a database table in a form already transformed to the canonical.

Consolidation

Consolidation of results from multiple shims (and their independent tables) is performed when all shims have completed, or a timeout has occurred. This activity combines results from the result-tables into the response message that is then sent back to the consumer.

If an assertion for multiple (cursor) operations has been made then the data is retained, otherwise it is deleted.

The Security (restriction Shim) will call the security service and ask for a list of keys for this domain that have a relationship for this operation to the security key for the data being recovered by the other shims. Results are fed back to the Consolidation component in a database table.

Managing Depth (Function)

The behaviour of an interface can be fixed and often is fixed in a first release. Over time the behaviour is not quite what is required but the interface (breadth) is fixed; there is no way to inform the implementation that a variation of behaviour, however slight, is required. Responses to this situation follow three themes:

1. Put up with the problem
- This response commonly leads to additional work or complexity in the operational environment in the form of new processes that compensate for the behaviour differentials. For example, an interface logs activity against the call-centre because the first consumers required that behaviour. Subsequent users of the interface must be removed from the call statistics by additional operational processes to allow accurate reporting.

2. Deploy a new version of the interface that exhibits the new behaviour
- The new version of the interface is a variant of the primary implementation. This adds testing and programmer confusion overheads to all subsequent use of the interface as well as additional duplicated work to maintain the interfaces. Interface variants exhibiting different behaviours are considered by the author as an anti-pattern.

3. Infer new behaviour from some data already present in the interface
- Another common response is to link behaviour implicitly to values provided in the interface, for example if the user asks for "ABC" then assume they are a mobile device and behave differently from those using the "XYZ" code. This links two concerns that are potentially divergent and overloads the meaning of a data value. This response is another anti-pattern that should be avoided.

A useful approach to this problem is to deliver basic and low-cost functionality that provides capability to the expected common use of the interface. Additional function or variation of the default behaviour is then explicitly controlled using assertions.

Standard Behaviour

Standardised and documented explicit rules that describe how the interface will respond in the default case; this should be useful to the majority of consumers but not necessarily all, and should represent the least-cost meaningful response.

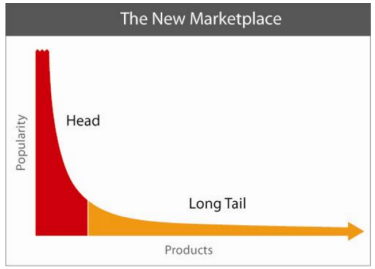
- Maximum response rows from repeating data will not exceed 32K or 200 rows
- Interfaces will respond using the same protocol as invocation (XML→XML, JSON→JSON)

Modified Behaviour

Modified behaviour is controlled explicitly using assertions such that any implied behaviour can be adjusted to suit the outlier cases without building new interfaces that are substantially the same as others. Example assertions include:

- Assert Response Format=JSON/Brief! Default behaviour is to respond using the protocol of the request
- Assert Debug=Performance, Basic

Conclusion



The granularity of a service with regards its breadth (fields included) and depth (behaviour) should be as closely matched to the needs of every consumer.

As with market segmentation there is a market demand for interfaces of a particular granularity but also a large number of consumers whose requirements move away from any predefined granularity. The cost of developing and operating a large set of interfaces that have overlapping functionality is considered excessively expensive.

The ability for consumers to declare their needs and have a service interface that adapts to their needs can extend the number of consumers that are well provided for by a single interface.

Programmers building consumers are helped by the reduced choice of interface and removal of details from their concerns. SLAs provide assurance that the interface will respond as required, and the mechanisms and complexity of how that response is achieved can be safely ignored.

Inefficiencies in the way a service is providing its capability are more quickly highlighted and any optimisations applied will benefit all consumers rather than the few that identify the problem.

Encapsulating cross-referencing and other aspects of the underlying systems landscape protects consumers from changes and failures in that domain as well as creating options for optimisation and maintenance that would otherwise be difficult to implement.

Choice of a capable canonical model for data representation extends isolation to the data and its structure in a way that allows data to be represented in different ways to optimise source system operations without impacting consumers.

Backward compatibility is provided by explicit use of assertions that modify the basic behaviour. This avoids variants and again provides a way to support minority requirements.

In short it is difficult to understand why these techniques are not more widely adopted.

Bibliography

C.J.Date. (1986). An introduction to database systems (Vol. 1). The Systems Programming Series Addison Wesley.

C.J.Date. (1994). Relational Database Selected Writings (Vol. 1). Addison-Wesley.

Chris Anderson. (2006). The Long Tail: Why the Future of Business is Selling Less of More. Hyperion Books.

Constantine, Y. a. (1979). Structured Design - Fundamentals of a discipline of computer program and systems design. Prentice Hall.

Fiammante, M. (2009). Dynamic SOA and BPM Best practices for Business Process Management and SOA Agility. IBM Press.

J.W.Lloyd. (1994). Practical Advantages of Declarative Programming.

M.A.Jackson. (1975). Principles of program design. Academic Press. Ould, B. a. (n.d.). A Practical Handbook for Software Development.

Shiller, L. (1990). Software Excellence. Yourdon Press.

Whyte. (2003). SOA Management and Governance. IBM iRAM Internal Publication.

Wood, D. (2013). Why Adopting the Declarative Programming Practices Will Improve Your Return from Technology. Forbes.

Web Resources

Service Integration Maturity Model (SIMM)

http://www.ibm.com/developerworks/webservices/library/ws-OSIMM/index.html
https://collaboration.opengroup.org/projects/osimm

WPS in relation to the SIMM model

http://www.ibm.com/developerworks/websphere/library/techarticles/0904_clark/0904_clark.html

SOA Reference Architecture

http://www.ibm.com/developerworks/library/ws-soa-ref-arch

(Chris Anderson, 2006)

Appendix: Concurrency and Read Integrity

ESB Services operate in a layer above the source systems and often have a role that aggregates data from more than one interface. This situation is true regardless of whether dynamic or late binding techniques are used. The overheads associated with use of transactions than lock resources are significant within a system and can have catastrophic impact if they are routinely applied across multiple systems.

Transactional integrity is considered using the dimensions of Atomicity, Consistency, Isolation, and Durability (ACID).

- **Atomicity** is the ability to commit or rollback all parts of a transaction as a single unit of work. The ESB should be capable of making requests to multiple systems as one unit of work but should not be expected to coordinate transactions that span systems except in very special circumstances. In the case of businesses using CICS or other transactional mediation it is recommended that this "lower-level" mediation be made responsible for cross-system transactional management if it is required.
- **Consistency** ensures that a given transaction moves a data repository from one known and predictable state to another and cannot result in an inconsistent or unknown state. Durability is the characteristic that a transaction once accepted by a system cannot be lost due to hardware failure. Both consistency and durability is managed by the source system and should not be of concern, except for its own management, of the ESB.
- **Isolation** ensures that concurrent transactions are properly serialised such that data is made visible to two or more transactions in a manner appropriate to the transactional intent. For example in one case reading uncommitted data may be acceptable (no isolation), in another all data read must be already committed but may be allowed to change after being read and before the read-transaction is complete (cursor stability). Full read stability insists that uncommitted data cannot be read and that, once read, data cannot be changed until the read transaction is complete. Finally, repeatable read insists that no changes to the dataset are allowed while the operation is being performed.

Two use-cases predominate when using transactions initiated by a service bus; "read-think-update", and "replicate-or-trigger" patterns.

Read-Think-Update

This pattern typically reads data from a system to display it for a user to then decide and confirm an update to the same data. The challenge is that the "Think-Time" is typically measured in minutes and is effectively unbounded, and user connections can fail during the think-time. As a result, the problem is typically solved using "optimistic locking".

Replicate-or-trigger

This pattern typically triggers an action when data is changed in a repository to cause the data or other change to another repository. These patterns attempt to ensure that multiple copies of the data exist or that two systems are synchronised without the use of locks. Replication can be triggered either by a change to data or combination of time and change.

In general concurrency behaviour associated with data-changing ESB Service operations is inherited from and remains the responsibility of the respective endpoint service implementation; situations where this is not acceptable should be managed below the ESB and made available as a single atomic API that the service bus can access.

ESB Service design should ensure that enterprise data resource managers on behalf of the ESB Service hold no locks.

Read Integrity when Accessing Data Across System Boundaries

1. The ESB Services do not hold locks or resources in source systems between ESB Service boundaries
2. ESB Services hold locks in some circumstances but generally this is bad practice
3. ESB Services use optimistic locking where appropriate

Each ESB Service controls access to several source systems, which it uses to retrieve and update information.

Reading data from source systems

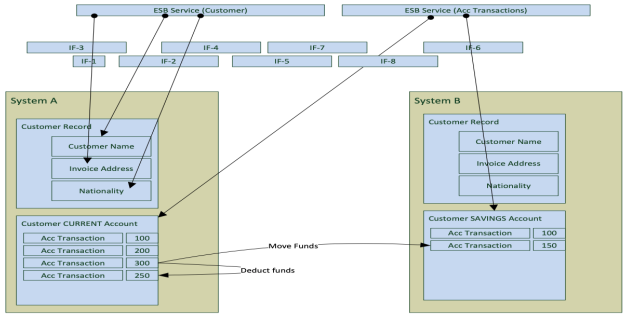


Figure 7; Read stability and data aggregation

Case 1: Stable read

A consumer requests information about a customer. The ESB Service determines that IF-1 and TF-2 will provide the information and calls them. Each call requires that the filter fields are converted from standard to IF specific format, which may include semantic transformation (lookup and cross reference).

This operation is stable under read because the fields are from different parts of the same record in the same system.

Case 1a: Stable read

It is possible for the ESB to determine that it should read some of the customer record fields from System A and some from System B. This will only happen when the fields are not duplicated, and they are marked as "SYSTEM OF RECORD".

It is intended that fields are marked with PSOR and ASOR to indicate their primary and secondary systems of record. To reduce the opportunity for inconsistent reads and writes all WRITE operations should update these records first and then apply changes to other copies of the data.

It is understood that this is not always the case for existing updates. Please see synchronisation strategies for more information.

Case 2: Potential instability (cash transactions)

A consumer requests detail of ALL accounts, specifically the account balances. This requires the ESB to collect information from more than one system and exposes the possibility that the read occurs between parts of an update transaction.

Action timing	Sys A	Sys B	Total	
Before dr/before cr	300	100	400	Stable read that provides the correct total
After dr/after cr	250	150	400	
After dr/before cr	250	100	350	Arguably a problem with all banking transactions but provides the wrong answer
Before dr/after cr	300	150	450	

Comment: While this possibility exists the cost of prevention is potentially high. Potential strategies are:

1. Record all transactions that move money between accounts owned by the same customer and check that the CR and DR are both applied.

2. Record a transaction date/time on every record. Use this to determine whether the record is likely to be stable

Case 3: Update stability after read

It is often the case that a consuming application reads data that is not locked, and subsequently performs an update or delete on the same data. The issue is whether the read/update was essentially atomic, or whether the record was modified between read and update.

Locking the record is not the answer for the following reasons:

1. There is no control over how long the consumer will delay between read and update
2. Read locks are shared and updates are exclusive; in a concurrent environment this will cause deadlocks
3. Exclusive locks for read will reduce concurrency and tend to slow the whole enterprise

There are alternatives that can provide protection:

1. All interfaces (this is a source system concern) have a consistent access path
2. If records can be changed
    a. All records have a date/time stamp to allow detection of intermediate updates.
    b. Consumer reads record (dd:mm:yy:ss:hh:mm) and updates record that contains the same value
    c. Re-read if the mark changed
3. If records cannot be changed
    a. ESB maintains a hash digest of the record that was read.
    b. When the update occurs, consumer provides hash and ESB compares hashes before update.
    c. Re-read if hash changed
4. If records across systems must be coordinated exactly
    a. This requires a distributed transaction; the plan is to offload this responsibility to the source system notified as the system of record.
5. If the hash/stamp changes too often then a timed lock can be applied
    a. Records can be exclusively locked for a short period (10-30 seconds) to avoid race conditions that cause all consumers to continually attempt update only to find stamps have changed.

Appendix: Open Service Integration Maturity Model (OSIMM)

Integration of IT systems and their underlying data has challenged firms for many years. In the age of the mainframe it was usual for large organisations to install a single computer to perform a specific departmental task, for example payroll or perhaps general accounting. Organisations with more than one machine would segregate the work to ensure that data transfer between the two was minimised.

The explosion of small and low-cost computer resources led to proliferation of databases and function. Initially data was replicated and synchronised between systems, however eventually the volume of data, improvements to network technology, and a desire to build distributed applications led to a distributed application style based on applications calling between host machines. The remote procedure call (RPC), DCOM and CORBA technologies sprang up to support these types of application. Generically these programmatic access points are referred to as Application Programming Interfaces (API).

The problem with an application (or several) integrating directly with remote capabilities is that it leads to every application needing to know how the interfaces operate and how to connect and use the interface. The practices lead to uncontrollable dependencies on the location of applications, firewall rules, languages and interface behaviours, and the way information is represented.

Figure 8; The Opengroup Service Integration Maturity Model (v2)

Along comes mediated integration. The introduction of an integration hub addressed issues regarding differences in interfaces, semantics, and protocols as well as isolating consumers from changes to location but left the fundamental proposal to be point to point API manipulation with the consumer responsible for understanding how to drive the API set associated with a specific package or system.

Enhancements to the model introduced cross-referencing and dynamic or hub centric routing, allowing the hub to decide the target of a request based on its content or origination. The enhancements threw up further challenges, but the benefits were worth the effort.

SOA extended the benefits of independent and well-defined APIs to the structure of the architecture.

The OSIMM model attempts to show how integration matures in a typical organisation, however it is only one perspective and most organisations have a unique "sweet spot" that represents a good balance between investment and benefit.

Source: http://www.opengroup.org/soa/source-book/osimmv2/model.htm

Appendix: Service Level Agreements

Service Level Agreements (SLA) should be defined and exist for all interfaces and underlying implementations. While often overlooked it is clear that the interface may have a different resilience or performance from that of its underlying systems. Interfaces that combine the results of multiple systems (implementations) or have a choice of implementation targets are particularly exposed to this difference. This section will refer to interface rather than system, but the points are applicable to both.

A good SLA will define the operational limits of the interface before defining how it behaves within those limits. It will then set out its performance characteristics along appropriate headings, for example:

SLA for the XYZ-Interface

This interface delivers information about members of staff. Within this context staff includes permanent, temporary, seconded, and agency staff that can be assigned activities.

Performance

	Throughput	*Latency*
Normal Operation	1-10,000 TPS	1ms
	10,000+	Undefined
After on failure (internal)	1-8,000TPS	1ms
After two failures (internal)	1-6,000TPS	1ms

Resilience

The service is reliant on several systems that have their own SLAs. Depending on the functionality required the resilience of this interface could change because of its reliance on different systems. In the example below the basic details of a staff member is available 99% of the time, however the staff availability is only 81% available and if you need all data available then the service is only 72% available.

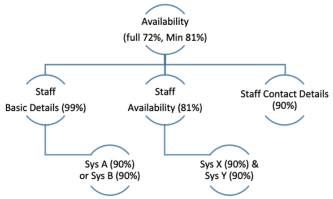


Figure 9; Interface availability tree

Data resilience is maintained to conform to dual site redundancy with standard fail-over delays in the event of DC loss.

Accuracy and Security

The interface will deliver calculations to an accuracy of 98%. Transactional integrity is managed using XYZ capability that assures Data sent to and managed by this system is secured to the level-X policy. Uses that require a higher level should not use this service.

Appendix: Design Tactics

Control and Re-use

Service interfaces capable of delivering backward compatibility will avoid proliferation of versions and the cost of upgrading consumers. They will contain the ripples of change and avoid proliferation of divergent interface variants.

The focus for control and re-use of capabilities is on the Interface rather than toolkits, systems, or arbitrary units of code. The cost of duplication inside service implementations is more than offset by reduction in dependencies and the containment of problems.

Separate Concerns

Service interfaces built to optimal granularity will avoid excessive work in provider systems, limit complexity to consumers, and avoid reducing availability unnecessarily. They will foster re-use and reduce latency and resource consumption, and maintain data confidentiality.

Separation of different aspects of a problem is a fundamental design strategy that allows potentially conflicting concerns to be handled independently. Interfaces factored to separate the exposed interface from the service implementation deliver greater flexibility than those that do not separate the concerns. Multiple implementations can co-exist, and the interface may fail or load balance between different implementations as required to maximise performance and availability.

Adopt a Data Model Agnostic of all Existing Providers

Service Interfaces conforming to a model that is independent of implementations remains consistent in the face of system replacement and extensions to the underlying capabilities. This drives consistent language and style across interfaces, and provides for a compatible set of identifiers to join information from different interfaces.

Industry models and data schema are good starting points for an independent data-model that is not based on existing providers or be influenced by their boundaries. For example, an interface that returns customer information should return ALL relevant information regardless of its current location. The model is divorced from current realities of provider systems and cannot leak details of system boundaries. The service must perform cross-referencing in a manner hidden from consumers to complete the isolation.

Cross-referencing is internal

Keys used by the consumer cannot include surrogate internal keys used by existing systems because they are not known to the consumer and have no meaning outside the system of origin. If that system is replaced, then the key sequence may change or duplicate. The Services internally need a way to reference one key from another, but this should not be exposed to consumers.

Availability

Of course, the downside of interfaces that span implementations is that the availability of this service is now dependent on more than one provider. This must be properly considered and managed.

See appendix for more detail. What we need is a way to optimise away calls to systems when the consumer is not really interested in the data from that system.

Appendix: Implementation Selection Mechanism

Fundamentally the mechanism allows attribute names in the request to be mapped to a physical shim (adapter), which in turn accesses the source of that data to read or write. If there was one big adapter that contained all fields in the SSI, then the mechanism can be seen to be trivial; it simply selects the one interface and uses it to complete the request.

[Acc-Name] ---- Path 1
---- Path 2
---- Path 3 → Value of Acc-Name

Single Version of Truth: The System-of-Record

To reduce the overheads of having one big interface that does everything we could have two, one that operates on attributes A-M and another for N-Z, actually we could have several to allow for A-B, B-M, N-X etc.

If an attribute [G] could be accessed through several different paths then it stands to reason that we should choose the path with the least cost. In the case below we can use the interface [F-H].

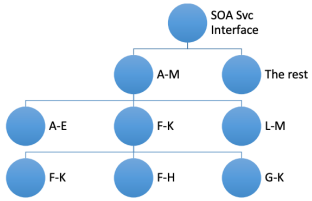


Figure 10; Example - interface hierarchy

In this case there is only ONE version of the data but several pathways with different associated costs. The mechanism simply looks for the least-cost route to the data.

Multiple Copies of the Data

[Acc-Name] ---- Path 1
---- Path 2
---- Path 3 → Value of Acc-Name (ASOR)
---- Path 4. → Value of Acc-Name (Fast Update)
---- Path 5. → Value of Acc-Name (Medium Update)
---- Path 6 → Value of Acc-Name (Slow Update)

Only one version of the truth is allowed but there may be many paths to that truth. The adapter (shim) attribute that accesses ONLY THAT COPY is marked as being the PSOR meaning Primary System of Record. All other copies of this data are indicated as copies. Multiple copies exist but the mechanism always chooses the copy designated as the system of record unless it has been informed that copies are suitable.

- Searches for data with no tolerance for update latency it will access only PSOR attributes.
- When it performs a write operation the PSOR is the first value updated.