**Fine-Grained vs. Coarse-Grained Services: Services and Interfaces**

The idea of fine-grained vs. coarse-grained services and interfaces has long been a tagline of service oriented architecture (SOA). However, the literature has little in the way of substantive guidance as to what the terms mean and how to make use of them. With respect to VistA and API 2.0, fine-grained often refers to the APIs giving direct access to VistA and any services that make direct use of these VistA APIs; coarse-grained refers to any combination of those fine-grained services. While we can adopt such a designation, it is somewhat misleading in any SOA sense, and this paper will attempt to provide some consistent guidance in the use of these terms.

Services and interfaces

There are fundamental differences between the granularity of services and the granularity of their corresponding interfaces. What exists in the literature [covered particularly well in ZapThink ref] and best practices indicate that the granularity of a service typically refers to the scope of the business functionality the service exposes [Tech Republic ref]. To be discoverable and have predictable use to a consumer, a service must have well-defined functionality and well-defined outcomes (defined as *real world effects (RWE)* in the SOA-RM [SOA-RM ref]), typically documented in its service description. However, what functionality corresponds to a fine-grained service as compared to a coarse-grained service is often a matter of context and who needs to use the service, e.g. a system expert vs. someone dealing with higher-order business requirements.

Coarse-grained (fine-grained) interfaces are more than the interfaces to coarse-grained (fine-grained) services. The granularity of an interface indicates the degree to which it is prescriptive vs. being configurable. Here, we characterize a fine-grained interface as one that is narrowly prescribed to a specific data element or set data elements and a coarse-grained interface as one that can enable use of a information ranging from a more arbitrary set of data elements to a wider array of more complex business entities. In general, a coarse-grained interface is less sensitive to changes than the fine-grained one. *[http://michaelpoulin.sys-con.com/node/523434]* In addition, a fine-grained interface tends to expose more of the implementation data structure and has been shown to increase vulnerability to attack.

It is particularly important to note that a fine-grained service may have a coarse-grained interface and a coarse-grained service can have a fine-grained interface. The granularity of services and interfaces will be explored further in separate sections that follow.

Fine-grained and coarse-grained services

The granularity of a service talks to the amount of business functionality the service exposes, but characterizing it as fine-grained or coarse-grained is often a matter of context. For example, a given service may validate a prescription, while a second service would check inventory, and a third service would handle fulfillment. Alternately, a single service could combine the previous three services into a single prescription fulfillment service without exposing the component services to its consumers. By definition, a composite service (e.g., the single combination of the three services) would be considered coarser-grained than its components; however, which is preferable depends on what the consumer needs to accomplish. For example, a prescribing medical practitioner may want to validate the order is correct before giving it to a patient and the prescription validation service is the appropriate granularity; for the patient, knowledge of this is unnecessary.

A goal of the SOA approach is often wide reuse of services, and granularity may play a practical role. A service that provides scoped but widely applicable functionality is a good candidate for reuse in many solutions. These could, for example, be services providing basic CRUD (create-read-update-delete) functions or a service providing authentication of a consumer and authorization for the service request. Note, the CRUD services could be considered fine-grained while the authentication/authorization functions are possibly the outcomes of a composite service that would be considered coarse-grained. However, it is clear that both are examples of functionality applicable to many solutions.

The components of a composite service are often discussed in terms of being *atomic*, where atomic services can be thought of as exposing a capability whose implementation is not itself composed of other services. Alternately, a capability implemented through the composition of other services is referred to as a composite service. However, as [Zapthink ref] notes, this distinction is quite murky. A service that seems atomic to a service consumer who has no visibility into the implementation of that service might actually be composite from the perspective of the service provider. Indeed, the SOA principle of opacity would say that all services should seem atomic to a consumer, and the only way to definitively know if a service is composite is for a service consumer to dive into details that contradict opacity and similarly the rule of loose coupling. No service consumer should be required to know how a service is implemented.

As composition proceeds, a business service may result that is constructed through several levels of composition and is likely to expose more specialized business processes. While such a service may be useful to a large group of consumers interested in that business function, it is unlikely to be a good candidate for further composition.

Fine-grained and coarse-grained interfaces

We can characterize the granularity of the interface in terms of the range of applicability the interface enables. Think in terms of business entities and whether a service is applicable to one or more than one business entity (or to different versions of the same entity). Also consider the scale of the entities. Are the entities discrete data elements (or a list of discrete data elements)? Does the interface support applying a business function to multiple classes of business entities? For example, getABC can be looked at as the interface of a single, dedicated service to get data element ABC. A second discrete service could be getDEF. Now, a coarser-grained service could be get(arg) where arg can be ABC or DEF. However, the interface implies that ABC and DEF are the same data types. A more general interface could specify get(ID) where the argument is an identifier of some entity without specifying the type of entity and it is up to the underlying capability to identify (or be told) the entity type and deal with it appropriately.

Similarly in a write mode, a fine-grained interface might appear as createABCRecord, where the result is solely creation of an instance of the ABC record, and likely just for one prescribed version of that record. A coarser-grained interface could be create(IDofABCRecord) and the payload of the service request would be data that conforms to the structure associated with the ABC record identified. Note, the latter interface can support numerous versions of the ABCRecord or potentially other records. It is up to the capability accessed through this service interface to apply the processing that corresponds to the record type and version or to return a fault indicating that such the required processing is not available.

A RESTful representation of the get(ID) example is for ID to be a URL and an HTTP GET is the function that is used. The response would typically include the Content Type header and a browser (or other client) would process the content type appropriately. A RESTful representation of the create(IDofABCRecord) example is for a URL to identify the resource processing the create request and part of the URL or a parameter of the URL would indicate the resource type to be created. An HTTP POST function would be used and the structure of the payload of the request should match the indicated resource type to be created.

As an example more relevant to the VA, the order for an immunization vs. the order for an x-ray could use the same interface template but one use would indicate the immunization order structure was being used while a second use would indicate the x-ray order structure. The underlying capability doing the processing could use common processes where that was appropriate and specialized processing where needed. For example, the first step in the order could be confirming the authority of the requester, and this could be provided by common access to an authorization service. This approach avoids creating a separate authorization access for each variation of the order creation and could minimize necessary modification work if the access to the authorization service is changed.

A design consideration could be to see if creating entities other than orders could be accomplished through the same interface. For example, under what circumstances could a patient encounter record be created through a generic create interface by providing the patient encounter data structure as the basis for input? A more generic create service may be possible, but other considerations, for example, privacy concerns, may make separate services a preferred solution.

Granularity and Interface/Contract First design

Except for the simplest, dedicated interfaces, coarse-grained interfaces are more compatible with an Interface First design. This is because the details of the payload structure can change without changes to the interface and with very limited changes to the payload itself. Moreover, if the request identifies the structure of the payload, new payload versions can be inserted by merely changing the identifier of the payload structure.

[out of steam here but should be able to expand]