Introduction

This paper provides an opinion on service oriented architecture.

We have been living with the buzz word SOA, service oriented architecture, since 1998. This opinion piece will argue that SOA when applied to integration, that is integrating applications you already have, doesn't work. It will also argue that using SOA for new applications sort of works, but you'd probably be better off using other approaches, such as REST. To say that this opinion puts us out on a limb is an understatement. SOA is everywhere in our industry except one place, customer success stories. Here is what Anne Thomas Manes of Burton Group says:

"… I think I've become a bit jaded from the interviews I've conducted thus far. It has become clear to me that SOA is not working in most organizations."[1]

This opinion tries to explain some of the reasons why this might be the case, what we think you should be doing instead and even looks at a couple of success stories. The thesis of this opinion is that the arguments raised in favour of SOA by the marketplace to its customers are compelling, but they are specious. The Oxford English Dictionary defines specious as:

"superficially plausible, but actually wrong"

We are saying that SOA is wrong.

What is SOA?

Of course, if we are going to attack SOA we have to say what it is. And, of course, all those who think we are wrong will tell us that we are attacking the wrong SOA. Their SOA is much better. Our response is to say, show us how it worked for your customers and we'll agree. We have been working with a customer that has run aground trying to use SOA for integration. They were using the SOA reference architecture from the Open Group. So we will use the Open Group SOA reference architecture to define SOA. The Open Group diagram from http://www.opengroup.org/soa/source-book/ra/perspec.htm is shown below:

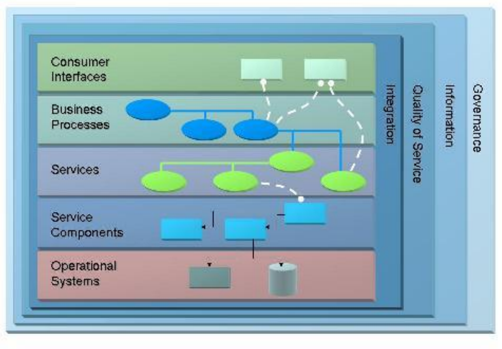

Figure 1 Open Group SOA Reference

This diagram is very similar to the IBM reference architecture shown below.

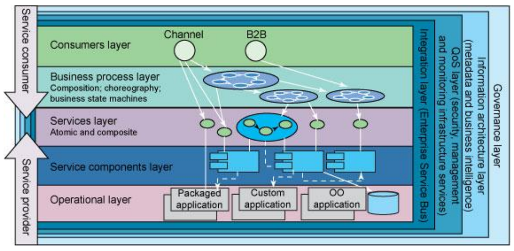
Service consumer
Service provider
Consumers layer
Channel
B2B
Business process layer
Composition; choreography; business state machines
Services layer
Atomic and composite
Service components layer
Operational layer
Packaged application
Custom application
OO application
Integration layer (Enterprise Service Bus)
QoS layer (security, management and monitoring infrastructure services)
Information architecture layer (metadata and business intelligence)
Governance layer

Figure 2 IBM Reference Architecture for SOA[2]

Both references rely on the concept of a service consumer and a service provider. The essence of this pattern is shown below:[3]

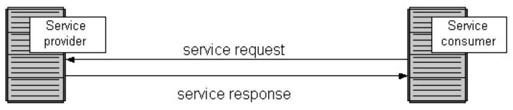

Figure 3 Definition of a Service

The IBM diagram makes it clear that a consumer can be either a person (that is, a person's agent) or another system. It is not explicit but clearly the service request is made first and the service response is received after. Also, there is no need for the request and response to be synchronous; they can be two one-way messages that are correlated in some way.

[1] From (Manes, 2008)

[2] From (Arsanjani, 2007)

[3] From http://www.service-architecture.com/web-services/articles/service-oriented_architecture_soa_definition.html

However, there is an implicit assumption that the consumer cannot proceed, in some sense, unless there is a response.

The Open Group and OASIS have been cooperating on the definition of SOA, so it is useful to include the OASIS definition:

"Service Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains."[4]

This is a very good definition of the purpose of SOA; we really do need a paradigm for organising and using capabilities under different ownership. In the rest of this document we shall use the Open Group diagram and the OASIS definition of purpose as our definition of SOA. To summarise, SOA is a way of organising and using capabilities under different ownership where the capability provides a service that is used by a consumer and may be mediated by workflow.

Transactions

Our first problem with SOA concerns transactions, which for us are the mechanism used for the capturing of business events. We will show that thinking about transactions imposes severe constraints on services, contrary to the naïve view of services.

In the environment of Figure 3 Definition of a Service, we need to ask the question: are the provider and consumer participating in a single transaction or in different transactions? The OASIS definition of 'different domains of ownership' answers this for us. If they are owned separately, they must (in general) be in different execution environments and so should be assumed to be in different transaction environments. It might be that sometimes, as a special case, they are in the same transaction environment, but in general not.

Given that we are assuming that the consumer requests the provider to do something in one transaction environment and the provider responds in another, it clearly makes a difference whether a transaction is needed. If the provider changes the state of some resource (a file, database, queue, or equivalent) then a transaction is needed. If the provider is just supplying information, then a transaction is not needed. This analysis generates the four cases summarised below:

Table 1 Characterisation of Service Interaction

		Provider	
		Transaction	No Transaction
Consumer	Trans action	Integration or Orchestration (1)	Autonomy (2)
	No Transacti	New application transaction (4)	New application request (3)

The case where both are transactional is dealt with in Orchestration and Integration below. The case where the consumer is transactional but not the provider is dealt with in Autonomy below. The two cases where the consumer is non-transactional are dealt with in Writing New Applications below.

Integration (1)

Here we discuss the case where a consumer requests a transactional service from within a transaction of its own.

We can implement a service with transactions at each end using a distributed two phase commit transaction. This is called the 'Managed, shared global transaction pattern' by (Booz, 2007). However, it is very difficult to get different transaction managers to distribute transactions. Even where it is possible to distribute the transaction, it is well known that this approach does not scale. For instance eBay give as a scalability best practice "Avoid Distributed Transactions" (Shoup, 2008). At best, this is a corner case for use when you have to, not a

[4] From (MacKenzie, 2006)

general way to architect interaction. In fact, we believe that ACID[5] is an inappropriate architecture for distributed systems, BASE[6] is the superior approach. Although it is admirable that Service Component Architecture (SCA) has taken the trouble to create a specification (Booz, 2007) for ACID behaviour, it is not supported by any existing SCA environments and, even if it were, would not be appropriate for reasons of reliability and scalability.

Our experience is that many of our clients do not appreciate this nuance - that calling a service implies a distributed transaction. In a current engagement we are seeing clients designing synchronous request/reply between the resource management parts of existing applications. It is worth noting that transactional messaging, which many businesses use, does not allow you to make this mistake. If you send a message to another system using MQ or JMS as part of a transaction, the message is not sent until the transaction has committed. This is, of course, perfect for event driven architectures. For those who naively think that a service can be called from a transaction it is tragic, because they are likely to deadlock their own transactions[7].

Another lost nuance is the difference between intention and causality. Sometimes one system needs another to do something before it can continue, such as an ATM issuing money, which has to wait for the bank to authorise the transaction. In this case, the first system intends to integrate with the second. Intention implies orchestration. Often, though, we just want an action at one service to cause an action at another. For instance, if I update my name and address in the mortgage application I want that to percolate through to the account management system, but there is no intention in the mortgage system to orchestrate the account management system. In fact, this update might go to many systems. Indeed, our current client is implementing a system that updates account data on five other systems when it itself makes an update. There is no need for the first system to wait for the others to complete. This is the integration use case.

SOA doesn't work for this use case. There is no need for an application to call services in other applications in order to integrate the two applications. For example, if you introduce a new sales order process using a package such as SAP, there is no need for SAP to orchestrate the applications in your warehouses that fulfil the orders. You just need to send the orders to the warehouse. If something happens in one application (say, we take an order) and that requires a consequence in another (say, we execute the order) then there is no need for orchestration. All we need is for a message to go from the first system saying 'I just took this order' which is transformed into a message for the second saying 'execute this order'. Effectively we are replacing swivel chair[8] integration with a message.

Orchestration (1)

Here we discuss the case where the consumer is a business service requesting the provider to do some work on its behalf. In this case, the consumer intends to have the provider do some work for it. For instance, when I take money out of an ATM, the device handler for the ATM has to get the bank account system to authorise the withdrawal.

The way it does this, without using distributed two phase commit, is the three transaction model - see the basics of information engineering (Schlesinger, Basics of Information Engineering, 2010).

[5] ACID stands for Atomic, Consistent, Isolated and Durable. A term coined by Andreas Reuter in the early 1980s.

[6] BASE stands for Basically Available, Soft state, eventually consistent. See (Pritchett, 2008).rag

[7] Deadlock occurs because the transaction waits for a response that cannot be received until the transaction commits.

[8] Swivel chair integration is where a person updates one system and then swivels on their chair to another screen and keyboard to update a second system.

This case, where one system makes a request of another using three transactions, is the base case of orchestration. Each activity in an orchestration workflow implies one such trio of transactions. In each case the consumer has to worry about errors, retries and compensation. The return message must include not just the expected reply, but all the possible application failure responses (such as 'authorisation denied') and all the middleware failure responses (such as 'service not found'). The consumer must set a time out and decide when to retry. In the case that the request succeeds but the consumer subsequently aborts, there must be another service for compensating the original service.

Note that when we implement orchestration, we are always implementing new business capability. This is because the consumer, by definition, cannot continue until the service provider responds. Therefore orchestration is never an appropriate approach for integration of existing applications. So the argument that orchestration helps you integrate is specious.

SOA is compelling because it offers the ability to orchestrate a set of services designed to do one thing so as to have them do another. But orchestration is much more complicated than the naïve advocate realises. Both the service consumer and the service provider must have been written to do the orchestration. There is no example we know of where one application orchestrates another, at scale, except where both were written for that purpose. The argument that it is possible to have more than one orchestration of a set of services, at scale, is compelling but specious.

Autonomy (2)

We come now to the case where both the consumer is transactional but the provider is 'read only', that is, there is no implied transaction at the provider. In this case, the consumer is requesting information from the provider that the consumer does not have. For instance, in order to decide whether to make a payment in a retail banking system, I might request the balance of an account from a deposit management system.

Leaving aside the problem of mapping the data received into my own view of data (see Semantics below), the act of making a synchronous request during a transaction (for example, the authorisation of a payment in the example above) delays the service and also makes it less reliable. Also, if the called service is not available, the transaction will fail.

To get around this reliability issue, as the consuming service scales, caching is used. This enables the consumer to get the answer without interacting with the provider.

However, this introduces the question of when the data is out of date, that is, knowing when the account balance has changed. The easiest way to solve this problem is to send a message from the provider to the consumer when a balance changes. If we do this, we never need to call the service.

So our argument against SOA is that one business service should never call another for read only, instead we should arrange for a message to go from the provider to the consumer informing when the data has changed. This is the essence of, for instance, financial market data where stock ticks are sent to all those systems that need to know the current price. Scaled stock trading systems never makes a request for a price.

Writing New Applications (3 and 4)

We now address the case in Table 1 Characterisation of Service Interaction where the consumer of the service is not transactional. It is useful, to make sense of this, to separate the application into two parts: a part that interacts with the user (the agent in Figure 6 Information System), from the part that manages transactions (the information resource in Figure 6 Information System).

If the consumer of a service is transactional, it must be the information resource part of some business service. Conversely, where the consumer of a service is non-transactional, it must be the user interface part of an application, or an agent acting on behalf of a person.

If we are making it possible for a user to access services that were not previously available, we must be writing a new application. If the service were already available, we would not need to be doing the work. If we are replacing a service built into the application with a shared service then we are again, writing a new application to replace the one we had before. Indeed, some companies we consult with have created shared services, and have then versioned the application to their customers while they move from the built-in to the shared service implementation.

Again, there is a serious constraint where the consumer is non-transactional and the provider is transactional. In order to maintain the consistency of the information, a single agent interaction can only call one transactional service. If the agent were to call more than one transaction then there is a possibility that the interaction would fail after doing one transaction but before completing the next. This would leave the two services inconsistent and the user would have no way of knowing what had or had not worked.

SOA offers the compelling argument that you can compose services from different applications to create a new service. This is only true, from a transactional point of view, if all the services run under the same transaction manager. SOA mistakenly frames the problem of service composition as a component problem, whereas it is actually a distribution problem (recall the OASIS statement of what SOA was for). The SOA argument is specious because in a distributed environment only one service in an interaction can be for update.

Semantics

Having dealt with transactions, we now deal with semantics. This generates a whole new set of problems for SOA. The problem of integration is not really a problem of interoperability, it is fundamentally a problem of semantics. In the naïve SOA approach one application can call a service in another without worrying about the semantics of the service being called, just as naïve SOA didn’t worry about the transactions. For example, SAP now has a set of services you can call using Web services, its ESA offering. Each service has a WSDL describing it. A program that calls such a service has to map the data elements in the WSDL into its own data elements. This mapping has to be right or data will be inconsistent between SAP and the calling application.

Getting the Composition Right

How do we ensure that the mappings make sense is the first problem.

Again, it is worth looking at Service Component Architecture to see how this problem is addressed there. The SCA specifications do not address this specifically; however (Caine, 2006) addresses the problem. He describes the problem in the context of composing trivial services for Area and Weather:

“At the data processing level the elements of these different messages could be the same data type e.g. decimal for XML schemas would be xsd:decimal. Without composition validation it would be possible to combine Area and Weather in a service composition that has no real meaning.”

He proposes taking the standard SCA approach

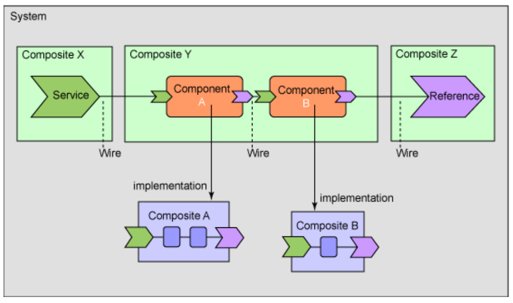

Figure 4 Service Component Architecture

And enhancing it with semantic

constraints

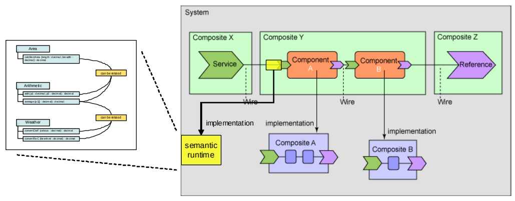

Figure 5 Enhanced Service Component Architecture

These constraints make sure that you map data elements that actually match.

Of course, there are no implementations of this available, and there are no examples of services specifying their constraints. For instance, SAP ESA provides no such metadata. Nor does SCA make it clear that translation is required even though the namespaces of the WSDLs defining the services are different from the name space of the composed service. SOA makes the compelling argument that you can compose services to create new programs, but this argument is specious because it ignores the need to get the semantics of the composition right. Naïve customers compose services inappropriately and create inconsistent and bad data.

Tight Coupling

By using the SAP WSDL in your program to call the service, you are tightly coupling your application to SAP. Your program has compiled the SAP interface into its code. If anything changes, you have to recompile or the service invocation is broken.

We have long known that the way to break tight coupling of this type is to introduce two transforms. The first transforms from the semantics of the service consumer to a neutral intermediate form. The second transforms from the neutral form to the form of the service provider (Schlesinger, Integration Architecture, 2010).

SOA provides the compelling argument that services can be composed but fails to point out that using the interface of the provider in the namespace of the consumer creates brittle tight coupling. This can be mitigated by using two transforms, but SOA architectures like WS-* and SCA ignore this, as discussed above. The argument is specious because all the benefits that are supposed to accrue from the sharing of the service are lost if everyone sharing the service has to use the same interface in their code.

Ownership

Remember the OASIS definition of SOA:

“Service Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains.”

We now look at some of the implications of those telling words, ‘different ownership’. What this means is that the service consumer may be separately owned from the service provider. For instance, the service consumer might be in HR and the service provider in Finance. Similarly, the consumer could be in a customer company and the provider could be in a supplier company. This means that there is no overall owner of the consumer to provider relationship. In order for the provider to provide the service, there has to be something in it for the provider or it wouldn’t bother to provide it. Quoting Anne Thomas Manes again, from the same article:

“More to the point, the techies have not been able to explain to the business units why they should adopt a better attitude about sharing and collaboration--which is the fundamental cultural shift required for SOA to succeed. The pervasive attitude is "What's in it for me?" As one of my interviewees said, "Altruism is not an enterprise strategy"”

SOA offers the compelling argument that, if an enterprise only offers 50 services, there should not be a need for 2000 applications to provide them. However, the fact that the IT crowd can see that there is no need for the same service to be implemented in two different business units, does not imply that these two units should share a service. Business systems are owned by business owners, not by IT. This is at the heart of our One Level approach (Schlesinger, One Level

Enterprise, 2010). Just because there could be a single service for a capability does not mean that the business has to own it that way. Sharing the service would require the business to reorganise to assign that capability to a single business unit. The owner of that unit would then assume ownership of all the services that implement that business capability. That owner might then decide to provide a single shared service to all the consumers. However, there might still be good reasons why the business owner might provide multiple services. For instance, the service might be say an insurance industry standard for some users, but others might use a banking industry standard for the same service. Alternatively, it might make sense to keep two implementations if you have two brands, in case the brands separate (like Shell and BP did in 1976).

The SOA argument about sharing services is specious because it forgets that business services have business owners and they are organised the way they are for business reasons, not to make IT more efficient.

Fundamental Confusion

We believe that there are two fundamental confusions in SOA. The first is the confusion between developing new applications, on the one hand, and integrating applications on the other hand. The second confusion is a category error[9] in the way SOA distinguishes process workflows from applications. In figures 1 and 2 above, business processes are shown as something different from operational systems. This is true at design time, but not at run time. Once a business process is deployed it becomes an operational system like any other.

All the problems we see with SOA stem from these two confusions. To clear them up consider what an information system is. It is the interaction of a person with a business system as shown in Figure 6 Information System.

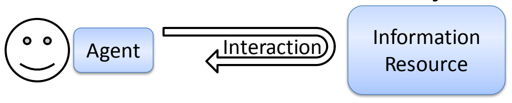

Figure 6 Information System

The essence of service orientation is evident here – the separation of the human interaction part of the application from the resource management part. This we agree with, indeed it is at the heart of our Inside Outside approach (Schlesinger, Inside Outside, 2010). When a person interacts with an information system there is always a request and a reply. So this is indeed a service as shown in Figure 3 Definition of a Service. However, when we introduce a second person and business system we get two new kinds of interaction as shown in Figure 7 Multiple Information Systems.

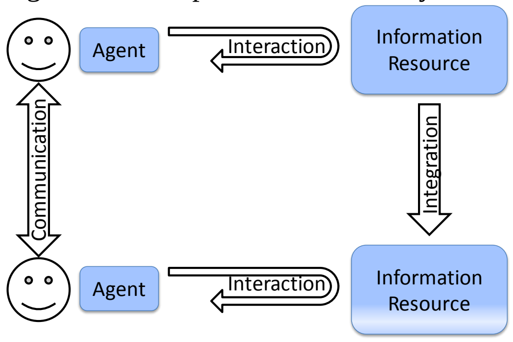

Figure 7 Multiple Information Systems

The diagram shows that each person interacts with an information resource to create an information system, but that, in addition, the information resources can interact with each other and the people can interact with each other. The interaction between the information resources is integration, the interaction between people is communication.

If we enable the agent for a person to access a new information resource, then we are in effect writing a new application.

[9] From http://en.wikipedia.org/wiki/Category_error
"A **category mistake**, or **category error**, is a semantic or ontological error by which a property is ascribed to a thing that could not possibly have that property."

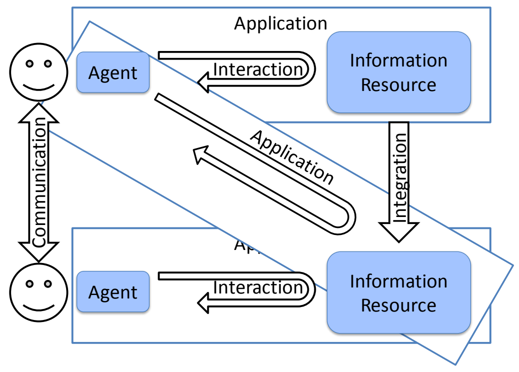

Figure 8 New Application

The idea that creating an information resource for sharing is an example of integration, which we see as central to SOA, is a category error. Actually, integration does not change the capabilities you have; it just makes it possible to have an event in one application cause an event in another. Whereas people use request-reply interaction, integration is always a one-way message. Whereas request-reply can be for information (read only) or for transaction (write and update), integration is only ever for transaction.

Writing an application by mistake is still the number one mistake you can make when doing integration and SOA applied to integration almost forces you into this error.

Conclusion

In a speech in the House of Commons on 11 November 1947, Winston Churchill said:

“No one pretends that democracy is perfect or all-wise. Indeed, it has been said that democracy is the worst form of government except all those other forms that have been tried from time to time.”

This might be true of SOA. It might be that, for all its faults noted above, SOA is the best of a bad lot. We do not think so. We believe that business events provide a much better architecture for integrating the enterprise. We describe how this works in (Schlesinger, Integration Architecture, 2010). In our experience event based integration is far quicker to implement, far more robust, far cheaper to build and own and far easier to scale. SOA is a compelling but specious argument.

Works Cited

Arsanjani, A. (2007 , March 28). *Design an SOA solution using a reference architecture*. Retrieved September 30, 2010, from IBM Developer Works: http://www.ibm.com/developerworks/library/ar-archtemp/

Booz, D. (2007, December 3). *ACID Transaction Policy in SCA*. Retrieved October 1, 2010, from Service Component Architecture Specifications: http://www.osoa.org/download/attachments/35/SCA_TransactionPolicy_V1.0.pdf?version=1

Caine, J. (2006, October). *The Open Group*. Retrieved October 1, 2010, from Service Domain Spaces v1.0: https://www.opengroup.org/projects/si/uploads/40/13836/Service_Domain_Spaces_v1.0_SWESE2007.doc

MacKenzie, C. M. (2006, October 12). *Reference Model for Service Oriented Architecture 1.0*. Retrieved September 30, 2010, from OASIS: http://docs.oasis-open.org/soa-rm/v1.0/

Manes, A. T. (2008, March 9). *Looking for SOA Success Stories*. Retrieved September 30, 2010, from Burton Group Blogs: http://apsblog.burtongroup.com/2008/03/looking-for-soa.html

Pritchett, D. (2008, July 28). *Base an ACID Alternative*. Retrieved October 06, 2010, from ACM Queue: http://queue.acm.org/detail.cfm?id=1394128

Schlesinger, J. (2010, July 22). *Basics of Information Engineering*. Retrieved July 22, 2010, from Atos Livelink CIO Advisory: https://km.atosorigin.com/livelinkdav/nodes/45275526/Architecting the Enterprise/Basics of Information Engineering.pdf

Schlesinger, J. (2010, July 22). *Inside Outside*. Retrieved July 22, 2010, from Atos Livelink CIO Advisory: https://km.atosorigin.com/livelinkdav/nodes/45275526/Architecting the Enterprise/Inside Outside.pdf

Schlesinger, J. (2010, July 22). *Integration Architecture*. Retrieved September 1, 2010, from Atos Origin Livelink: https://km.atosorigin.com/livelinkdav/nodes/53601095/Integration Architecture.pdf

Schlesinger, J. (2010, July 22). *One Level Enterprise*. Retrieved July 22, 2010, from Atos Livelink CIO Advisory: https://km.atosorigin.com/livelinkdav/nodes/45275526/Architecting the Enterprise/One Level Enterprise.pdf

About John Schlesinger

John Schlesinger is a Principal at Atos Consulting where he leads its Enterprise Architecture practice. John is an advisor to enterprises specialising in middleware and integration architecture. He has lead integration architecture development in retail banks, investment banks, retailers and manufacturing, both for integrating applications and for integrating information.

John has worked both as a consultant and also as a developer with software companies. He has taken over two dozen program products to market at IBM, Information Builders, One Meaning, SeeBeyond and iWay Software. These products included the world's most successful commercial software (CICS) and the world's most successful data middleware (EDA/SQL). John also led the Architecture department at Dun and Bradstreet when its IT department went global.

A member of the ACM and the IEEE, John has an MA in Physics and Philosophy from Oxford University and a Post Graduate Diploma in Software Engineering from Oxford University.

John has spoken at numerous conferences including the CIO Cruises run out of New York, during one of which he was the first speaker on after the collapse of the World Trade Towers in 2001.

John can be contacted at john.schlesinger@atosorigin.com