lustratus

Research

Best of Breed API Middleware for Core Systems

What tools are needed to API-enable your legacy systems?

Author: *Steve Craggs*

April 2018

Version 1.00

Sponsored by

ADAPTIGENT

lustratus

Table of Contents

Executive Summary

In today’s IT marketplace it is easy to think that core systems are being marginalized in companies across the world. Indeed, not so long ago the story was that core systems were the dinosaur, headed towards global extinction. This forecast has, of course, been thoroughly disproved, as as many companies continue to gain great business benefit from their legacy system investments. The strategic thinking has evolved; many companies that were developing strategies to move off these systems now realize it is much more effective to keep them and their core applications at the heart of the business while building new capabilities around them As technology has continued its breakneck speed of change, the world of the ‘connected mainframe’ is very much here. Companies can continue to benefit from these core systems' unsurpassed levels of availability, scalability and performance while gaining all the advantages of leveraging new channels, markets and opportunities.

> *“The strategic thinking has evolved; many companies that were developing strategies to move off core systems now realize it is much more effective to keep them and their core applications at the heart of the business while building new capabilities around it.”*

The most recent demonstration of this shift is the emergence of the so-called API model, which enables the aggregation of a diverse set of IT assets in order to deliver business services that support operations more effectively, with minimal effort and without the need for specialized skills. Essentially, system of record business activities are made available externally through the use of APIs that can then be embedded into phone apps, web pages, chips and any other desired delivery channel.

Companies with large legacy systems investments will immediately appreciate the benefits of the API model. It has similarities to the service-oriented architecture (SOA) movement, but with the major difference that APIs require far less skill to use and lend themselves to rapid development. At a stroke, years of core systems investments become accessible in a relatively simple way to all areas of the business, and indeed, a host of new applications also become accessible to your legacy system, allowing it to take full advantage of technological developments such as social markets, phone apps, tablets and the Internet of Things (IoT). These different technology investment areas can feed off each other, creating maximum value and improving the return on assets.

However, no change is without risk. Executives of companies that rely on legacy systems are often uneasy about allowing them to merge with the rest of the IT structure. For example, concerns abound about preserving service levels and maintaining security and integrity, and there is also a general feeling that extensive retraining is going to be needed. The reality is that generic tools developed without considering special core systems needs will not do the job effectively. Success or failure with an API model will be governed, to a large extent, by the effectiveness of the tools to support this highly specialized environment, and so companies should look for toolsets that are specifically oriented to core technology API enablement, deployment and operations.

This paper considers the topic of API enablement for core systems, and identifies the best-of-breed characteristics to look for in evaluating any API middleware toolset, in order to help companies make the decision that will best suit their individual needs.

Introduction

Despite years of predictions of their demise, many companies still have millions of dollars invested in their IBM core technology solutions, encompassing application code, skills, scripts and general working practices and procedures. Even with the wide range of modern technology, they still offer an ideal platform for business critical computing, offering scalability, reliability and predictability with the operational efficiency of a 'single system' perspective.

Rather than taking the high-risk approach of replacing them and thereby throwing away these huge investments, most legacy systems users are instead looking for ways to increase the return on these assets by placing them squarely at the centre of the modern computing world. This strategy offers the possibility of leveraging the existing portfolio, while at the same time exploiting the advantages of different technologies and delivery channels, maximizing opportunities for business productivity, efficiency and overall success.

One of the main challenges, however, is to bring the different technology worlds together while at the same time maintaining enough separation to avoid contamination issues. For example, opening up access to selected legacy system applications from social marketplaces may deliver a huge increase in market reach and broadening of product offerings, but if it comes at the expense of compromising the legendary integrity and reliability of the core system environment, then the price is unlikely to be worth paying. Then there is the skills problem; trying to find developers who are equally expert in legacy systems programming and later technologies such as JSON, OAuth and NodeJS is going to be extremely difficult and expensive. Surely there must be a way to keep the core technology and web-based worlds separate but connected in such a way that these risks are minimized?

"Then there is the skills problem; trying to find developers who are equally expert in core technology programming and later technologies"

Service oriented approaches emerged as one way to try to address this integration issue. The concept was to enable business application code, processes and data to be assembled into 'services' that can be called externally to execute a standalone business function, such as 'Place Order'. When the 'Place Order' service is called, the tooling 'orchestrates' the flow between the various components to replicate the order process to provide the desired result. However early architectures such as SOA were very standards-based and formal about the way these services are driven, and as a result building the front end calls to these services is quite skills-dependent. At the same time, companies were realizing that new platforms like phones, tablets and the Internet of Things could offer new delivery channels for new solution types. The digital marketplace has started to emerge, and one key characteristic is the speed with which it moves. New application innovations need to be seized on quickly in order to compete. A faster, simpler way was needed to enable developers to build solutions that consume these services.

Enter the API model. The API model is based around modern linkage techniques like REST where the call to drive some back end service is as simple as what programmers would generally think of as an API (Application Programming Interface) call; hence the name. The API model has in turn spawned the API Economy, a digital strategy covering an ecosystem of suppliers and developers where developers can rapidly pick up APIs, for example from a social exchange or marketplace, and quickly bring new solutions to market. It worth noting in passing that although the API model may look completely different from other service-oriented approaches, the same concept remains of having a back-end service that can be driven externally; it is just the method of achieving this that is different.

The API model offers an ideal opportunity for core systems users to bring their existing assets into play as part of broader, multi-channel solutions, but the absolute key will be to API-enable core systems with the right tools in place, to provide the protection layer to prevent any contamination while at the same time offering the speed of delivery expected in modern business operations. This paper will focus on some of the key aspects of these critical tools and the respective best of breed characteristics.

The API Architecture

Although more and more companies are becoming familiar with the API model, there are plenty of opportunities for confusion. This is particularly the case for some core systems users who are used to thinking about ‘API’ in a different sense. Therefore, a quick recap is in order.

What is an API?

The key of the API model unsurprisingly, is the concept of an API. Much confusion stems from the fact that a term familiar to many programmers is being used to cover a similar but distinct concept. In core systems, a CICS or IMS programmer tends to think of ‘API’ as the Application Programming Interface to be used to call a particular function provided by the CICS or IMS platform, the database, the operating system or whatever. In other words, it is one or more lines of code, supplying parameters as laid out in a specified ‘API’, driving the desired function in the underlying system software. A legacy systems programmer is likely to view an API call as a piece of distinct, technical functionality that is only a tiny part of a larger business application.

The API in API Economy terms is in principle not dissimilar in that it is a way for one component to call some sort of activity provided by another, frequently in the sense of a front-end component driving a backend ‘system of record’ or ‘legacy’ one. The driver of the service is referred to as the API ‘consumer’, and the provider of the service as the API ‘provider’.

The first clue to a difference in usage is right there; rather than executing a technical command such as reading from a database, this usage of ‘API’ refers to the driving of a discrete piece of business functionality, or ‘service’. The big issue for core systems users is that this ‘service’ may well not exist in isolation but be part of a larger application. A major part of the work to API-enable core systems is actually to define these services and orchestrate the relevant application parts to deliver it on request. Think for example of a ‘place an order’ service. In legacy system application terms, placing an order may just be one particular menu in the order entry interface; in order to be usable externally, work will be needed to carve out this function and give it a programmable interface.

Once core systems functionality has been suitably packaged into a service, the next challenge is to make this programmable interface accessible in whatever fashion the service ‘consumer’ wants. Perhaps this will mean enabling it to be called in a RESTful fashion, or through a SOAP message, or through some other mechanism. This is where things get really tricky for core systems users, because this usage must be enabled without compromising the values to which the company has become accustomed, such as security, integrity, reliability and scalability.

Once the mechanism has been provided to call the service, the calls need to be built in to whatever the delivery channel needs for the use in question. This might be as a widget on a web page, a phone App or perhaps an automatically triggered link from a chip in a car or some other intelligent device in the Internet of Things. This part is usually best carried out by developers specializing in the relevant environment, for example phone App developers, because the skills requirements are likely to be completely different to core systems ones. The diagram below provides a simple illustration of a core systems API to hook price quotes into a phone-based price comparison App.

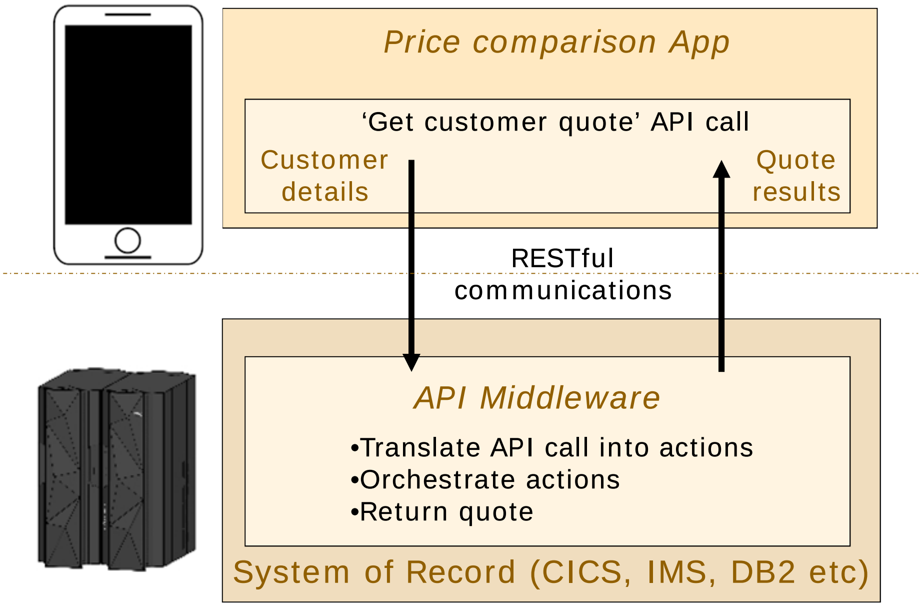


Figure 1: An illustration of an API call

It is worth noting that in the API Economy, 'API' can sometimes be assumed to encompass both the consumer and provider parts, although API providers normally regard the term as referring to the services that they choose to make publicly available for external consumption. Perhaps more accurately, the back-end work and the connectivity mechanism to drive it tend to be referred to as 'API-enabling' the business systems. This paper will not spend much time on the front end challenge of building the calls into the Apps or widgets or whatever, but instead will concentrate on the issues of API-enabling core systems, to make selected pieces of business functionality publicly available for building into new solution types that can be delivered over a wide range of channels.

Implementing an API Architecture

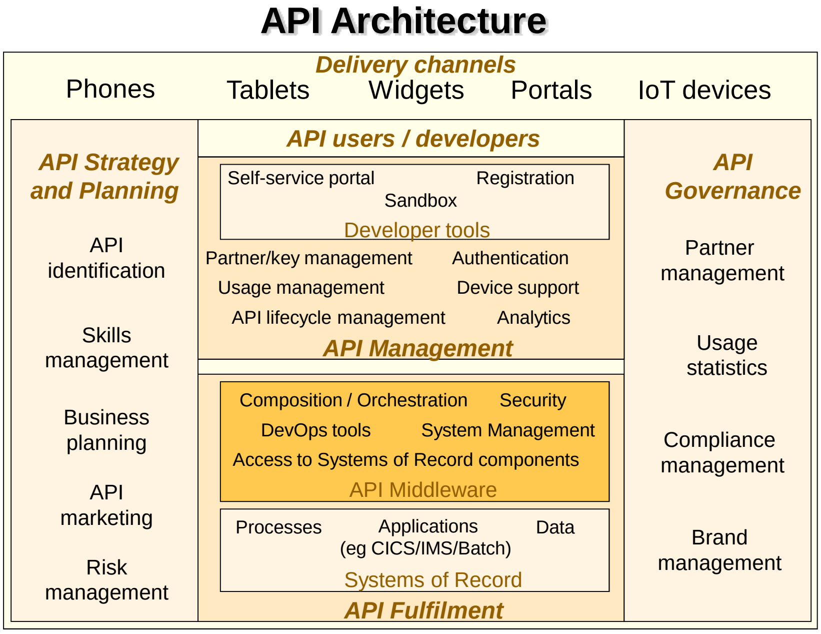
API architecture is really a natural evolution of early service oriented architectures that came to prominence in the last ten or fifteen years. It retains the basic concept of some sort of 'back-end' business service that can be driven externally, but the API approach carries the independence of the service consumer further. Previous approaches depended on relatively technical and complex programming to invoke a back-end or legacy service, making it something that was normally done in-house, within the same company boundaries as the business services themselves. By contrast, in the API world, while the building blocks for the services are still provided by the business service owner, the consumers of those services are often components built by third party developers, possibly with specific device expertise for example, with the invocation often being as simple as a call to a URL. Since they are easier to use, developers can go to online marketplace to view available business services and build them in to the solutions they are providing. The API approach has much less restrictive skills requirements and offers greater opportunities for flexibility and innovation.

However, even the simpler and more flexible API approach has specific challenges that must be addressed. For example, once the service request has been passed to the service provider, the service provider still needs to handle all the necessary activities such as managing security, controlling traffic volumes, orchestrating the legacy components to deliver the requested service and delivering the results. These are just the practical challenges of course; there are also the usual challenges around what functionality a company is prepared to expose externally and to whom, and how to keep any external activities from interfering with internal systems of record.

An API architecture is therefore an essential requirement for successful, enterprise-class API enablement, and this is particularly important for core systems users who rely on their enterprise-class reliability, scalability, security and performance. It is worth spending a few moments considering what types of functionality and supporting activities will be required to deliver a successful API deployment. These include:

- Support for a wide range of delivery channels (e.g. phone Apps, IoT chips)
- An environment to attract and enable API-based solution developers
- An API middleware layer to make desired and authorized business functionality available to API consumers safely and reliably
- Strategy and planning activities to make the optimal set of APIs available
- Governance activities to manage partner involvement and to ensure business cases are met

The diagram below illustrates the make-up of a generalized API architecture; the specifics of core systems API architecture are discussed later.



Source:- Lustratus

Figure 2: The API Architecture

Having clarified what sort of architecture is required to succeed in the API Economy, the next area to tackle within the overall theme of this paper is the applicability of the API Economy to a legacy systems environment.

Why API-enable Your Core Systems?

The reality is that API-enabling core systems is becoming a key topic for most major companies – indeed, IBM itself now places the API model firmly in the legacy systems world as an important and relevant development. There are a number of reasons for the appeal of the API model to companies who rely on legacy systems. The benefits that attract these companies can be summarized as:

- Improved return on assets
- Wider and deeper market reach
- Faster time-to-market / increased agility
- Opportunities for new revenue streams
- Mitigation of disintermediation

The first point has already been touched upon. Over the years, companies have invested heavily in their core systems, and financial executives in particular are keen to ensure that these investments bring the maximum possible returns. But these assets in general are fairly difficult to access from the outside. There are connectivity issues, syntactic and semantic issues at the invocation level, and a huge skills chasm between core systems and other IT staff. An API approach offers a way to overcome these issues. It addresses the connectivity and invocation problems, and cunningly bridges the skills chasm by enabling each skills group to concentrate on developing services. This is a key point – instead of telling a COBOL programmer that he has to work with OAuth and JSON, or a phone App developer that she must work with COBOL, each person is enabled to develop in his or her own environment.

> *“One of the main reasons for creating APIs is to make them available to solution developers working in modern delivery environments. By enabling these developers to rapidly build new solutions that bring business to the company through APIs, innovation is greatly accelerated”*

One of the main reasons for creating APIs is to make them available to solution developers working in modern delivery environments. By enabling these developers to rapidly build new solutions that bring business to the company through APIs, innovation is greatly accelerated. Whether the work is done by third party developers or in-house departments, new solutions can be quickly brought on line, delivering new channels and ways for customers to buy. New revenue streams may be created by offering an innovative new solution to customers and consumers and companies can respond much more quickly to both new opportunities and threats.

It is even possible that an API approach can mitigate the threat of disintermediation. By providing APIs to drive business activities as close as possible to the buyer, it reduces the risk of some other party getting into the gap and cutting the provider out.

All these potential benefits support the crucial importance of the API model for core systems users.

API Middleware for core systems users

Having recognized the potential value of the API approach for core systems users, before moving on to general considerations, it is worth highlighting the key section in the centre of the API Architecture diagram above; the API Middleware layer. In essence, the API Middleware layer plays a similar role as middleware plays in other IT solutions. It sits between the client level and the systems of record, translating the desires of the client into execution within the core systems of record.

Typical roles of the API Middleware layer are:

- Provide a connectivity bridge between the requestors and the back end systems of record
- Handle any format and mapping requirements between differing formats and protocols at either end
- Orchestrate the necessary back-end components to deliver the requested business service
- Securely authenticate and protect usage of the systems of record layer
- Satisfy systems management, security, analytic and audit requirements for proper governance

In some API architecture implementations, the API Middleware layer is fairly minimal. This is the case for example where the ‘back-end’ systems of record are already packaged as programmable services, perhaps accessed through RESTful interfaces. Indeed, for these simpler environments a generic layer of API Middleware is sufficient to meet most needs, and for this reason it is common for API Management tools such as Apigee API Management, IBM API Connect, Red Hat 3Scale and CA API Management to include a generic subset of API Middleware in their offerings. This generic layer typically supports simple web service and SOAP calls and sometimes provides some limited level of orchestration support.

However, for core systems users the API Middleware layer is absolutely key. Many applications, services and processes will not be available through a simple call interface. A specific layer for core systems will be needed to handle all the complicated resources like 3270 applications, CICS and IMS transactions, databases and corporate systems of record processes. This core systems specific layer of API Middleware, will be critical for delivering a successful API-enabled environment while mitigating the inherent risk. The diagram below indicates how the API tools for core systems, such as the Fabric from Adaptigent and IBM z/OS Connect relate to the generic API Management tools mentioned above in terms of the basic architecture.

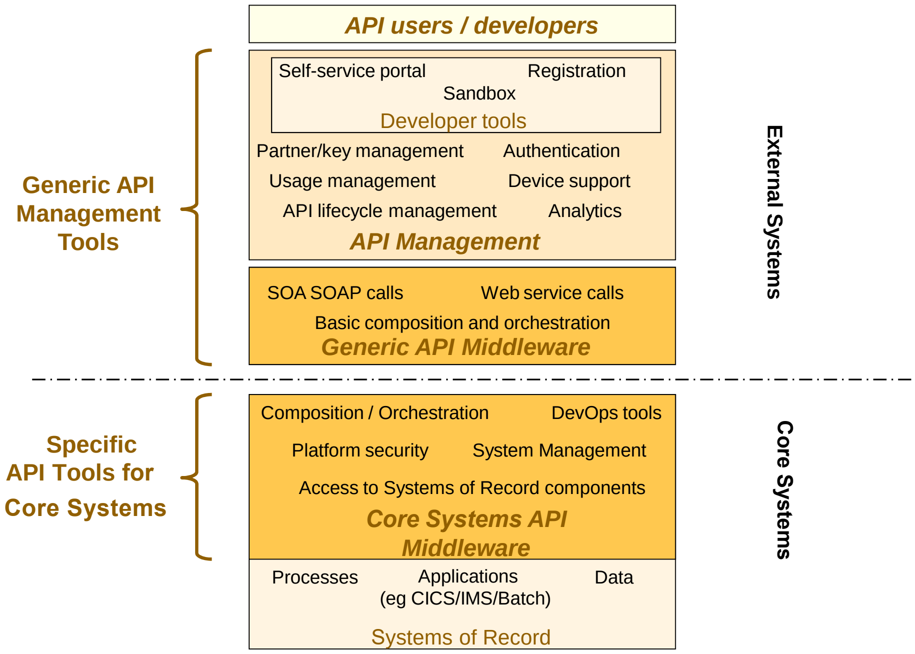


Figure 3: API Middleware packaging into generic and legacy system-specific layers

It is worth mentioning the Enterprise Service Bus (ESB) category of products here too. While ESBs are mostly about the join between on prem and external systems, some of the suppliers such as IBM, Oracle, TIBCO and Mulesoft also offer a limited set of generic and legacy specific API Middleware.

Core Systems Considerations

Before looking at the API Middleware for core systems layer in more detail, there are a number of other special considerations that must be taken into account. On the one hand, there are a number of technology-related factors that are either unique or particularly relevant to core systems, and on the other there is a considerable body of experience built up in integration projects of the past. The idea of integrating core systems more widely is not new; it has gone through numerous iterations including messaging middleware, ESBs and SOA before arriving at APIs. The lessons learned can drastically shortcut the effort to API-enable core systems while increasing the likelihood of a successful project.

Technology-related factors

There are four main categories of technology-related factors that users should consider when embarking on API-enabling their core systems:

- Applications and resources
- Environment
- Unique core system attributes
- Legacy systems skills

Most systems of record embody applications, environments and resources that are alien to those not steeped in core systems tradition. The IBM transaction processing products, CICS and IMS, provide a complete environment in which to run high volumes of transactions, reliably and effectively. CICS is ubiquitous, used by almost all legacy systems establishments, while IMS is more specialized but heavily used in the finance industry in particular. Non-IBM products such as CA-IDMS, CA-Datacom, CA-IDEAL, Natural and Adabas are also quite common. In programming terms, COBOL is by far the most popular language, although PL/1 has its fans. The DB2 database system and the WebSphere MQ messaging middleware may be a little less inscrutable to outsiders due to their existence on other platforms, but other system facilities such as RACF and SAF are largely unknown outside of core systems. So any toolset designed to API-enable them must be able to address the needs of these specialized resources and environments.

To some extent when API-enabling core systems, technology can shield the rest of the world from these specific products and environments, but the greater challenge comes in meeting expectations in terms of unique core systems attributes. Companies that rely on legacy systems for much of their business have come to expect a range of benefits from their implementations. These benefits accrue in areas like reliability, robustness, scalability, performance, security, integrity and manageability. The problem is that when services are delivered within the API model, client-side components in particular will be running in a wide range of environments and technologies, each with its own associated characteristics. The risk is that core systems users are used to a high level of service quality based on innate system capabilities, and this quality of service could be jeopardized by influence from other technologies like phones, tablets and chips sitting inside household appliances or cars. For example, while a core systems user will typically run their workstations or other devices in at least a semi-secure environment, a phone user may well leave the phone unattended for a while, quite possibly in a public place. Any tools or technology involved in API-enabling core systems must take these sorts of factors into account.

Regarding skills, as discussed above, it is likely to be difficult and expensive to find IT developers who are comfortable programming in both legacy and mobile environments. Therefore, any toolset for enabling core systems should make it easy for all programmers to quickly create API components and services without the need for expensive and time-consuming re-education.

"Regarding skills, as discussed above, it is likely to be difficult and expensive to find IT developers who are comfortable programming in both legacy and mobile environments"

All of these environment-specific factors must be taken into account in evaluating best-of-breed tools for API-enablement of core systems.

Learning the lessons from past integration projects

The API model has definitely become a major consideration for a growing number of companies across the world. As discussed previously, the API approach has particular attractions for legacy systems users. However, past attempts at integration have typically run into a range of problems, and today there is a much greater understanding of the specific issues to take into account before embarking on a business services-based legacy system integration strategy such as API enablement.

A number of the lessons learned reflect directly back to the technology-based considerations just discussed. But one issue in particular stands out – that of **core system business service composition**. The idea of a business service is the cornerstone of numerous core system integration initiatives and was mentioned in the introduction to this paper, but as a reminder it refers to the need to provide discrete business functions that can then be accessed externally, for example through APIs. If a phone App needs to be able to get an accurate product price, for instance, then it has to have some mechanism to drive whatever applications and data components make up the 'get a price' process on legacy systems.

A common difficulty stems from a collision between the purist world of the systems architect, and the pragmatic needs of operational service quality. Companies looking to open up their core systems and leverage them across other environments often see a pure, clean architecture where every business activity is packaged as a business service and all these services exposed through APIs. This is a great ideal, but can be disastrous if implemented without due consideration. The main issue is that, given the number of core systems transactions in existence, there is a danger this approach will result in a huge number of low-level services being created, for example 'get customer details' or 'check service history'. This may seem very logical, but in reality the danger is this exports design issues to the API developers. An App developer working on a new phone-based digital marketplace wants to be able to drive a 'product quote' process; the App developer now has to work out which low-level services are needed and in what process flow to deliver the final price.

> *"An App developer working on a new phone-based digital marketplace wants to be able to drive a 'product quote' process; the App developer now has to work out which low-level services are needed and in what process flow to deliver the final price"*

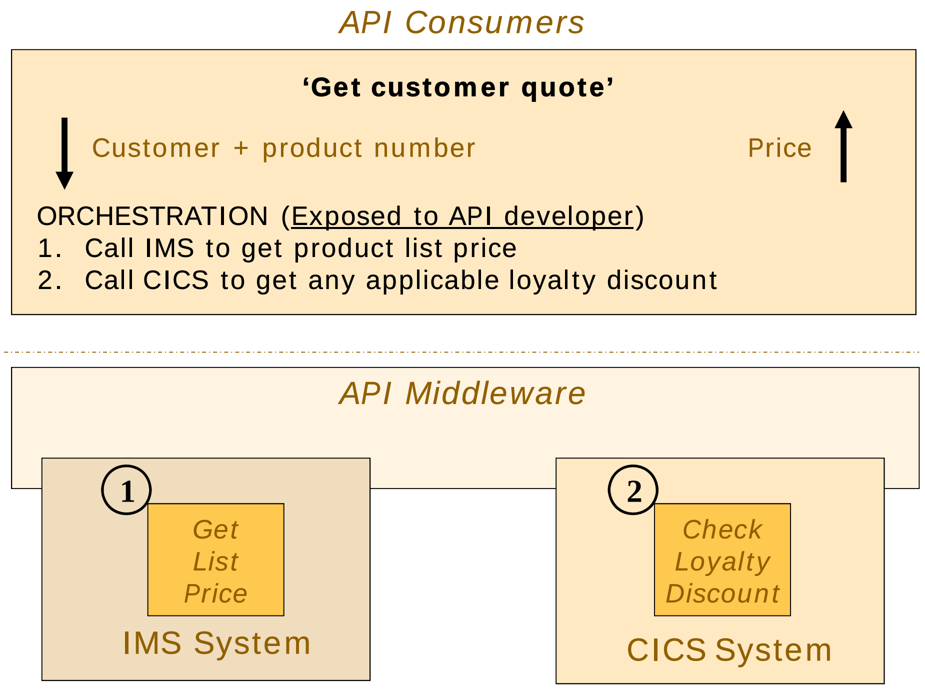

API Consumers
'Get customer quote'
Customer + product number
Price
ORCHESTRATION (Exposed to API developer)
1. Call IMS to get product list price
2. Call CICS to get any applicable loyalty discount
API Middleware
1
Get List Price
IMS System
2
Check Loyalty Discount
CICS System

Figure 4: Excessive granularity requires procedural knowledge for the API developers

Contrast this approach with a more considered one, where a higher level 'Find Customer Details' API is implemented. The consumption of the API has been de-skilled, removing any need for the API solution developer to have any knowledge of internal processes and implementation details.

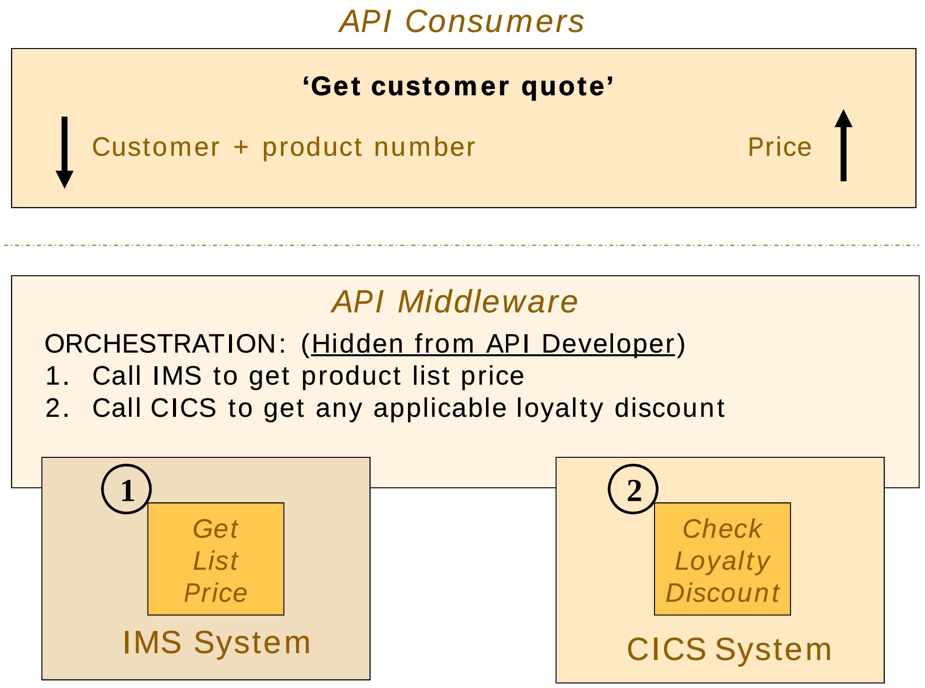

API Consumers
'Get customer quote'
Customer + product number
Price
API Middleware
ORCHESTRATION: (Hidden from API Developer)
1. Call IMS to get product list price
2. Call CICS to get any applicable loyalty discount
1
Get List Price
IMS System
2
Check Loyalty Discount
CICS System

Figure 5: Getting the granularity right insulates the API developers

Note however that the 'many small services' approach can work if the right API middleware layer is present. If a company chooses to implement a design where every discrete business operation has a corresponding service, the API middleware can perform the necessary orchestration of all the lower level services offered by the systems of record to present the API developer with a simple high-level API. It turns out that the API middleware is the key to the whole issue, because provided the middleware enables services to be composed into APIs that satisfy the API developer skills and needs, it doesn't really matter whether the packaging of those services (access, orchestration, data formatting etc) is carried out by the middleware alone or combined with other business service initiatives within the legacy system platform such as BPEL or BPM.

In short, defining the optimal level of granularity:

- Decouples the API developers from the implementation details of the operation
- Ensures that APIs meet the business need more closely
- Keeps the number of APIs and related definitions under control
- Reduces the development effort required
- Optimizes performance and network load by limiting the trips to and from the system of record

In fact, core system integration user experiences generally show that a good guideline is to avoid imposing too much of the API model on the environment. Legacy systems are different to other platforms; data is often in proprietary formats, XML is almost never used, the skills set is highly specialized and expectations of performance, scalability and reliability are much higher. Therefore, the key to API enablement success in regards to core systems is to implement only those APIs that are required to achieve company goals. The API middleware should handle as much of the packaging and managing of the various systems of record components as possible, to keep the APIs presented as simple and easy to use as possible.

> *"The API middleware should handle as much of the packaging and managing of the various systems of record components as possible, to keep the APIs presented as simple and easy to use as possible"*

API Middleware for Core Systems

Having set the framework for API enablement for core systems, it is now possible to focus on the key section of interest, the core system API Middleware component. As a reminder, many API Management vendors include some basic generic API Middleware in their offerings, but the focus here is on the core system API Middleware layer.

Development tools for building external API-based solutions for consumer applications are generally independent of whether back-end systems are legacy systems or not, as discussed earlier. Instead, it is the API middleware that is the critical differentiator for these users. There is a basic set of functions required to enable API enablement at the purely mechanical level, and then a range of best-of-breed characteristics that can be used as a checklist to judge relevant differentiators in any API middleware selection. In other words, every toolset for API enabling core systems has to include the basic functionality, but the support for the best-of-breed characteristics will depend on the particular vendor concerned.

Basic functions

At a minimum, API middleware for core systems must include the following basic functionality:

- Programmable access to applications on core systems
- Basic orchestration to execute API calls spanning multiple components
- Wrappers / Adapters to provide a standard invocation interface

Essentially, this list covers the ability to present APIs to API solution developers in a reasonably standard way. Regarding programmable access, as mentioned in the previous section, legacy systems have specific application environments that control execution of online transactions. The most prevalent is IBM's CICS, used by almost all customers, and any basic API middleware toolset must at the very least address the

CICS transaction. Modern CICS applications are usually designed in such a way that they can be driven externally through a programmable interface, using the COMMAREA in conjunction with the LINK function to provide input to and execute the particular application. For applications that fall into this category, the API middleware can fairly easily enable them to be driven from outside of CICS. Similarly, IMS TM applications can be driven externally using the IMS resource adapter, enabling them for API usage too.

However, for older applications the access problem is more difficult. These applications were designed to be run from a screen, and terminal handling is built into the application together with the business logic. Screen handling is through the manipulation of 3270 data streams. Again in CICS terms, these programs are often referred to as ‘BMS Applications’, that is, applications that use the CICS Basic Mapping Service facility to process screen-based menus. In order to bring these applications into the API fold, it is necessary for the API middleware to provide a mechanism to drive them through their built-in screen-based interfaces.

Given that the functions of interest for deployment as APIs are likely to consist of multiple different applications or parts of applications, a basic level of orchestration will also be required in the API middleware. This may be based on some sort of standard, such as BPEL, but it must be able to handle the creation of a process flow to execute the desired API function.

Finally, in order to be open to third party developers and a wide range of API consumer platforms, the API middleware needs to provide a standard invocation structure regardless of where or on what technology platform they are running. For example, this standard form of execution could be through a REST URL-based interface or a WSDL-based web service. But it is up to the API middleware to provide the necessary wrappers or adapters that can bridge from the desired standard interface specification to the required access mechanisms such as COMMAREA-based LINKs.

Best-of-Breed Characteristics

With this basic level of functionality, it is possible to API enable at least CICS core system applications. However, with just the basic level of API middleware functionality, the task is likely to prove cumbersome, error-prone, and time and resource-intensive, and certainly does not naturally fit with a modern DevOps approach to application development and deployment. In addition, many companies have important core system applications running in other environments, such as IMS, IDMS or even batch.

To address the limitations of basic-level API enablement of core systems, the API middleware will usually provide a range of other functions and capabilities. These will now be considered as potential best-of-breed characteristics. It is important to recognize that the following characteristics may not all be required by every company looking to API-enable their core systems environment. Instead, the characteristics are provided as a checklist of functionality that may or may not be required. This allows a company to choose the characteristics relevant to its own requirements when assessing API middleware.

Best-of-breed characteristics will be considered in three main sections:

- Development / Deployment
- Operations
- Flexibility

Development / Deployment

As discussed earlier the issue of **core systems service composition** is critical. Applications, services, data and processes must be able to be packaged into APIs in such a way that they make optimal use of systems of record without the imposition of unnecessary constraints and technical complexity for the API consumer. Achieving the **appropriate granularity** ensures that knowledge of internal process and implementation detail is decoupled from use of the API, but there are other factors that need to be taken into account too. In a legacy system environment it is important to be sensitive to local policy on resource usage. Each communication backwards and forwards between core systems and the API consumers will have a significant cost, so the provided APIs need to take that into account as they optimize the granularity. Also, some companies like to keep a very tight rein on the growth of their COBOL libraries, so it may be unsatisfactory for any tool to generate or require new COBOL programs.

Another key point for a best-of-breed tool is to support the two distinct forms of designing and composing APIs with the API middleware layer: **bottom-up and top-down.** These two design approaches reflect the different points of access to APIs, from legacy systems or distributed worlds. Typically, the bottom-up approach involves the core systems team looking at the assets to be exposed, considering the interfaces used such as CICS COMMAREA, and then mapping this up through the API middleware layer to the corresponding APIs. The top-down approach tends to be used when driven from outside of the core systems. The solution developer looking to leverage on prem services defines the required systems of record activities and then passes this across to the team so that the desired service can be composed with the API middleware to package up the specific application steps. Different organizations will feel most comfortable with different approaches, and therefore support is required in the API middleware for both these design methods.

Some organizations still reliant on core systems may have already moved down a web services-based approach to their access. Therefore, API middleware may well need to offer **web services support** to leverage this investment. Admittedly, web services do provide an overhead, because they are very standards-based and formally structured, and they are not necessarily a good fit in terms of skills requirements to fit within a modern, DevOps-based rapid development environment. But it is that formality that is often valued as a way of ensuring quality of service levels and mitigating risk. The best-of-breed API middleware, then, will provide a mechanism to help developers to bridge between their core systems and perhaps less well known web services technologies.

Another best-of-breed characteristic will be to **reduce coding / code generation**, or preferably eliminate it entirely. Although it is obvious that less coding will result in lower costs and faster time-to-value, it is also particularly beneficial in the case of legacy systems, because of the fact that any coding required might well involve such concepts such as REST and JSON, where programmer unfamiliarity may lead to a greater potential for error. DevOps support will particularly benefit from minimal coding requirements, enabling rapid API development and deployment. Related to this point, ideally API middleware should also **automate** as much of the service creation task as possible on core systems so that it can be done quickly, intuitively and without the need for extensive retraining. This will allow legacy developers to quickly and efficiently create and deploy new or modified services, making the best use of available development resources.

Language support will obviously be another key area of added value. Core systems applications might be written in COBOL, PL/1, Assembler or even in a higher-level language like Natural. The company API-enabling these systems should ensure that any toolset under consideration supports the necessary languages used for its applications. This support will almost certainly need to encompass tools to map certain programming structures such as COBOL copybooks into standards-based formats, using technologies like XML or JSON.

API middleware must be able to support various different types of applications such as CICS COMMAREA and screen-based ones, IMS-based ones and even batch routines. For many years there has been an active and well-developed aftermarket of vendors supplying a whole range of applications and platforms for legacy systems, and support for the appropriate ones will be an essential requirement for any user embarking on a project to API-enable them. The middleware may also need to access legacy data and applications, which would require access to DB2, VSAM, Adabas and other data sources, preferably under a single SQL-style interface.

Basic orchestration is a fundamental requirement for any API middleware. But this is an area where enormous advantage can be gained by supporting value-added orchestration capabilities. Best-of-breed API middleware needs to make the task of composing resources and processes as simple and error-free as

possible, covering application platforms like CICS and IMS, data sources such as DB2 and

VSAM and existing processes that may already be flowcharted with BPEL or some sort of Business Process Management (BPM) solution. The more automation the API middleware can provide, the less demand will be placed on the API developers and consumers, reducing risk.

All of the best-of-breed areas discussed so far lead to what is likely to be one of the most critical areas of differentiation for legacy system API middleware – **ease-of-use**. It is quite possible for the API middleware to satisfy the previous requirements but still leave a lot of work for the technical staff to perform. Tools need to be intuitive, with minimal training requirements, and allow both legacy system and distributed programmers to concentrate on composing services of the right level of granularity for use in both environments without the need for expensive third-party services. It is well worth verifying these facts with any prospective supplier before any acquisition is made.

There are a host of functions that belong to the generic API model discussion rather than the API middleware itself. Examples are partner management services to manage third party API developer partners, sandbox services for rapid development and specific device support. However, although the API middleware does not necessarily need to provide these functions directly, it must offer **API ecosystem support.** Security and management are key aspects of an API model, especially given the fact that APIs may be being used by third party developers. Most API solutions offer some sort of key system to authorize particular API development partners for what they can and cannot access, and it may be necessary for the API middleware to enforce that level of authorization across the core systems services being utilized. There will also be a need for material to provide API developers with the necessary information about the APIs supported on core systems, in terms of what they do and what inputs and outputs they expect.

An essential part of any development process is **testing**, and this is another important best-of-breed area for API middleware toolsets. This relates partly to the skills optimization issue, but also to the nature of APIs. Testing will be very much easier if different components or services can be tested in isolation, rather than having to wait for all the relevant components to be completed and assembled before any testing can occur. A test harness that enables a legacy systems developer to test a particular composed operation or its individual parts, creating inputs and outputs to simulate real operations, will be invaluable in terms of reducing time-to-market for new projects and speeding up the overall development process.

Once the new services are developed and composed into APIs for external presentation, the core systems API middleware toolset will need to provide **governance and lifecycle support**. This should allow the creation of new APIs to be controlled and managed appropriately, fitting into corporate governance procedures and then passing through development/test/QA/ production-levels to ensure that development, deployment and production operations can be managed safely. Versioning should also be supported, to allow for legacy system services to be updated in flight. Otherwise, there is an increased risk that an unprepared or incorrectly leveled change might enter production, with potentially damaging consequences. A critical issue for core systems users, where skills are at a premium, is to ensure that APIs can be modified, enhanced and reused as quickly and easily as possible with minimum additional effort.

Finally, an important aspect of a best-of-breed toolset will be the **core systems expertise** of the tool supplier. Although this is not actually a functional requirement, this point reflects the discussion about system values and the need to be sensitive to the special requirements of working in a legacy environment. In order for the toolset to be usable, effective and efficient, it will be vital that it is created based on an extensive understanding of legacy systems. For example, these environments typically have stringent requirements on integrity and recoverability. There are also many operating system functions that will be useful, and in the end the ‘look and feel’ of the toolset will be important in order to gain acceptance within the legacy systems community.

Best of breed c haracteristic : Development / Deployment	*Additional c omments*
Core systems service composition	■ *Developing orchestration flows* ■ *Building the right level of API granularity* ■ *Optimizing load and resource usage*
Bottom-up and top-down service development	■ *Flexibility to choose the API development approach*
Support for web services / SOA	■ *For companies already invested in these technologies*
Minimal code generation	■ *Providing codeless ways to drive core systems components*
Automation facilities	■ *High degree of automation*
Language support	■ *COBOL, PL/I, Natural, …*
Support for additional resources	■ *CICS, IMS, 3270, Batch…* ■ *Other native packages and systems* ■ *Adabas, DB2, …*
Added-value orchestration	■ *Spanning all resource types* ■ *Leveraging existing flowcharts and processes*
Ease-of-use	■ *Intuitive and efficient* ■ *Minimal training requirements*
API Ecosystem support	■ *Security and management* ■ *Usage information* ■ *API invocation details (inputs, outputs etc)*
Testing tools	■ *Modelling / testing for individual components*
Governance and lifecycle support	■ *Creation, deployment, retirement*
Core systems experience	■ *Vendor - provided experience and skills*

Figure 6: Best of breed characteristics – Development / deployment

Operations

Once the core systems APIs have been created and tested, focus moves onto production operations. At this stage, the wider API ecosystem becomes heavily involved with facilities, usually through some sort of API Management tool. But it is important that best-of-breed legacy system API middleware still maintains a close working relationship with the ecosystem. For example, the middleware needs to ensure that the ecosystem has all the necessary information from the legacy perspective to function effectively, and there will be specific areas where the ecosystem will almost certainly be somewhat blind. In particular, API models where third party developers are involved often need **to support API analytics** of some sort, accurately measuring and reporting API activity and usage. This is a key component in quality control, helping to identify successful third part partners and allowing greater usage as confidence grows.

It is important that any core systems API middleware considers the question of **administration**, since all assets need to be managed seamlessly to gain the most advantage from the API model. New APIs need to be implemented, new users need to be authorized for allowed usage, and APIs being replaced may need to be closed out, to mention just a few administrative tasks that need to be addressed. In addition, API execution may span different locations as well as environments, even spreading as far as into partner and other third party-companies. Therefore, the administration capabilities must include remote operations to encompass end-to-end transaction needs. While some of these issues fall within the responsibilities of the ecosystem, the API middleware needs to support such activities.

Of course, **security** is another issue that will be absolutely key to companies reliant on core systems. Users are accustomed to a high level of security, and their operations are often mission-critical in nature. These factors combine to create a high level of concern and potential risk when deploying a core systems-based API, particularly because prized assets are now made available to the great wide world of tablets, phones and the Internet of Things (IoT). Therefore it is crucial that security is managed carefully, linking up with existing legacy systems security facilities in use such as RACF. Security in this sense may need to address all four main areas of authentication, privacy (encryption), integrity and non-repudiation. Once again, this will be primarily the responsibility of the full API ecosystem but API middleware toolset must play its part. For example, the tools themselves must be secured from unauthorized usage.

The complexities of operating an API model, exacerbated by the use of asynchronous and event-driven modes of operation, create a challenge in understanding what is actually happening in the live, production environment. APIs may be executing and driving activities asynchronously, crossing platform and location boundaries at will, and it can become extremely confusing to the operations staff. The result is that it can be hard to maintain service levels and general responsiveness. The answer is for the API technology to offer some level of **monitoring** and **problem determination** capability, and the API middleware has to play a key part in this since it knows which activities are being driven by which APIs. These facilities need to be able to glean information from within the core systems to be combined with additional information in order to understand what is happening, and then provide the necessary investigation and action tools, such as service tracing and data flow analysis, to identify and resolve any problems spotted by the monitoring component. Ideally, this functionality should also provide some method **for integrating with the enterprise management framework**, for example offering an SNMP agent to alert the framework of problems within the wider legacy system environment.

Best of breed characteristic: Operations	*Additional comments*
API analytics support	■ *Gather usage statistics for individual APIs* ■ *Generate reports* ■ *Pass analytics information to API ecosystem analytics tools*
Administration support for the API ecosystem	■ *Handling all environments* ■ *Bring legacy systems APIs online and offline* ■ *User management liaison* ■ *Remote operations capability*
Security	■ *Interoperation with API ecosystem* ■ *Authentication, integrity, privacy and non-repudiation functions* ■ *Protection of API middleware assets*
Monitoring and problem determination	■ *Information sharing with ecosystem tools* ■ *Tracking of APIs and their internal flows*
Integration with existing management framework	■ *Systems management* ■ *Alerts* ■ *Resource management*

Figure 7: Best of breed characteristics – Operations

Flexibility

The final set of best-of-breed characteristics relates loosely to the flexibility of the toolset to support the core systems APIs. For some companies, **bi-directional** support will be critical, allowing API components controlled by the API middleware to drive other APIs that may reside on other systems. Efforts to gain more value from legacy system investments by enabling them to support the API model frequently focus on exposing resources for outside usage. This will certainly be the mindset of a company that is looking to stabilize their core investment. However, many companies have accepted that their system of record remains a key element of strategic planning for the future. These companies are likely to be very interested in the additional benefits to be obtained by being able to interlink APIs – that is, for core applications to be able to run external APIs as well as the other way around. This ensures that *all* IT investments are leveraged, not just core investments. This bi-directional capability may require some additional work in the toolset since a way must be provided for a core systems application to issue API requests.

Another aspect of flexibility relates to where the API middleware runtime processing is carried out. This decision will usually be made based on specific service-level requirements or restrictions. Among other things, the API middleware needs to deal with navigation issues for core applications, such as which CICS program should be called depending on the result of the previous one, and formatting issues like dealing with 3270 data streams when accessing screen-based applications. Depending on issues such as performance requirements, capacity and ease of programming, it might be desirable to run all of the API middleware functionality in its own legacy address space or within an existing legacy system environment such as CICS or IMS. Alternatively, it might be desirable for some or all of these activities to take place within a specialty engine, to reduce costs and improve performance and scalability, or even off prem altogether, perhaps in a distributed server, appliance or the Cloud. Having said that though, it is highly likely that orchestration should be restricted to core systems to avoid having to expose specific processes and procedures to additional professionals. Whatever combination works best for the user, it is clear that a best-of-breed core system API middleware toolset will need to support this choice of processing location.

As just touched on, support for different processing locations and the previously discussed monitoring support are important factors in another area of consideration for best-of-breed API middleware toolsets - that is, **scalability and performance**. As well as these areas, the toolset will probably need to provide some level of statistics reporting capability to allow effective capacity planning and load management and to provide proper governance of third-party API developers. In addition, when running in a mode where most of the API processing is carried out on core systems, the API middleware toolset should **exploit native legacy high-performance options** such as specialty engines and cross-memory support to optimize performance.

Best of breed characteristic: Flexibility	***Additional comments***
Bi-directional support	■ *Access to external APIs* ■ *API access to core systems* ■ *API access across core systems*
Choice of processing location	■ *Inside CICS* ■ *Outside CICS but on cor systems* ■ *Making use of specialty engines* ■ *Off prem, on distributed servers, appliances or in the Cloud*
Performance / Scalability	■ *Ability to choose processing location based on performance/scalability needs* ■ *Statistics on services usage for capacity planning purposes*
Exploit native high-performance options	■ *Leverage legacy system - specific optimization technologies*

Figure 8: Best of breed characteristics – Flexibility

Summary

For many companies, core systems remain a key asset for the foreseeable future. As these companies strive to deliver increased business value from their IT investments, opening up core systems becomes a major focus. The API model and its related toolsets offer a way not only to extend the life of legacy systems, but also to ensure that it continues to play a valued role in driving the business forwards. Breaking down the barriers between the core systems and the rest of the IT world ensures that investments can be made wherever they make the most sense, where the entire IT installation and its users may benefit.

But success or failure with legacy technology in the API world will be governed to a large extent by the effectiveness of the tools to support API enablement combined with a disciplined focus on delivering the right legacy-based APIs for the right solutions. Generic API middleware tools developed without considering special core system needs will not do the job effectively. Instead, companies need to look for toolsets that are specifically oriented to legacy systems, taking account of architectural, development, deployment and operational needs.

All core systems companies will have their own specific requirements and hot buttons, and these will affect the selection of the most appropriate tools. The best-of-breed characteristics presented in this paper are intended to provide a checklist that companies can evaluate against their own requirements, and then against the toolsets offered by the various suppliers. Some of these characteristics address purely functional needs, while others have implications for wider critical aspects of core systems API initiatives, such as skills requirements, security and manageability. However, the goal remains to ensure that companies end up with the API middleware best designed to ensure the success of their own legacy system API initiatives.

About Lustratus Research

Lustratus Research, founded in 2006, aims to deliver independent and unbiased analysis of global software technology trends for senior IT and business unit management, shedding light on the latest developments and best practices and interpreting them into business value and impact. Lustratus analysts include some of the top thought leaders worldwide in infrastructure software.

Lustratus offers a unique structure of materials, consisting of three categories—Insights, Reports and Research. Insights offer concise analysis and opinion, while Reports offer more comprehensive breadth and depth. Research documents provide the results of practical investigations and experiences. Lustratus prides itself on bringing the technical and business aspects of technology and best practices together, in order to clearly address the business impacts. Each Lustratus document is graded based on its technical or business orientation, as a guide to readers.

Terms and Conditions

While the information is based on best available resources, Lustratus Research disclaims all warranties as to the accuracy, completeness or adequacy of such information. Lustratus Research shall have no liability for errors, omissions or in adequacies in the information contained herein or for interpretations thereof. Opinions reflect judgment at the time and are subject to change. All trademarks appearing in this report are trademarks of their respective owners.

lustratus

Steve Craggs trading as Lustratus Research
www.lustratus.com

Ref SC/LR/31675249V2.0