Realtime publishers

The Essentials Series

Mainframe Application Modernization

sponsored by

by Don Jones

Article 1: Is Your Mainframe Application Working as Hard as You Are? 1
Setting the Scene 1
The Opportunities for Mainframe Applications 2
The Problem with Mainframe Applications 3
The Hardware Problem 4
The Software Problem 4
The Data Problem 5
What's the Solution? 5
Article 2: Options for Modernizing Mainframe Applications 6
Solution: Rehost 6
Solution: Recode 6
Solution: Repackage 7
Strategic or Tactical? 9
Repackaging Your Mainframe Application 10
Article 3: Considerations for Injecting Life into Mainframe Applications 11
Repackaging Techniques 11
Host Integration 11
Bridge Integration 13
Transaction Integration 14
Keep Your Mainframe—and Gain Agility 15

Copyright Statement

© 2009 Realtime Publishers. All rights reserved. This site contains materials that have been created, developed, or commissioned by, and published with the permission of, Realtime Publishers (the "Materials") and this site and any such Materials are protected by international copyright and trademark laws.

THE MATERIALS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. The Materials are subject to change without notice and do not represent a commitment on the part of Realtime Publishers or its web site sponsors. In no event shall Realtime Publishers or its web site sponsors be held liable for technical or editorial errors or omissions contained in the Materials, including without limitation, for any direct, indirect, incidental, special, exemplary or consequential damages whatsoever resulting from the use of any information contained in the Materials.

The Materials (including but not limited to the text, images, audio, and/or video) may not be copied, reproduced, republished, uploaded, posted, transmitted, or distributed in any way, in whole or in part, except that one copy may be downloaded for your personal, non-commercial use on a single computer. In connection with such use, you may not modify or obscure any copyright or other proprietary notice.

The Materials may contain trademarks, services marks and logos that are the property of third parties. You are not permitted to use these trademarks, services marks or logos without prior written consent of such third parties.

Realtime Publishers and the Realtime Publishers logo are registered in the US Patent & Trademark Office. All other product or service names are the property of their respective owners.

If you have any questions about these terms, or if you would like information about licensing materials from Realtime Publishers, please contact us via e-mail at info@realtimepublishers.com.

Article 1: Is Your Mainframe Application Working as Hard as You Are?

Many of today's most successful companies run critical portions of their IT operations on a mainframe or midrange computer—and have done so for years and years. Mainframe applications offer power, stability, and reach—all things that enterprises rely on. In recent years, mainframe vendors have kept up with the times, offering the ability to host Web-based applications, email and collaboration software, and much more.

But mainframe applications present a problem, too: They lock up all your most important data in what is, in many ways, a black box. Mainframe applications tend to be highly proprietary and often include much of their business logic within the presentation layer, making it difficult, impractical, or outright impossible to safely access the application data by any other means.

But today's businesses *need* to utilize their mainframe applications in *many* other ways. Today's businesses need to move faster, be more flexible, and offer more services to internal and external customers. How can they do so when their application is locked up in a box?

This *Essential Series* guide will explore exactly that problem: How to bring your mainframe applications into the modern world, how to leverage mainframe-based data in powerful and exciting new ways, and—most importantly—how to do so without losing or compromising your existing mainframe investment.

Setting the Scene

Before we dive in, let's take a moment to make sure you and I are on the same page with what constitutes a mainframe application. I think this is important only because many folks are using their mainframes in creative new ways, and some of those newer and more flexible ways don't have the same problems as a more traditional mainframe application.

Traditional mainframe applications—the kind I'll be discussing in this guide—are commonly used for bulk data processing, such as census data. They're also used to collate statistics from large bodies of data, for complex tasks such as Enterprise Resource Planning (ERP), for mass-data manipulation such as Customer Relationship Management (CRM) applications, and for financial transaction processing.

Mainframe computers—*big iron* to some—often descend from the venerable IBM System/360 family, including modern mainframe machines such as IBM's zSeries, System z9, and System z10 servers. Other mainframe vendors include Unisys, Hewlett-Packard, Fujitsu, NEC, and so on. Your environment may feature a somewhat older—but still reliable—IBM AS/400 or other mainframe computer.

IBM estimates that 90% of their mainframes have Customer Information Control System (CICS) transaction processing software installed, and that their IMS and DB2 database software is also popular. Many IBM mainframes also run WebSphere MQ and WebSphere Application Server middleware—all enterprise-grade, mission-critical software that contains a great deal of data you'd probably like to use elsewhere.

Most mainframe applications store their data inside fairly standardized databases—DB2 being a star example on IBM mainframes. Accessing this data directly is straightforward, but ultimately not the best way to extend mainframe applications because the data itself doesn't contain any of the business logic. When we speak of extending a mainframe application, we usually want its business logic and other capabilities, not just the raw data. In fact, what would be ideal is to somehow transform the entire application from being a user interface (UI)-centric application into some kind of modern middle-tier component that encapsulates the application's business logic.

The Opportunities for Mainframe Applications

Let's take a simple example from my own experience: I used to work for a traditional, brick-and-mortar retailer of computer and video game software. Our business was run entirely on an IBM AS/400—in fact, our office didn't even include a Local Area Network (LAN) for several years because we all worked almost entirely on the AS/400 via 5250 terminal emulation. Even our email lived in the AS/400 (this was at the dawn of the public Internet, so email was strictly an internal thing at the time). Our AS/400 had a variety of prepackaged and custom-written (in the RPG programming language) applications that ran the business, from gathering sales transactions from our retail stores and processing restocking orders to communicating purchase orders to distributors and wholesalers to managing the product distribution center and inventory. The AS/400 even managed a fairly successful phone-in direct mail business, where customers would place orders by phone from paper catalogs, and we would fulfill those orders right from the main distribution center. A *lot* of business logic lived in the AS/400—how we handled inventory, how orders were processed, payment processing, and so on.

As the Internet gained in popularity and more of our customers got online, we wanted to offer an online shopping alternative. We were faced with a problem, though: All our data was wrapped up in the mainframe. It was *good* data, too—detailed product information, package sizes and weights (useful when calculating shipping), system compatibility information, and so on. All of our business processes were encapsulated in those applications, too. It wasn't enough to simply dump orders into a database; we wanted to use all the order-processing software we already owned—on the AS/400.

The AS/400 even had a robust customer order tracking system, which our mail order business unit relied upon and really liked. We *wanted* to simply re-use all that capability in a new, Web-based e-commerce application. Ideally, the online catalog would be built from the AS/400's database. Product availability would be taken from the AS/400's inventory information. That couldn't just come from database tables, though; *availability* in our world was a complex calculation involving on-hand stock, expected incoming shipments, existing commitments to our retail stores, and so forth. Ideally, orders would also be processed in the AS/400 and fulfilled just like our existing mail orders. We figured that we already had all the right pieces in place—we just needed to hook the Web site up to the AS/400. We didn't want to run the Web site *on* the AS/400—for a number of business reasons, we'd decided to use a farm of less-expensive Microsoft Windows-based Web servers—but we did figure that the entire backend could simply stay right where it was, on the '400.

We were wrong. We wound up implementing a complex system of data dumps and imports. We would dump data from the AS/400 each night, and import that data into a Microsoft SQL Server database that provided the backend for the Web site. Sales transactions were processed not by the AS/400 but by the individual Web servers using an all-new credit card processing infrastructure we had to build. Transactions were then exported from SQL Server and pulled into the AS/400 in a twice-daily import process so that order fulfillment could still take place within the mainframe—that import process had to leverage all the business logic in the AS/400 needed for order processing. All this importing and exporting still required hundreds of hours of custom RPG programming because we needed to make sure the incoming data was "clean" before dumping it into the AS/400's—because the other applications using that data assumed that anything in the database had been thoroughly validated. The entire project was painful, and it concerned us because we knew there would be other, similar data-sharing projects coming in the future. Looking back, I think that Web project may have been the beginning of the end for the AS/400 in our company. Great as it was at doing its job, we just couldn't keep investing in something that wouldn't expand and extend quickly when we needed it to.

So what was the problem? There were really three.

The Problem with Mainframe Applications

The big problem with mainframe applications is that they are generally monolithic. That is, the folks who wrote the application assumed that the application would be entirely self-contained, never need any major new capabilities, and never need any capabilities that the designers didn't think of in the first place. The application's presentation—generally screens designed for display on a terminal—business logic, data, and in many cases hardware are created in a single stack, often with very little differentiation between those layers. That makes it very difficult to break out *pieces* of the application, and while the application's data might be readily accessible, it's tough to find ways to leverage the application's business logic without extensive, expensive reprogramming.

The Hardware Problem

Some companies—especially those who are starting to eye their mainframe with a bit of distaste—focus on the mainframe hardware. It's expensive, although pricing has come down a bit as vendors begin to rely on commodity components such as Intel processors. Much of the hardware remains proprietary, and for companies who have moved beyond the need for the mainframe hardware, keeping it around—just because it happens to own all the important company data—can be painful.

For example, in the company I described earlier, we began moving more and more critical functions off the mainframe, utilizing an ever-more-complex arrangement of data transfers between the AS/400 and its PC-based successor systems. As more and more business processes began to depend on these delicate data dumps and imports, we began to see the AS/400 as a kind of weight around our necks. The hardware was expensive, the maintenance contracts were expensive, and we resented the mainframe simply because we felt it had taken our data hostage. We had a visceral feeling that simply ditching the hardware once and for all would solve a lot of our problems.

The Software Problem

The *real* problem, of course, lay mainly within the application software that ran on the mainframe. Mainframe applications are often written in what are powerful, but relatively primitive, programming languages like COBOL and RPG. These languages are well-suited to applications that need to process batches of data quickly, produce reports quickly, and interact with humans primarily through a text-based, field-oriented UI. These languages do *not,* however, lend themselves well to modern application development practices such as componentization, multi-tier development, and so forth.

Most modern languages—Microsoft's C#, Sun's Java, and even Web-oriented languages like PHP—are object-oriented and support object-oriented programming techniques such as encapsulation and inheritance. These, in turn, help foster a multi-tier development environment. For example, a modern application's UI is often little more than a set of basic data-entry routines. That data is submitted to a middle-tier component, whose job it is to validate the data before passing it to further tiers for processing. The business logic, in other words, resides in a discrete layer, enabling it to be used by not only the data-entry UI but by other means of input such as Web sites, automated business processes, and so forth.

Monolithic mainframe applications, however, tend to intermix the UI, business logic, and data management layers of an application. The *only* way to ensure that the right data gets into the application is to enter it into the UI. That's great for manual data entry but not so great when you begin looking for methods to leverage your data in other ways and to connect your data with other systems.

The Data Problem

Mainframe applications may use a wide range of database engines under the hood. Many are built on IBM's DB2 database; others may use internal, proprietary database engines. But even when the data is in a readily-accessible engine such as DB2, the data is often less than useful when it comes to connecting external applications.

One reason is the one I've already stated: You often can't just stick data into the database; you have to go through the application's entry screens to ensure the data is validated, that referential integrity is maintained, and so forth. My AS/400 experience included DB2, and we were delighted that Windows computers could communicate directly with the database using Open Database Connectivity (ODBC) drivers. We were dismayed when we realized that doing so would mean re-creating every scrap of business logic encompassed by our mainframe application so that the data we were sticking into DB2 would be valid and wouldn't cause problems for the many mainframe applications that utilized the data. In other words, just getting to the data didn't solve our problem: We basically needed to re-engineer all our application's business logic in a brand-new set of middle-tier components written in COM, .NET, or Java. That's a heavy task, even though we only needed to do it for *portions* of the mainframe application. We'd also have an ongoing maintenance problem, as any changes to business logic would then have to be reprogrammed in parallel—once on the mainframe, and once in the more modern middle-tier components.

What's the Solution?

Like anything in the IT world, there's never simply *one* solution. In the next part of this guide, I'll explore common solutions to the problem of using mainframe data in more modern scenarios. We'll look at the pros and cons of each solution, and try to come up with a wish list of capabilities and technologies that will lead to the best long-term solution.

Article 2: Options for Modernizing Mainframe Applications

So how can we begin exposing our mainframe application's data and services outside the boundaries of the mainframe computer? I want to try to address the problem from three different angles: dealing with the hardware, dealing with the applications themselves, and dealing with the data directly.

Solution: Rehost

One option is to *rehost* your applications, something that's commonly done with applications written in a language such as COBOL. Rehosting lets you get rid of your mainframe hardware by moving the COBOL application to a non-mainframe host, which might be a UNIX or Windows computer, and might even be virtualized. Doing so gets the application running elsewhere, but it ultimately doesn't do much to change the monolithic nature of the application. Sure, you can cut back on mainframe hardware support costs, but that's often the only savings. The data in that application, and the services it provides, are still pretty trapped inside the application—you haven't extended it to any other portions of your enterprise. This solution is a good option if the *only* thing you need to do is eliminate the mainframe hardware, but it's not really an option for truly modernizing the application itself.

Solution: Recode

Software is malleable. With the right skills, tools, and time, software can be changed to do almost anything. So if your current mainframe application doesn't do what you need—recode it!

This is the exact direction my company took with our AS/400 for several years. We had a team of about a half-dozen dedicated RPG programmers. We acquired the source code for most of our enterprise applications (at great expense, by the way), and we started customizing them mercilessly. Once we'd acquired that on-staff skill set, "reprogram the AS/400" became the answer to nearly every IT-related challenge that came our way. When our distribution center needed to integrate with new distribution hardware, we reprogrammed the AS/400 to talk to that hardware's UNIX-based controllers. When the distribution center wanted to use wireless terminals to facilitate stock movements, we reprogrammed the AS/400 to provide the needed functionality. Our AS/400 programmers had lengthy wish lists from nearly every department of the company, and it seemed like new and changed AS/400 entry screens were released every day.

What we gave up, however, was the ability to easily use more off-the-shelf applications. We could never upgrade our enterprise mainframe applications when the software vendors released new versions because we'd lose our massive customizations. We were unable—well, not unable but *unwilling*—to purchase other off-the-shelf applications because we had all these on-staff programmers; shouldn't we be writing this stuff instead of buying it? We were *very* unwilling to purchase applications that ran on anything but the AS/400, simply because we wanted to get the most from our heavy investment in the mainframe. So we mentally limited our options to what our programmers could do, and we physically limited ourselves to whatever they could actually accomplish in their 10 hours a day.

Solution: Repackage

These days, one of the most intelligent—I think—solutions to the whole problem is *repackaging.* Simply put, you write some kind of wrapper around your existing mainframe application to expose portions (or even all) of the application through more modern, standard interfaces. Maybe you want to expose the mainframe application as a set of Web services, a bunch of .NET Framework components, or a set of Java objects.

This approach works well because it directly addresses the fundamental problem of mainframe applications. As I said in the first article of this guide:

> Monolithic mainframe applications, however, tend to intermix the user interface (UI), the business logic, and the data management layers of an application. The only way to ensure that the right data gets into the application is to enter it into the user interface. That's great for manual data-entry, but not so great when you begin looking for ways to leverage your data in other ways and to connect your data with other systems.

Repackaging, done properly, can actually turn *the entire mainframe application into a middle-tier component.* It essentially automates the use of the existing mainframe application, exposing data and accepting input through a Web service, Java object, or whatever. External applications see a .NET Framework component; that component is actually a sophisticated engine that manipulates the native mainframe application, taking advantage of its inbuilt business logic and everything. This technique typically requires *no changes on the mainframe,* meaning you can use prepackaged, off-the-shelf applications *with no changes,* and finally get your data *off* of the mainframe—all while keeping your mainframe investment completely intact.

Better yet, in many cases, you can turn the entire application into not a single exposed service but a *set* of exposed services, essentially componentizing the application without touching the application itself. This means a large, monolithic inventory management application might be able to provide an inventory-query service, a reordering service, and other services—each of which is an element of the original mainframe application.

Consider Figure 1, which represents a typical mainframe application: input screens for data entry and retrieval, including embedded business logic, and an on-mainframe data store of some kind. This figure represents a standard, monolithic mainframe application like the ones you are probably already using.

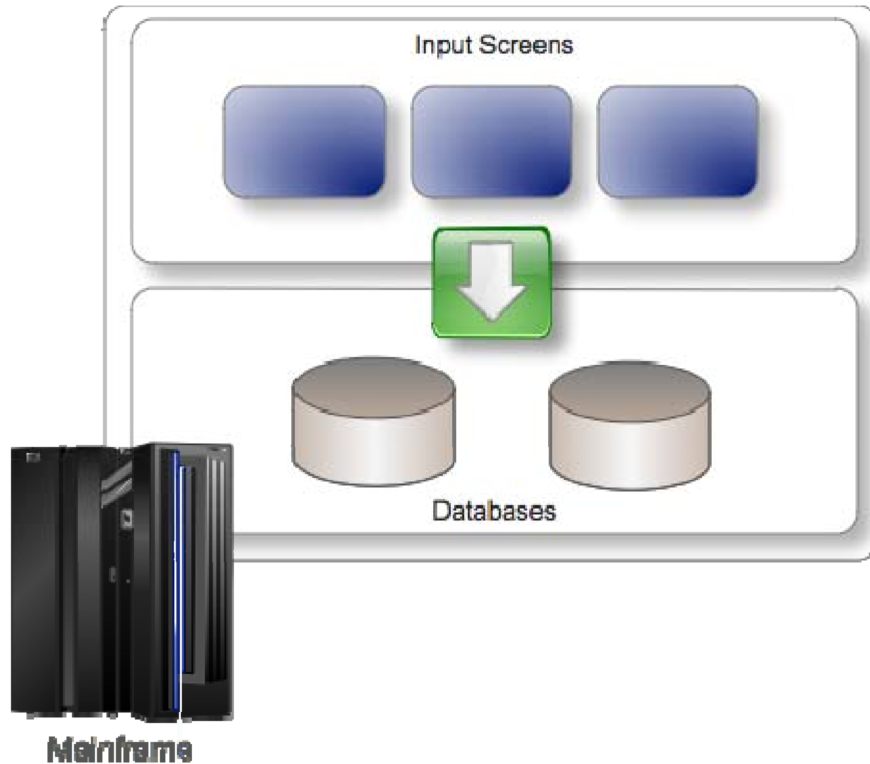

Figure 1: Monolithic mainframe application.

Because the application is monolithic, it can't be easily broken down into smaller components. That means, from the mainframe point of view, the entire application *is* the smallest component. However, you may well have external uses that just need the data from one or two screens, or just need to input data into a couple of screens. In other words, you've identified some element of the mainframe application that *could* be externally exposed as a distinct service. And you can do it: You simply have to repackage the application in such a way that the application can be used as-is—albeit in an automated, abstracted fashion—by other applications. Figure 2 illustrates this concept.

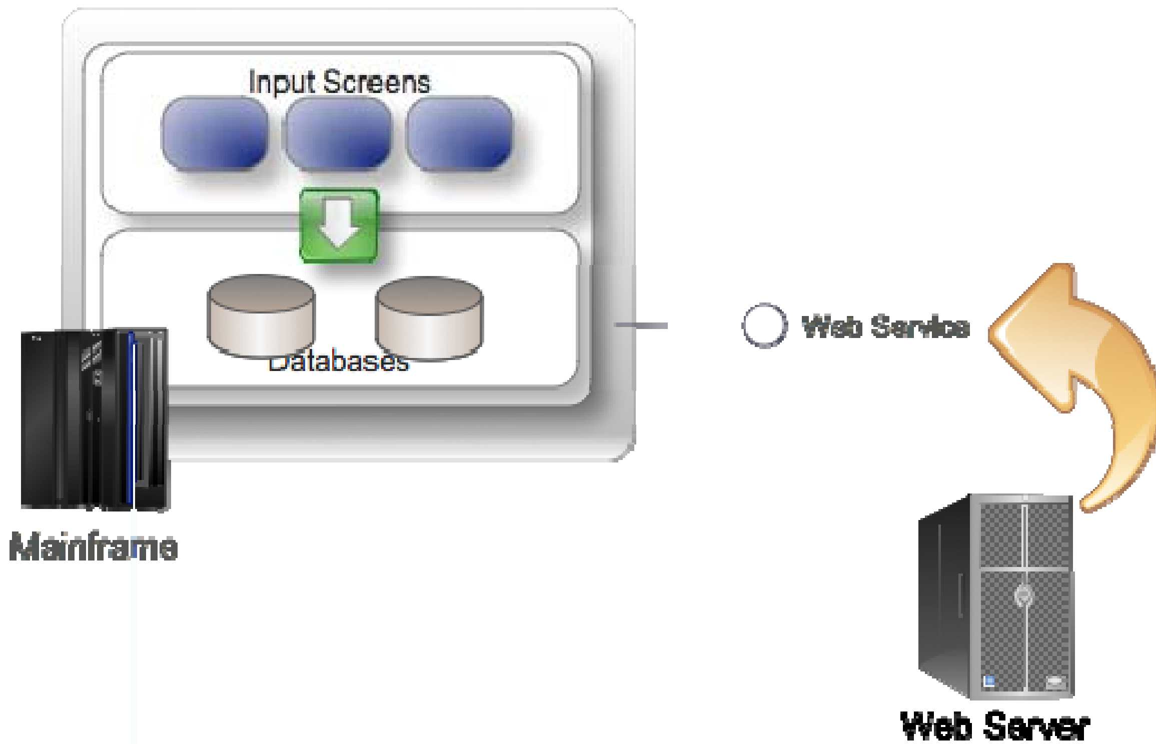

Figure 2: Repackaging a mainframe application.

This task requires some kind of repackaging engine. Essentially, that engine is taught how to automatically operate the mainframe application by reading its input screens and inserting data into the fields on those screens. The engine then exposes that data as a Web service (or Java object, or whatever) for non-mainframe applications to access—or as a *set* of Web services. You might very well *not* need to expose the entire mainframe application; you might only need to expose a portion of it, and treat that portion of the application as a component.

What's the benefit? The main benefit is that you still get to take advantage of the business logic that lives within the mainframe application—without modifying the application in any way.

Strategic or Tactical?

The general solution you choose is going to depend largely on your future plans. For example, choosing to rehost your application can help lower hardware costs, but it honestly won't do anything to change your ability to expose pieces of the application externally—it'll all still be locked up in a monolithic application. Rehosting is really a tactic designed to lower costs and driven by a larger strategic decision to *keep the application as-is.* You're not really modernizing anything; you're simply moving pieces around.

Recoding your mainframe applications is also a tactic driven by a larger strategic decision to—again—*keep the mainframe.* After all, why invest hours in recoding applications if you plan to move off the mainframe eventually? Recoding is also making a decision to continue operating in a mainframe-centric universe, where you want as much functionality as possible to live within the mainframe, and where you're willing to invest the necessary time and money to making that a possibility. Because recoding mainframe applications can be expensive, and because the skills needed to do so are frankly not in great supply, you may also be making a tacit decision to forgo business requirements that may *need* recoding, if the recoding skills, time, or money aren't available. In other words, you're happy letting the mainframe's technology drive, to a point, what your business can and cannot do.

Repackaging, however, doesn't lock you into anything. You can use it as a short-term tactical move: "We're going to repackage key portions of our mainframe application and use them elsewhere because we plan to migrate off the mainframe at some point." Repackaging also supports the longer-term strategic direction to stay *on* the mainframe: "We like our mainframe, we want to keep it, but we also want increased flexibility—which repackaging provides." Done properly (which I'll discuss in the next article), repackaging doesn't change anything *on* the mainframe, so it preserves your mainframe assets while minimizing or eliminating the need for costly custom programming on the mainframe. Repackaging, in other words, offers the most flexible kind of solution, and a solution that can fit many different strategic or tactical directions.

Repackaging Your Mainframe Application

In the final article of this guide, I'll look at the repackaging solution in more detail and discuss one approach that can either offer a long-term strategic direction or be utilized as more of a short-term, immediate tactical solution that helps support a particular long-term strategy.

Article 3: Considerations for Injecting Life into Mainframe Applications

To quickly review the key points from the prior two articles in this guide:

- We need to be able to safely interact with portions of the mainframe application in non-mainframe applications
- We don't want to access the data directly, because then we're losing the application's business logic
- Repackaging allows us to build a wrapper around the entire mainframe application, or portions of it, making it accessible via Web services, .NET Framework, or Java

There are different techniques for accomplishing this repackaging, depending on exactly what kind of mainframe application you're dealing with.

Repackaging Techniques

These repackaging techniques all rely on the presence of some kind of purpose-built mainframe integration engine. The engine is what does the work of talking to the mainframe in the proper fashion, and exposing the resulting data to the outside world through whatever means you choose. These engines commonly use a variety of techniques, and I'll discuss the three primary ones here.

Host Integration

The easiest to understand technique is probably *host integration.* Using this technique, the integration engine literally operates the mainframe application in exactly the same way a human being would. The basic technique is as old as mainframes themselves, and is commonly called *screen scraping.* Essentially, you explain to the hosting engine how the mainframe application operates: where data is located on each screen, what type of data is expected for each input field on each screen, what keypresses move between screens, and so forth. This information is called a *model.* You then indicate which pieces of information and which capabilities you plan to expose to the outside world. The engine exposes that information as properties and methods (of a Web service, Java object, or .NET Framework component), which can be used by any compatible external application. Figure 1 illustrates the concept.

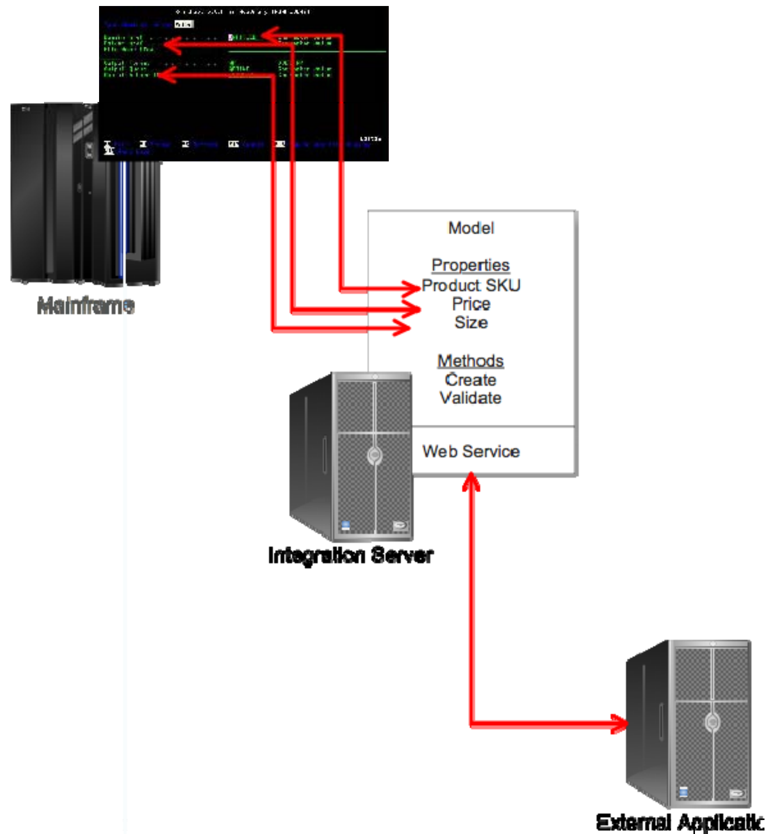

Figure 1: Modeling the mainframe application.

When an external application accesses data, the integration engine uses its model to determine where the mainframe application exposes that data. The engine manipulates the mainframe application to get to that point, reads the needed data from the screen, and then passes it to the external application. Integration engines commonly support a variety of terminal emulation protocols that enable them to talk to almost any mainframe, including IBM 3270, IBM 5250, VT/UNIX, HP700/92, and so on.

A great side benefit of this technique is that it's easy to manage change. If the underlying mainframe application changes—say, you upgrade a version—you don't need to modify your external applications. You simply update the integration engine's model so that it knows where to find all the data again. The engine thus serves as a kind of abstraction layer, helping to hide any complexities of the mainframe application and exposing the data in a straightforward, standardized, *modern* fashion.

The main benefit, though, is that you're getting the full capability of the entire application: its data, it business logic, and so forth. You're simply enabling things other than human beings to interact with selected portions of the application. In essence, you've taken a user interface (UI)-centric application and turned it into a sort of middle-tier component, exposed through modern architectures such as Java, COM, or .NET.

Bridge Integration

Some highly-standardized mainframe applications may be accessible through somewhat more sophisticated means. IBM's CICS software, for example, exposes a well-documented set of application maps called BMS maps. Rather than manipulating CICS' UI screens directly, an integration engine can tap into the underlying application through these maps via an IBM-provided bridge, as illustrated in Figure 2.

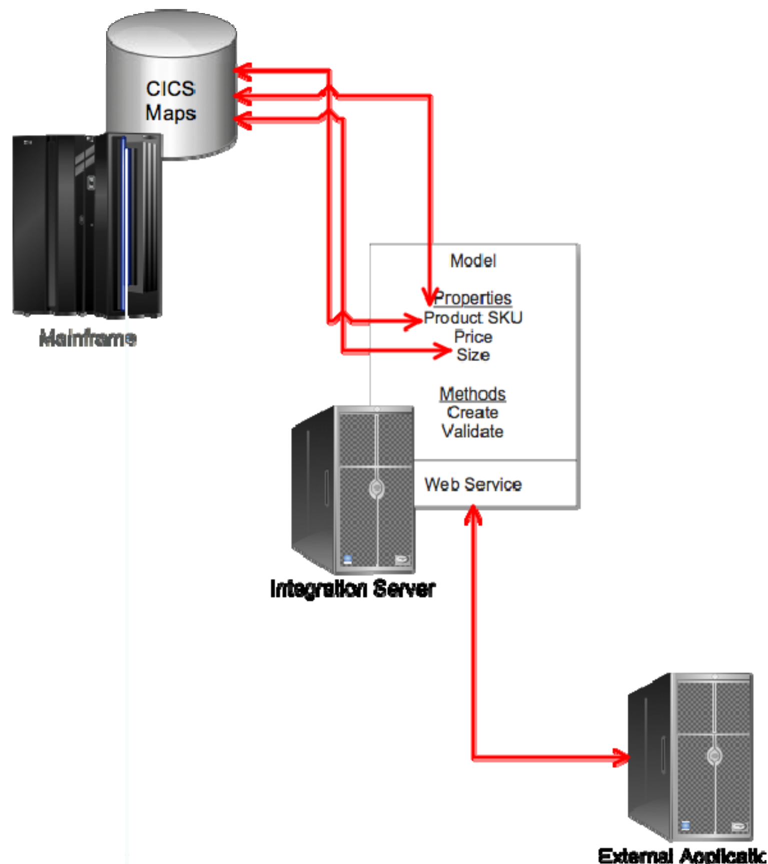

Figure 2: Tapping into CICS by using bridge integration.

Note

I've illustrated a "model" here, although it's important to note this isn't something you have to build directly. It's something the integration engine maintains on its own.

This technique is somewhat faster because the integration engine can talk directly to CICS application logic before the application renders a screen. The bridge, in essence, has direct access to COBOL fields and variables *behind* the column-level data on the screen. There's no need to manually build a model, and there's no dependency on the mainframe UI, so if that changes, you don't have any extra work to do. Best yet, the bridge is actually manipulating the COBOL code of CICS directly, which means all the application's business logic remains intact and usable, even though you're not going through the UI.

The end result, however, is identical: Portions of the mainframe application (or all of it, if needed) are exposed to external applications *with no changes to the mainframe.* You don't have a single line of extra code running on the mainframe to make this happen, so it's a low-impact, *very* fast way of integrating mainframe data into many external applications. As with host integration, you're taking a UI-centric application and transforming it into a set of middle-tier components.

Transaction Integration

Finally, an even more sophisticated technique can leverage the COM area of CICS and IMS applications because these applications are actually designed to support a certain degree of integration by external applications. As Figure 3 shows, the integration engine is simply mapping the applications' native integration protocols with more modern, standardized interfaces such as a Web service. This technique interacts with the mainframe application *below* its workflow and application logic levels, meaning you're far further "under the hood" and able to do things that the application might not normally allow, which is both good and bad.

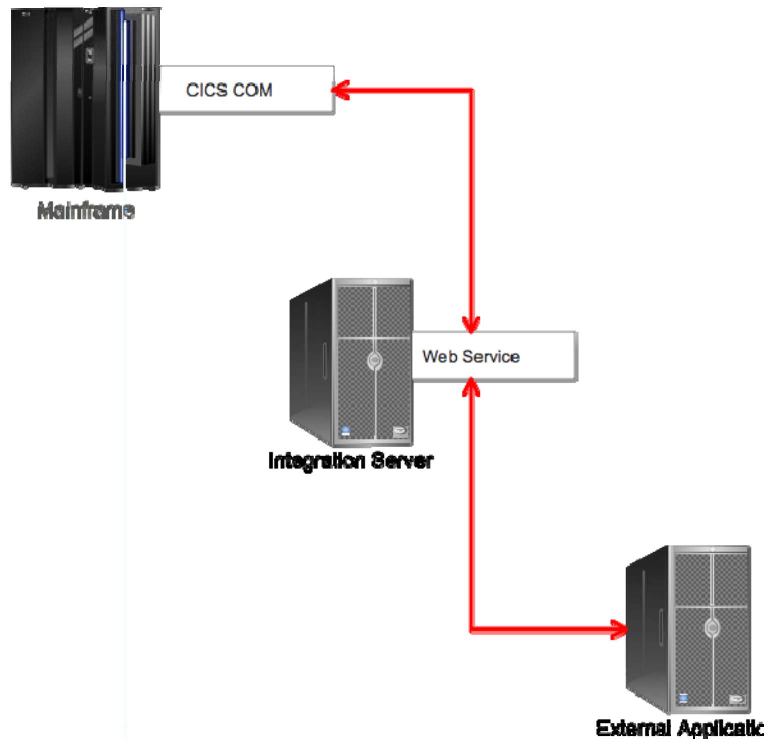

Figure 3: Transaction integration with a CICS or IMS application.

This technique offers a lot more flexibility than the others because you're working with the data at a low level. However, this isn't without risk: Because you're bypassing higher levels of code, you'll have to ensure that you don't break things like data dependencies. So the additional flexibility of this technique is offset by the greater responsibility you take on yourself.

Keep Your Mainframe—and Gain Agility

There are countless business reasons for getting your mainframe application services *out* of the mainframe. You can implement new business processes and new business capabilities and help keep your organization as agile as possible, ready to adapt to changes rapidly. Repackaging mainframe applications through the use of an integration engine that uses the techniques I've discussed in this guide, offers the fastest way to connect your mainframe applications to non-mainframe applications, using modern software interfaces such as Web services, .NET Framework, and Java. You essentially turn entire mainframe applications into sets of middle-tier components, which can be utilized by modern software built on standardized platforms, protocols, and techniques.

Best of all, these techniques require *no* changes to your mainframe hardware or its applications. This helps reduce the skills required to implement these techniques, helps reduce the risk to your critical data and applications, and allows you to leverage the existing business logic that's wrapped up in your mainframe applications.

There's no doubt that mainframe applications can play a major, positive role in the modern business world—with the right integration.