ORACLE
FUSION MIDDLEWARE

An Oracle White Paper
December 2013

# Back-end to the Future:
# Using your Existing Oracle ADF Applications as a Pillar of your Mobile Strategy

ORACLE

## Disclaimer

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

## Introduction

Mobility has been a significant trend in IT for several years now. Laptops have been outselling desktops since 2008[1], and it seems that tablet shipments will surpass desktops in 2013 and laptops in 2014[2]. In that context, the real question for your organization is not *if* it will build mobile applications, but *when*. To negotiate this transition successfully, one cannot ignore other technology trends. The most influential one is probably the *commoditization* of personal computers and smartphones; traditional differentiators have become meaningless and price itself has become the main purchase criterion. In that context, advanced capabilities that were the hallmark of high-priced workstations and servers ten or fifteen years ago, such as multi-core processors, have become widespread. Application developers quickly took advantage of the situation. This, in turn, greatly influenced expectations towards technology in the workplace. Every day, business users access web sites with rich AJAX interfaces; every day, they download and install mobile applications that exhibit superior production values. The impact on IT departments everywhere is clear: new applications cannot simply fulfill functional requirements. They must do so in a fashion that will exceed the heightened expectations driven by technology commoditization and ubiquitous innovative web and mobile user interfaces.

Since mobile applications are the future of computing, the future, then, will require significant investments. Obviously, a mobile application offering a great user interface will be more costly to build than a run-of-the-mill web application meant for simple data entry. On the other hand, even the best mobile application will fail to get traction with its intended user base if it performs

[1] TechHive.com, *« Notebook PCs Outsell Desktops, First Time Ever »*, December 23 2008. Retrieved on May 8 2013. http://tinyurl.com/c7rksyx

[2] IDC, *« Worldwide Smart Connected Device Market Crossed 1 Billion Shipments in 2012, Apple Pulls Near Samsung in Fourth Quarter, According to IDC »*, 26 March 2013. Retrieved on May 8 2013. http://tinyurl.com/bvqr93k

poorly due to an underperforming or unstable back-end. Nevertheless, how can an organization build a robust back-end for mobile applications without diverting too many resources to it? Service-enabling existing applications is a possible avenue, but is a delicate proposition depending on the architecture, the systems and the toolkit used to build them. Fortunately, developer productivity is the core tenet of Oracle ADF. This is why the framework puts so much emphasis on reusability. Developers can reuse and share Entities, View Objects, Data Controls, Task Flows, Pages, Fragments and Components throughout multiple applications. The benefits are twofold:

- Every new line of code in a system is a potential bug. Reuse reduces the footprint of the code base, thus making it easier to maintain its quality level. In addition, existing code has probably been purged from most anomalies, which will result in a more stable and scalable back-end.

- Reusing existing assets increases your return on investment on them, and reduces the resources needed to build new ones.

In this light, building a service-oriented back-end for your mobile applications on the top of your current applications makes perfect sense.

The aim of this white paper is to show you how to use the Oracle ADF applications you already have as one of the pillars of your mobile back-end. We will do so through a case study based on the *Summit ADF* sample application[3]. Beforehand, we will consider the basic principles of service-orientation as well of the challenges that are specific to mobile applications through the

[3] See: Grant Ronald and Lynn Munsinger, *A Case Study in an Oracle Forms Redevelopment Project to Oracle ADF*, July 2011. http://tinyurl.com/c7bwqy6 Retrieved on May 8 2013.

lenses of an ADF developer. We will also discuss in depth the two main service styles currently available to ADF Developers: *REST* and *SOAP*.

## SOA design principles

Most mobile applications need to interact with back-end systems, whether they fetch data for display purposes or initiate new business transactions. In ADF Mobile, this typically happens through web services. Service-oriented architecture (SOA) is thus the foundation on which you will build your applications. Consequently, it is essential to master SOA's design principles in order to build strong mobile back-ends.

### What about APIs?

Nowadays, when a freshly incubated technology startup reveals itself to the world for the first time, its founders will usually dedicate a significant portion of their announcement to the API[4] their product offers. Google, Facebook, Twitter and a myriad others all offer some sort of publicly available web services, typically using the RESTFul style and the JSON data format. In fact, such services have nearly become synonym with API. Is that to say that SOA is now irrelevant? Not at all.

At its core, an API is the interface to a service. There is a difference, however, between a service and an interface. The former is focused on the provider and the technical implementation; the latter is all about the functionality and the service consumer[5]. Thus, while integrating APIs to your business model does not necessarily make sense right now, building services today will give you the necessary materials if the need arises to offer one. In addition, the API model and its related patterns are well suited to mobile use.

APIs may have a disruptive impact on your business model, but the core tenets of SOA still hold true.

### What is a service?

SOA is not about a specific technology, but is rather a state of mind: an architectural style that emphasizes loose coupling and dynamic binding between services. It is possible to build services in a wide variety of technologies, since SOA assumes the implementation details are abstracted from the service consumer. A service, in its simplest expression, is a publicized package of functionality. This means a service must be *discoverable* by potential consumers, based on *metadata* that describes the service[6]. An explicit *contract* with the service provider often governs service usage. This contract is a

[4] Application Programming Interface.

[5] Sanjiva Weerawarana, *API Management: the missing link for SOA success*, WSO2.com, August 31 2012. http://tinyurl.com/pcu8ydr Retrieved on September 28 2013.

[6] Sanjiva Weerawarana *et al.*, *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. Upper Saddle River, Pearson Education, © 2005, p. 13.

comprehensive description of a service interface, which includes the technical interface (signature), the semantics, and nonfunctional aspects such as service-level agreements[7].

From a technical standpoint a service is comprised of three parts[8]:

- the implementation (deployed code and configuration of infrastructure),
- the interface (means by which the Service is invoked), and
- the contract (a description of what the Service provides and its constraints)

It is possible to build complex processes by assembling smaller, simpler processes together. This process is called *composition*. There are two main composition styles[9]:

- *Orchestration,* where a top-level service has central control over the process and calls other services.
- *Choreography*, where rules and policies define how services collaborate to form a process.

From the consumer's point of view, composition is completely transparent since complexity and implementation details are encapsulated.

[7] SOA in Practice: SOA Glossary. http://www.soa-in-practice.com/soa-glossary.html Retrieved on May 15 2013.

[8] Samrat Ray *et al.*, *Maximize the Benefits of Oracle SOA Suite 11g with Oracle Service Bus*

[9] Sanjiva Weerawarana *et al., Ibid.,* p.18.

## From services to web services: the Enterprise Service Bus

SOA is an architectural concept, an approach to building systems; web services, on the other hand, is one possible approach to build a SOA[10]. Since they rely on industry standards, such as the HTTP protocol, the JSON format[11] and the XML language, web services have been widely adopted by the IT community. While they provide good interoperability, web services alone cannot guarantee the loose coupling and dynamic binding of services that are core SOA tenets. Thus, the concept of an *Enterprise Service Bus* (ESB) has emerged over time.

The ESB is a message-driven distributed backbone for a SOA. Its main function is to abstract communications between services.[12] The documentation for Oracle ESB states: *« An ESB provides* […] *an effective approach to solving common SOA hurdles associated with service orchestration, application data synchronization, and business activity monitoring. »*[13] Typically, an ESB will handle three central duties[14]:

- Messaging: asynchronous store-and-forward delivery with multiple qualities of service
- Data transformation: XML to XML and JSON to XML
- Content-based routing: publish and subscribe routing across multiple types of sources and destinations

In addition, an ESB will usually support message delivery over a wide variety of protocols and in multiple data formats. Consequently, it is an essential tool to deploy in organizations where the computing environment is heterogeneous. The shift towards open public APIs, driven in part by the needs of mobile applications, reinforces the need for the reliable intermediary that is the service bus. With an ESB, your internal infrastructure can evolves at its own rhythm; the bus will bridge any gap in technology support and will implement the API resources your customers and partners need.

## The problem of synchronicity

In a software architecture, there is no coupling more constraining than the requirement of having two or more systems available at the same time. In fact, synchronicity is probably one of the biggest limitations to the scalability of a solution. Consequently, although most of the technologies currently used to build services support synchronous calls, most SOA implementations favor asynchronous exchanges.

---

[10] Steve Graham *et al.*, *Building Web Services with Java: Making sense of XML, SOAP, WSDL and UDDI*. Second Edition. Indianapolis, Sams Publishing, 2005, p. 17.

[11] RFC 4627 is the specification for JSON.

[12] Arnon Rotem-Gal-Oz, *SOA Patterns.* Manning Publications, 2012, p. 164.

[13] *Oracle® Fusion Middleware Concepts and Architecture for Oracle Service Bus 11g Release 1 (11.1.1.7)*, section 1.1.2. http://tinyurl.com/bzusv27 Retrieved on May 15 2013.

[14] *Idem.*

Because it abstracts protocols and data formats, the ESB is a great enabler for service loose coupling. The fact it is message-driven pushes this advantage even further. Asynchronous messaging offers many advantages, although it requires developers to make use of transactions in a different way. In a SOA, it is perfectly possible to handle a long-running transaction over the span of several messages; the processing of each individual message, however, will be atomic - and this will probably result in a commit operation at the database level. Thus, each message must contain a unique identifier, also called *correlation identifier,* to specify to which transaction it belongs.

Obviously, some scenarios require synchronous interaction between the user interface and the services. In such cases, it is possible to simulate synchronous interaction by freezing the user interface while waiting for a callback from the service bus. However, this is relevant to transactional scenarios only. The added overhead and latency are not worth the cost in the case of simple queries on reference data, for example.

## Being stateless is not an option

Asynchronous service calls decrease coupling and enhance scalability. Their main disadvantage is that they introduce a level of uncertainty in the processing. The contracts for the services and the service bus define a range for that uncertainty, but do not eliminate it. In addition, factors external to software, such as outages or human decisions, can block long running transactions. In that context, it does not make sense to hold on to precious resources such as processing power and memory space. SOA service calls are thus, by definition, stateless. In addition, the technical foundations of most SOA implementations, such as the HTTP protocol, are stateless by design.

Extensions to the base SOAP protocol can bring state management on top of those stateless foundations. This does not mean that the services themselves are stateful, but rather than the infrastructure will handle state persistence and restoration. Typically, this involves using a relational database as a state persistence store. This mirrors, in a way, what happens inside typical web applications, where various memory scopes, such as the HTTP session, keep track of the state for each user.

When repurposing such an existing application as a service back-end, it is necessary to ensure the service layer is stateless, but without impeding state management for the web front-end. Fortunately, the configuration for ADF BC Application Modules is distinct from their implementation. It is thus possible to create distinct instances at runtime from the same code base, each instance using a configuration tailored to a specific use case.

## Flexibility through service facades

State management is not the only challenge you will have to tackle when reusing an existing application as the basis for a web services layer. Balancing the stability requirements of the service contract with the lifecycle of a production application is equally important. In most cases, incompatible changes to a service interface, such as changes in parameter types and method names, will break existing service clients. This is only a minor inconvenience when only a few internal systems use a service, but could spark outrage among customers in the case of a widely used mobile application. On the other hand, tying application updates to the service contract would be unwise, since such an approach reduces the

organization's agility. The tools you offer your non-IT colleagues need to evolve at the speed of today's rapidly changing marketplace to be relevant.

A good way to address both concerns is to make use of the service facade pattern[15], illustrated below.

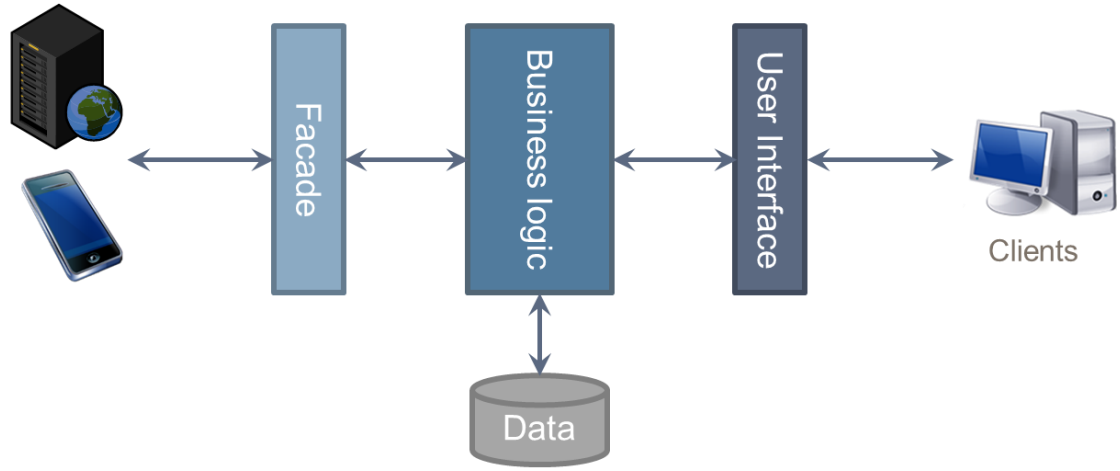



Figure 1. The service facade pattern.

Its main advantage is to decouple the service interface from the application's internal logic. Thus, it is possible keep the former stable while having the flexibility to update the latter. Service facades have the added benefit of encapsulating the complexity of the application; the service facade only exposes what consumers actually need, as defined by the contract.

## Other important issues

Service facades, while useful, do not address the full range of architectural issues raised by SOA. Service versioning, for example, is essential and requires both strong governance processes and effective coding conventions. Without it, some long-running transactions could fail because of changes in the service's interface or business logic. Navigation in web service result sets also needs careful consideration; it must be implemented in a consistent way to enable developers to be productive, as standard Java-based web service stacks do not handle this on their own. Long-running transactions can be tricky to handle, since they must be saved to permanent storage in order to free the resources they consume while waiting for the next message. When considering mobile web service clients, a crucial issue is offline transactions. Network outages and spotty coverage mean such clients must have means to cache result sets and queue new transactions. Back-end services must be able to process the latter and handle potential collisions since the asynchronous nature of SOAs means services lack locking semantics in the name of better scalability.

[15] See it described on Soapatterns.org. http://tinyurl.com/pugdkss Retrieved on May 28, 2013.

## From theory to practice

As you can see, SOA implementations are rife with complex issues. However, clever design decisions in ADF and ADF Mobile provide you with the tools to mitigate them. Furthermore, various products in the Fusion Middleware family can help you increase the availability and scalability of your architecture.

Oracle ADF is a great foundation for SOA-enabled applications, since reusability is one of its core tenets. Entities can take part in many View Objects; View Objects can participate to the data model of several Application Modules. Each Application Module, in turn, can be tuned for a specific usage model through its configuration. Thus, whereas ADF web applications typically contain a single root Application Module where others are nested, ADF web service layers will usually sport several root application modules, each configured for stateless operation. In addition, it is straightforward to implement meaningful service facades through View Object inheritance, as shown in the case study that concludes this white paper. Service-enabled application modules also make it trivial to expose asynchronous web services, supported behind the scenes by JMS queues.

It is certainly possible to build an Enterprise Service Bus on the top of Oracle WebLogic Server alone. Nevertheless, if you are designing a SOA infrastructure, Oracle Service Bus (OSB) certainly deserves your consideration. On the top of its routing and XML transformation features, it supports a wide array of non-xml data sources, such as flat files, EJB, FTP, MQ, JMS and Tuxedo. In addition, it provides everything you need to integrate with Siebel and PeopleSoft. OSB also features one-click enablement of service result caching, implemented through Oracle Coherence. Consequently, it can not only improve service decoupling in your architecture but also reduce the latency of service requests. This is especially important in mobile scenarios, since SOAP service endpoints and REST resources must offer a streamlined interface adapted to the processing and bandwidth limitations of the devices.

Governance, as we have hinted earlier, is an essential component of a SOA. It also plays a fundamental role in making APIs viable. Without it, anarchy quickly plagues services development, giving birth to a *bunch of services* instead of an architecture. Effective governance is difficult to implement, however. This is why Oracle provides a comprehensive and integrated offering called the *SOA Governance Suite*. The various technical components of the suite all implement default processes based on recognized practices that you can tailor to your own needs. In a developer perspective, the main components of the suite are:

- *Oracle Enterprise Repository*, which provides design-time governance support for the service lifecycle, enabling storage and management of extensible metadata for composites, services, business processes, and other IT-related assets.
- *Oracle Service Registry,* a UDDI version 3 compliant service registry. It provides service binding and runtime location transparency, as well as access to an appropriate service version based on the environment. You can look up services published to the registry at runtime for dynamic service location.

- *Oracle Web Services Manager* allows IT management to define centrally security policies that govern Web services operations (such as access policy, logging policy, and load balancing).

Web Services Manager is not the only product that you can use to secure your web services. OWSM is suitable for use inside your organization's network and, at its boundary, to secure endpoints. To extend your enterprise security further to cloud and mobile applications, Oracle API Gateway is the proper Fusion Middleware component to use. Among other things, API Gateway can inspect inbound and outbound cloud traffic on the wire and validate a wide range of web services-related artefacts, such as HTTP parameters, REST query/POST parameters, JSON data structures and XML schemas. It also offers protection against cross-site scripting (XSS) and denial of service attacks (DoS).

## Considerations for mobile applications

Up to now, we have considered things purely in a server-side perspective; we have covered how to architecture a generic service-oriented backend. Now is the time to add some specificity to our reflection. A well-thought back-end for mobile applications needs to address issues that are specific to that environment.

### About Mobile Networks

Mobile networks are now ubiquitous. The theoretical bandwidth and download speeds they offer are extremely impressive, as is the speed at which the industry introduces new technologies. After decades of improvement, mobile networks are still unreliable. Even in urban areas benefiting from good coverage, it is sometimes difficult to get a steady connection. Do not forget: signal strength indicators exclusively measure how well the phone can connect to the infrastructure, but do not consider network congestion, for example[16]. Thus, it is possible to get miserable bandwidth even while standing at the foot of a cellular phone tower. Moreover, while it is now possible to get generous broadband subscriptions for a reasonable price, the cost per megabyte of mobile data subscription plans is typically much higher. One of the reasons for this is that mobile networks are costlier to scale than broadband ones[17].

An enterprise-level mobile application cannot make optimistic assumptions about the network it will run over. Consequently, when building your ADF Mobile application, you must assume that the network will be unreliable and slow, and that bandwidth is expensive. This will directly influence your architecture. Most, if not all, the data you will retrieve though web service calls must be cached. In addition, if you mobile application initiates business transactions, it will preferably record them in a

[16] Marguerite Reardon, *5-bar phone signal: What's it get you? (FAQ)*, CNet.com, July 2010. http://tinyurl.com/32ktg9t Retrieved on May 30, 2013.

[17] Michael Mace, *The Truth about the Wireless Bandwidth "Crisis"*, Mobile Opportunity blog, June 2011. http://tinyurl.com/mqt9wnj Retrieved on June 3, 2013.

transaction buffer on the device before making the web service call. The application will then discard this local copy once the business process dictates it, typically right after a successful invocation of the web service. The local database feature of ADF Mobile, which relies on the SQLite embedded database, is well suited to that use case.

Properly managing the transaction is more complex than it appears, though. By default, there are no locking semantics on web service calls. This means two users are susceptible to edit or delete the same record in parallel. There are two ways to handle this. First, you can add *locking* to your business logic, either by making use of WS-BusinessActivity (for SOAP web services) or by rolling your own. This adds complexity to your back-end and reduces its scalability, since only one client at a time can change a specific record. You will need to handle collisions, such as when a user tries to update a record deleted by another just before, properly. Users will notice degradation in their experience if the resources are subject to high contention. In addition, this approach only applies to short-lived transactions. Your other option is to implement *compensation*. Each party to a transaction must then store all the data necessary to proceed to a rollback, among other things. Once again, the ADF Mobile local database can help, although simplification of the business process is often a more productive way to handle such tricky use cases.

## Resources: still limited

Mobile devices have come a long way in the last few years. They now possess multi-core processors, vast amounts of RAM and gigabytes of storage. Desktop computers - and even servers - of a decade ago did not offer that much power. There is a fundamental difference, however, between yesterday's PCs and today's mobile powerhouses. The former were usually *production* devices, which one would use to build colossal spreadsheets or develop a complex e-commerce web site. The latter are fundamentally *social* and *consumption* devices; their owners need them to communicate with their peers and access a wide variety of digital media, such as web sites, ebooks, music, videos and games. Consuming such media require significant computing power; gaming requirements alone probably explain why the graphics performance of today's mobile devices rivals that of mid to high-end desktop computers from 2006. [18]

While the smartphones and tablets we use in 2013 offer ample resources, any given application should still use those as if they were scarce. On a typical device, tens of applications will vie for them. And finding more nimble alternatives to a resource-hogging application is easy. Mobile operating systems offer deep integration with application stores, which effectively remove distribution as a barrier of entry in that specific market.

Fortunately, ADF Mobile makes it easy to manage resources since it is built on the top of the Java virtual machine. In addition, application and feature level listeners enable developers to execute specific

[18] Anand Lal Shimpi, *The Great Equalizer 3: How Fast is Your Smartphone/Tablet in PC GPU Terms*, Anandtech.com, April 2013. http://tinyurl.com/bwhwjlj Retrieved on June 11 2013.

code when the application is activated or deactivated. Such logic, however, must be supported by the service back-end. This is especially important in the case of in-flight transactions, since service calls themselves are stateless. Once again, this highlights the importance of correlation and compensation.

## The price of mobility

Mobile devices are all about convenience. They enable near-instant access to mission-critical applications from nearly any place. The downside of this convenience is vulnerability. With mobile devices, disaster is never far and physical security fairly weak. Any given device can be lost, stolen, dropped or even worse. In addition, the myriad of wireless interfaces they offer (Bluetooth, wifi, and NFC) are as many vectors of attack for malevolent parties. Fortunately, mobile applications themselves can do their part to ensure the security of data and transactions. The ADF Mobile local database, for example, can be encrypted. In addition, access to specific features or the entire application can be password-protected. And web service calls can be secured through the use of SSL/TLS. On the other hand, service calls routed through public networks should never be trusted.

Here, the deep integration between the components of Fusion Middleware is your best ally. ADF mobile applications can authenticate against existing Oracle identity management (IDM) back-ends out-of-the-box. Moreover, Oracle Web Services Manager and Oracle API Gateway can contribute to the enforcement of your organization's security policies with little to no impact on the code of mobile applications. This is because ADF Mobile web service security is policy-based, at least when SOAP based services are used. Furthermore, ADF Mobile applications fully support role-based authorization. In the current 11.1.2.4 release, the list of roles for a specific user is obtained by calling a custom service implementing a specific interface. This service, if built with Oracle ADF, can delegate the actual role retrieval to the IDM back-end, and even make use of specialized middleware such as Oracle Entitlements Server[19].

Including IDM and security products in your architecture, while helpful, is not enough. Risk mitigation, including validation and audit, must be present in every layer of your services infrastructure. In that context, service facade play an important role in rejecting unauthorized requests. The powerful validation mechanisms found in ADF Business Components can play a role in this and make you more productive, but are hardly enough. Special care must be given to long-running transactions, as the authorizations for the various parties involved may have changed over time. Typical examples include employee dismissal or reassignment. Thus, the services must be built in a way that will ensure transactions fail gracefully instead of getting through in such scenarios.

[19] See http://www.oracle.com/technetwork/middleware/oes/overview/index.html

## Service styles: SOAP, REST or both?

The guidelines we have laid down apply independently of the technical platform the services are built on. However, web services are used in an overwhelming number of cases. There are very good reasons for that. Web services have been designed to be interoperable, and are available in all current operating systems and in most programming frameworks. In addition, they are implemented through lighter protocols and are easier to secure than legacy technologies such as RMI and Corba. There are currently two distinct web service styles in use: SOAP and RESTFul. Deciding between the two is a matter of context; each style has its own strengths and weaknesses.

### SOAP: enterprise-level features

SOAP is an XML based protocol. The term stood for Simple Object Access Protocol at the technology's inception, but this is not the case anymore. SOAP is protocol-neutral by design. Typically, implementations use HTTP, but some platforms support other transport protocols such as SMTP (Simple Mail Transfer Protocol) and JMS (Java Message Service)[20], among others. When working with SOAP, development tools usually generate the client APIs from the service's metadata. This metadata is made available through a WSDL (Web Services Description Language) file – which is an XML descriptor using a specific language.

SOAP is a WC3 (World Wide Web Consortium) recommendation. Its designers built it with extensibility in mind. The most widespread of these extensions are collectively known as WS-*; they are developed under the stewardship of the OASIS Consortium (Organization for the Advancement of Structured Information Standards). WS-* extensions cover a wide array of functional areas. Here are a few extensions you should be aware of:

- Messaging
  - WS-Addressing: adds support for non-HTTP protocols
  - WS-ReliableMessaging: defines a set of delivery models for messages, such as « at least once », « at most once », « exactly once » and « in order ». Such models enable reliable delivery even in the occurrence of software, system or network failures.
- Security
  - WS-Security: covers integrity, confidentiality, and security tokens
  - WS-Trust: concerns itself with the issue, renewal and validation of security tokens
  - WS-SecureConversation: creation and sharing of security contexts
- WS-AtomicTransaction: Two-phase commit support for distributed ACID transactions

[20] Support for SOAP over JMS has been introduced in JDeveloper 12c (12.1.2).

The main advantage of SOAP is its broad standardisation and the interoperability guarantees it provides. Its focus on enterprise-grade features make it well suited for a variety of complex scenarios. However, its reliance on XML makes for verbose data exchange messages. Thus, SOAP messages are usually slower and more resource intensive to process than those sent using competing technologies.

### RESTFul: a lightweight alternative

In many ways, the RESTFul (REpresentational State Transfer) architectural style is the polar opposite of SOAP. It supports a single protocol (HTTP), possesses no standard API, does not mandate specific data formats and is bereft of a standard definition language for the interfaces[21]. By convention, RESTFul services rely on four on the eight standard HTTP methods to perform specific operations:

- GET (Retrieve a resource)
- POST (Create/Update a resource subordinate)
- PUT (Create/Update an entire resource)
- DELETE (Delete a resource)

Moreover, each resource should resolve to an URI. In addition, results should be cacheable.

Nowadays, most RESTFul service implementations rely on JSON (JavaScript Object Notation) as their data format. An open standard[22] derived from the popular JavaScript language, it is specifically designed for data interchange and meant to be readable by humans. Below is a sample JSON structure retrieved from a public RESTFul service.

http://api.rottentomatoes.com/api/public/v1.0/movies/770672122.json?apikey=[key]

```
{
  "id": 770672122,
  "title": "Toy Story 3",
  "year": 2010,
  "genres": [
    "Animation",
    "Kids & Family",
    "Science Fiction & Fantasy",
    "Comedy"
  ],
  "mpaa_rating": "G", …
}
```

[21] Sun Microsystems proposed the Web Application Description Language (WADL) to the W3C to fulfill this purpose in 2009, but the consortium currently has no plans to standardize it. The specification is available at https://wadl.java.net/ This blog post by Ole Lensmar at the API UX blog discusses WADL and popular alternatives: http://apiux.com/2013/04/09/rest-metadata-formats/

[22] See RFC 4627: http://www.ietf.org/rfc/rfc4627.txt

The RESTFul style emphasizes simplicity, and runs on the top of a nearly ubiquitous protocol. The vast majority of web developers already have the skills required to use it. The same can be said of JSON. Typically, RESTFul services will exhibit better performance and scalability in production implementations since they rely on a leaner technology stack and forgo XML for more efficient data formats. This explains why public APIs are usually implemented using a combination of REST and JSON.

### Which one should I use?

The performance advantage of RESTFul services do not mean you should systematically prefer them to SOAP-based ones. The Enterprise-grade features offered by the WS-* extensions, for example, can greatly reduce the complexity of your own code while fulfilling critical functional and non-functional requirements. In addition, the skillset of the development team must be taken into account. Most major IDEs offer extensive support for SOAP-based services and WSDL, whereas the RESTFul style requires a certain level of technical familiarity with HTTP.

SOAP versus REST, in our opinion, does not have to be an exclusive choice. Using both makes sense for most organizations. Nevertheless, how can you select one over the other in a specific context? One rule of thumb we can think of is the focus of the service: is it about data or business logic? Typically, the RESTFul style is well suited to queries and CRUD operations. SOAP, on the other hand, shines when there is a need for complex business logic. As it is often the case, theoretical knowledge is no substitute for experience. Eventually, you will determine what works best for you and your team.

Oracle development tools will support you whatever your choice will be. The 11g releases of Oracle JDeveloper, for example, enables you to expose ADF BC view objects as SOAP services through service-oriented application modules. An upcoming release of the 12c branch will enable you to expose them as REST services as well. Moreover, ADF Mobile applications can consume both SOAP and REST services with ease. Thus, you choice is in no way constrained by the capabilities of the tools.

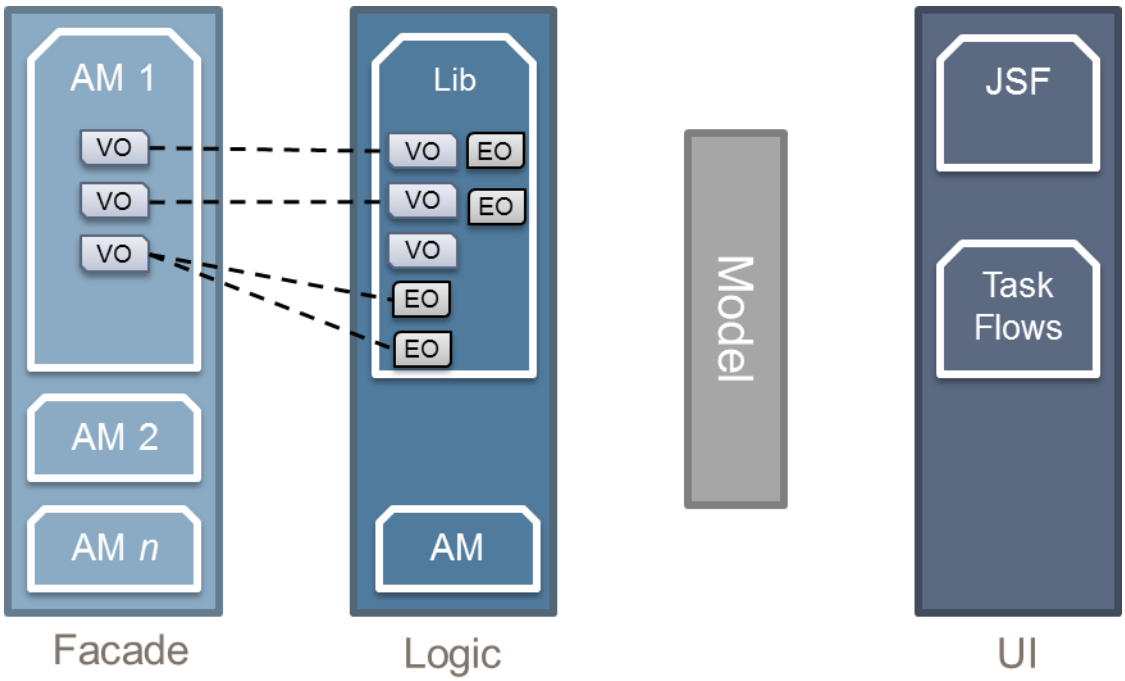
## Exposing services on Summit ADF: a case study

There are probably as many ways to build a SOA as there are software architects. One of the fastest, however, is to base your services layer on existing applications. This makes sense on several fronts. Financially, using assets you already have enhances ROI and reduces risk. In addition, the code for those applications is typically of higher quality than new code, since its bugs have surfaced through real-world usage. Finally, reuse greatly increases developer productivity and reduces time-to-market. This is not to say that this is a perfect, worry-free approach. Services built on the top of existing applications are dependent on them. There is a possibility that changes made to the application will break the service layer, or that the evolution of the application will be constrained by service layer requirements. There are many ways to mitigate those issues. Strong governance and an effective software development lifecycle are a start.

Reusability is one of ADF's main concerns. This is why you can package business components and task flows inside libraries, for example. This also explains why it is so simple to create web services from View Objects. Since ADF is an object-oriented framework, you can also achieve greater

reusability through inheritance and polymorphism. Thus, entity and view object can derive from others and even be used in a generic way if the inheritance hierarchy allows it.

The figure below illustrates one of the many possible implementation of the service facade pattern using ADF Business components artifacts.



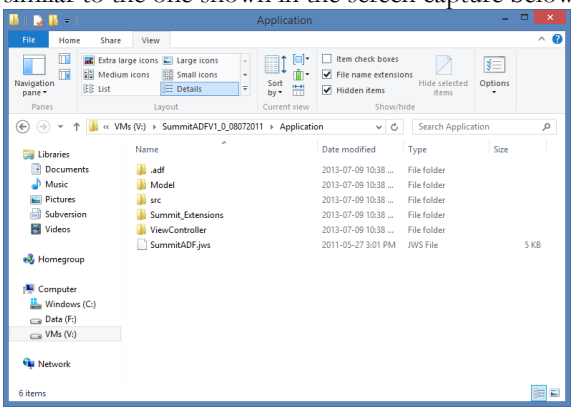

As you can see, the service facade and business logic are completely isolated from each other. The facade contains view objects only. This makes sense, as it does not access data on its own but uses the existing Entity Objects. On the other hand, the business logic is exposed as an ADF Library. The facade's VOs inherit from the application's existing VOs or can be completely new. In that case, they simply use the Entity Objects available in the library.

Please note that, even when View Objects inherit from others, the View Links are not taken into account. If you create a brand new child VO, you will have to create the links you need in the facade project. Entity associations are still available, however. You can base your new links on them as needed.

## A step-by-step process

Implementing the pattern as described above is a straightforward process, already described in detail in an ADF Insider recording[23]. Here, we will simply summarize the steps required. Our starting point is the Summit ADF Sample application, introduced in a 2011 Oracle white paper[24]. The source code and database schema are available on the Oracle Technology Network[25].

1. Download and unpack Summit. Open the *Application* subfolder. You should see a structure similar to the one shown in the screen capture below.




2. In JDeveloper, create a series of applications using the *Generic Application* template. The names for these applications should reflect their role in the system. The roles we propose are:

   - *Model*: Shared business logic. Contains the Entity Objects and View Objects used by the web application and services layer, as well as Application Modules.
   - *View*: Hosts the ADF Faces web application.
   - *Services*: Contains the code for the service layer.

   In addition, Summit contains a project dedicated to shared business logic. In this case, we will bundle it with the *Model* project although we could have spun it out as a standalone ADF library.

---

[23] *Building Web Services From an Existing ADF Application* http://youtu.be/6QAkq3bI97M

[24] Grant Ronald and Lynn Munsinger, *A Case Study in an Oracle Forms Redevelopment Project to Oracle ADF*, July 2011. http://tinyurl.com/c7bwqy6 Retrieved on June 20 2013.

[25] The version built for JDeveloper 11gR1 (11.1.1.x) is available here: http://download.oracle.com/otn_hosted_doc/jdeveloper/11gdemos/SummitADF/SummitADFV1_0_08072011.zip A version adapted for ADF 12c (12.1.2.x) is also available: http://www.oracle.com/technetwork/developer-tools/jdev/learnmore/index-098948.html

Thus, you should have three applications named *SummitModel*, *SummitView* and *SummitServices*. Ensure you remove the default project (typically called Project1) from each application and delete its contents.

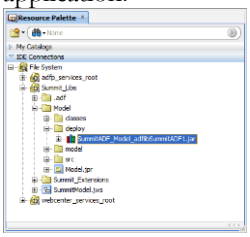
3. Take the *Model* and *SummitExtensions* projects of the original application and move them to the *SummitModel* application. Add the projects to the application in JDeveloper. Ensure the database connection is valid for your environment.

4. By default, *Model* projects are given an *ADF Library JAR file* deployment profile when they are created. You need to change this profile. Specifically, you must add a dependency to the build output of *SummitExtensions* to it. Create the JAR file for the *Model* project inside the *SummitModel* application.

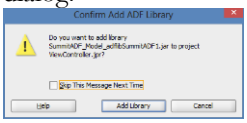
5. Move the *ViewController* project to the *SummitView* application. Add the project to the application in JDeveloper.

6. In JDeveloper's *Resource Palette*, create a file system connection in the *IDE connections* category. Point it to the root folder of the *SummitModel* project (for example: `v:\SummitADF_SOA\SummitModel`).

7. Through the connection, locate the JAR file for the *Model* project of the *SummitModel* application.



Right-click on it and select the *Add to Project*…option in the contextual menu. You will get this dialog:



Simply click on the *Add Library* button to confirm the operation.

8. In the *Resource Palette*, open the contents of the library JAR file. Locate the database connection and add it to the project. You will need to re-enter the password for the connection.

9. At this point, you are ready to populate the *SummitServices* application.

How you will proceed from there entirely depends on which style of web services you intend to use. For SOAP, you simply have to create one or more Application Modules and create service interfaces

for them. A future release of ADF 12c (12.1.x) aims to enable you to create RESTFul service interfaces as easily. However, building RESTFul services is a bit more involved if you are using ADF 11g release 1 (11.1.1.x), 11g release 2 (11.1.2.x) or early versions of 12c (12.1.2). In the first two cases, you may need to obtain the JARs of a suitable JAX-RS implementation[26], and add it to the project dependencies. This is not needed in the case of 12.1.2, since JAX-RS is part of the Java EE standard since version 6. Then, you will implement the service endpoint through a plain Java class with annotations. When invoking ADF BC business logic from the REST resource, you will create application modules instances directly from the Java API. You should structure you code in a way that will ensure that all instances are released once they are not needed anymore. It is highly recommended to release those instances to the application module pool if possible, since this will enhance the scalability of the service layer. Please note an ADF Insider Essentials recording made available in October 2013 describes the whole process[27].

Here is some additional guidance about service implementation:

- SOAP and RESTFul services must have distinct context roots. Thus, if you want each style to share their application modules and view objects, each style should have its own project in the *Services* application. We suggest to use *ServiceFacade* and *RestServiceFacade* as project names; the JEE context roots could be *services* and *rest*, for example. On the other hand, SOAP and RESTFul services should each get their own JDeveloper application if you want each style to have distinct application modules and view objects.
- Your application modules must have the proper granularity. Ideally, you will have enough of them to tune each service properly with acceptable overhead. Too much is as bad as too little in that particular context.
- If you have services exposed outside your organization, do not hesitate to create distinct projects for them. You can easily create a separate archive for these services by using a different deployment profile.

## Conclusion

Mobile applications are not just the future; they are the present. Already, they reshape not only how we do business but society itself. The opportunities they open are boundless. To capitalize on these opportunities, you need reliable and scalable back-ends.  SOA is probably one of the best ways to build them.

[26] The JAX-RS reference implementation is Jersey, a component of the GlassFish project. https://jersey.java.net/

[27] http://youtu.be/jDBd3JuroMQ

An API is the interface to your services. And a service is simply a discoverable package of functionality. The design principles of SOA are straightforward: messages, asynchronicity, statelessness and facades are all simple in concept yet difficult to implement. Fortunately, Oracle ADF simplifies implementation and enhances developer productivity. Other products in the Fusion Middleware family, such as Oracle SOA Suite, facilitate the implementation of more complex scenarios. In addition, JDeveloper 12c will eventually add easy generation of RESTFul services to its already strong SOAP support – which is good news as you should use both.

Mobile applications come with their own set of challenges: limited resources and unreliable networks still pervade the industry. Oracle ADF and ADF Mobile make it easier to tackle those challenges for you, your customers and your partners.

Reusing your existing ADF applications as one of the pillars of your service layer is probably the most efficient way to build robust back-ends. As you have seen, the process is simple. Once again, the JDeveloper motto of *Productivity with Choice* proves true.

Back-end to the Future: Using your Existing Oracle ADF Applications as a Pillar of your Mobile Strategy
December 2013
Author: Frédéric Desbiens

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200

oracle.com

Oracle is committed to developing practices and products that help protect the environment

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

This document is provided for information purposes only, and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document, and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group. 0113

Hardware and Software, Engineered to Work Together