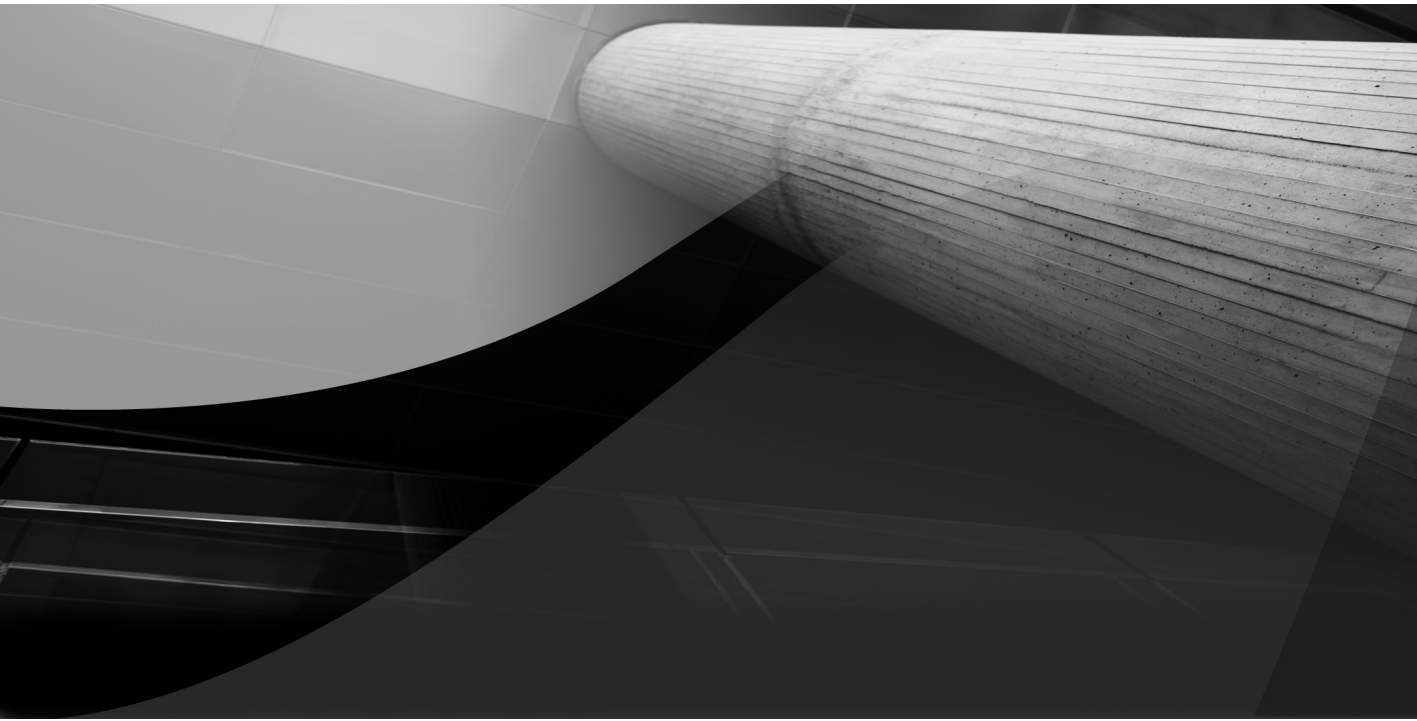


PART I

Overviews

*There is nothing more difficult to take in hand,
more perilous to conduct, or more uncertain
in its success, than to take the lead
in the introduction of a new order of things.*

—Niccolò Machiavelli (1469–1527),
The Prince



CHAPTER 1

Overview of Fusion Development and ADF

4 Oracle JDeveloper 11g Handbook: A Guide to Oracle Fusion Web Development

When you innovate, you've got to be prepared for everyone telling you you're nuts.

—Larry Ellison (1944—)



Over the last few years we've heard an ever-increasing buzz around the term Fusion within the Oracle community as Oracle pushes forward with its next generation of business applications. Fusion means different things to different people, something that we will discuss shortly, but its use as a term to define an architecture and development methodology is the focus of this book.

This first part of the book provides an essential overview of some of the key technologies and tools that you will encounter as a developer working in this space. This chapter introduces some of the key architectural issues and concepts, Chapters 2 and 3 provide an introduction to the JDeveloper IDE and Chapter 4 discusses the key technologies that define the web platform as a whole. Subsequent parts of the book drill into some of the fundamental framework pieces and provide a guided tutorial to help you to build your first Fusion application. We start, though, with this chapter, which provides an introduction to Fusion architecture and technology from the point of view of the developer. In the course of this introduction, it also touches on the Oracle Application Development Framework (ADF), which eases development chores when working with the Fusion platform. The chapter discusses answers to the following questions:

- **What is Fusion?**
- **What is a framework?**
- **What is Oracle ADF?**
- **Why should I use JDeveloper?**

What Is Fusion?

We have deliberately used the term “Fusion” in this book’s title rather than listing specific technologies or frameworks. Fusion as a term is used in multiple places as a brand by Oracle, and of course it’s a popular brand name for many other, nonsoftware products, ranging from razors to cars. Oracle refers to Fusion in two distinct branding contexts:

- **Oracle Fusion Middleware** This is the entire suite of middleware used to run modern, standards-based applications. It provides a runtime environment for applications written to the Java Platform, Enterprise Edition (Java EE) APIs, Service-Oriented Applications, and much more. We’ll be explaining what this infrastructure supplies a little later on in this chapter.
- **Oracle Fusion Applications** This is a project undertaken by Oracle to both unify and modernize its entire suite of packaged business applications. The term Fusion Applications covers packaged applications for financial management, human capital management, customer relationship management, and much more. This is a huge undertaking in software terms involving thousands of programmers working for several years, but it is important to note that the techniques and software used for that project are exactly what we describe in this book.

Oracle Fusion Middleware provides a runtime and development environment for Oracle Fusion Applications, so there is a natural usage of a common branding there. However, when we refer to Fusion in this book, we're really employing a third use of the word: Fusion Architecture. The common link here is that the *Oracle Fusion Architecture* defines the core architecture and technologies used to build the Oracle Fusion Applications products, which in turn will be deployed on Oracle Fusion Middleware (or indeed, in principle, to any other standards-based application server). As continually writing the phrase "an application based on the principles of Fusion Architecture" is a bit of a mouthful, we will tend to use the shorthand form "Fusion application" when referring to such an application. Understand from this that we mean any application architected on these principles, rather than specifically meaning one of Oracle's packaged applications.

The Origins of Oracle Fusion Architecture

Before looking more closely at the details of the Fusion Architecture, let's take a step back and consider what the traditional database-driven application looks like. Note that we will make no apology for concentrating on applications that connect to databases here; the majority of our business applications do so, and since you are reading a book with "Oracle" in the title, we might expect at least a degree of interest in where the data is stored. In addition, although we do not focus specifically on Service Oriented Architecture (SOA) in later chapters, we need to spend a bit of time in this chapter explaining it so you can understand where Fusion fits with SOA.

Taken as a group, the authors of this book have over 50 years of experience working with and writing applications for the Oracle relational database. For most of that time the development pattern has largely been static, although the technologies and platforms have certainly changed. Throughout this period the core development paradigm has been one of a very close match between the user interface and the relational tables in the database. In this model, business logic for an application is either encoded into the application tier in the relevant language—Java, C, or PL/SQL—or into the database in the form of PL/SQL (or occasionally Java). What is more, a lot of key functions within this model are implemented as batch processes, as they are too expensive or complex to run inline. Such batch programs may be responsible for reporting functions, data transformation, communication with other systems, and so on, but every system has them, as shown in Figure 1-1.

Figure 1-1, represents several generations of Oracle tooling, from Oracle Forms with over 20 years of service all the way through to some of the newer Java-based frameworks. This traditional model is obviously very successful; otherwise, it would not have persisted for so long and in so many forms. All of these frameworks and development methodologies share a common bond with the database as the hub of all things. However, the development landscape is rapidly changing, and standards compliance is now a key business driver rather than a nice-to-have. The watchword nowadays is "integration."

The Rise of SOA

When you think of integration today, you think of *Service-Oriented Architecture (SOA)*. SOA is not really anything new; it is a re-expression of an old idea: reusable components that are distributed throughout the network. The attractive propositions behind such a component-based model are that they encourage reuse of functionality and that the components or services themselves can be located in the most suitable place for that function, rather than being embedded within your program. As a simple example, imagine a business process that attempts to audit transactions to look for patterns that might indicate fraudulent activity. A process like this has needs that illustrate

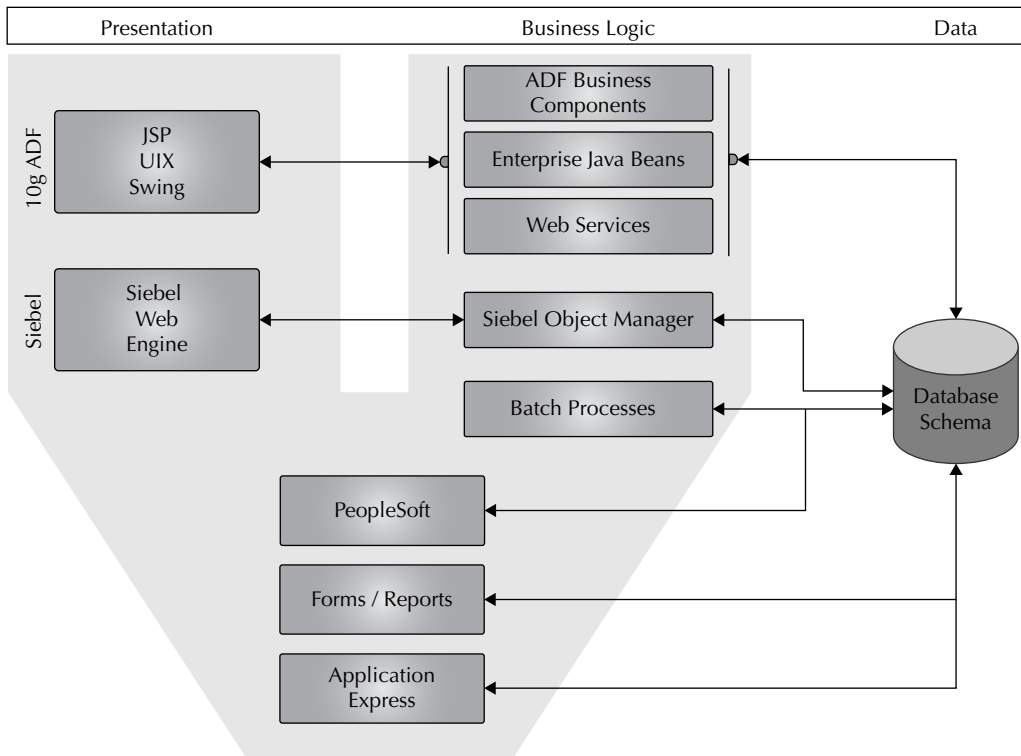


FIGURE 1-1. *The traditional Oracle application architecture*

both location importance and reusability. On the factor of location, such a process will probably crunch through a lot of database information to look for warning signs, so running it on or near to the database will be important. On the factor of reusability, aspects of the process, such as alerting the correct staff, need access to the corporate escalation map. This escalation processing is likely to be reusable throughout the enterprise, so it should not be programmed into the core audit function.

SOA is all about using and assembling these services without compromising their functionality in the process. For example, if you wanted to include this higher-level audit function into your data entry system, you can do so, but without having to worry about the performance aspects of pattern matching or how to establish whom to alert. All of this is encapsulated within the high-level audit service, which in turn might be consuming multiple subservices to achieve its goals.

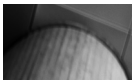
The idea of remote reusable services itself is well established; it has surfaced before in various forms such as Microsoft's DCOM (Distributed Common Object Model) and CORBA (Common Object Request Broker Architecture). Both of these attempts at a distributed service model failed to become pervasive for a variety of reasons such as complexity and cost; however, the most significant factor in the failure of both platforms was the transport mechanisms that they use. Both DCOM and CORBA use low-level TCP/IP protocols to communicate between remote servers and as a result have a real problem when the application needs to cross firewall boundaries or pass between proxy servers. This factor constrained these architectures to use within the enterprise and

prevented their use as a transport mechanism between partners or trading groups. In fact, even within a single enterprise, firewalls will exist to create demilitarized zones (DMZs) for sensitive applications and databases. These firewalls experience the same communications issues when DCOM and CORBA are used.

How Is SOA Different?

There is no doubt that the SOA concept has evolved from and aims to solve the same reuse problems as the earlier attempts at distributed computing such as DCOM and CORBA. However, it has been much more successful for the following reasons:

- **Communication is based on hypertext transfer protocols** SOA primarily uses web services protocols for communication. *Web services* (centrally stored application functions available through the Web) use the Hypertext Transfer Protocol (HTTP) or HTTP Secure (HTTPS) as a transport layer and have already solved the problem of how to traverse firewalls and proxy servers. Additionally the web services community has already solved many of the secondary problems associated with third-party communication, including how to standardize security, encrypt the data, and define levels of service for remote service use.



NOTE

Chapter 4 discusses web communications and HTTP protocols.

- **Through web services, SOA has a common lexicon** These include a service description format in the *Web Services Description Language (WSDL)* and a common discovery and publishing mechanism through the Universal Description, Discovery, and Integration (UDDI) standard, which acts as a yellow pages for services.
- **SOA emphasizes orchestration, not just communication** Protocols like CORBA and DCOM allow programmers to write code that utilizes both local and remote function calls; however, all of that code was at a functional code level. SOA concentrates on the more abstract task of integrating at the service level rather than the function level. Using orchestration languages such as the Business Process Execution Language (BPEL), this task is something that can be achieved, to a certain level, by a suitably trained business analyst rather than requiring a programmer at every step. Generally the process of orchestration itself will involve the use of a diagram to map the outputs of one service to the inputs of another.

Unlike a standard such as CORBA, SOA has not been built top-down by committee. Instead it is much more of an accepted usage model for these existing standard pieces such as web services, BPEL, and UDDI; all of these standards are well served by the Oracle tooling and, of course, provided as runtime implementations by Fusion Middleware or any other standards-based application server.

What Are the Key SOA Technologies to Understand?

In a large enterprise development, core SOA tasks such as service orchestration and creation are likely to be managed by an entirely different team from the team building the UIs for an application. This approach makes a lot of sense; the skill sets are very different, and teams building SOA services need to take the broader view across the enterprise so as to identify areas of common functionality and reuse.

The reality is of course that many developments are conducted on a smaller scale and may not have the luxury of dedicated teams devoted to architecture and shared service development, so even if you will be concentrating on building core Online Transaction Processing (OLTP) screens, you may have to venture into the world of SOA. So, let's look at the key terms and technologies of SOA in Fusion Middleware:

- **Business Process Execution Language (BPEL)** *BPEL* is an Extensible Markup Language (XML)-based language used to orchestrate multiple services together. BPEL processes look a little like flow charts with inputs, outputs, branches, and so forth. Fortunately the diagram-based editors provided by JDeveloper make it easy to pick up and use. Oracle also provides another similar modeling tool, called the Business Process Modeler, which is discussed in the sidebar "About BPM."
- **Enterprise Service Bus (ESB)** The *ESB* acts as a central clearing house for SOA-based transactions. It offers three key functions. First, the ESB acts as an event broker where clients can simply post a message to the ESB and then interested services can subscribe to that event and be triggered on demand. Second, the ESB acts as a service abstraction layer, allowing services to be moved without disrupting any clients that consume them. Third, the ESB provides transformation services to facilitate message translation between different services. Most Fusion applications integrate with the *Mediator*, a lightweight implementation of the ESB (explained further in the section "Business Logic" later in this chapter).
- **Rules Engine** This component provides a standardized way of defining sets of rules for expert systems and complex processes. The rules engine is driven by sets of rules that can be altered at runtime rather than being coded into the application.
- **Human Workflow** This part describes a business process that is not specifically computational but rather concentrates on tasks such as approvals. Workflow systems are driven by an understanding of approval chains, permissions, and hierarchies. As the name suggests, Human Workflow processes involve interaction with a person, which is what distinguishes them from the fully automated processes implemented in BPEL.
- **Composite** A composite is a term used for a complex or multi-part application that combines many of these technologies into one logical unit that is easier to deploy and monitor than its component parts.

About BPM

At the time of writing, BPM was a separate tool from the JDeveloper IDE, which is the focus of this book (although it will be merged into JDeveloper in the future). Rather than being focused at developers, it is more of a tool and notation for business analysts, providing them with a very visual approach to defining processes and workflows. As a higher-level abstraction of a process, it allows the analyst to both define the flow of a process and, to a certain extent, generate user interfaces that it will require. It also provides the infrastructure parts to actually execute those processed.

BPM generates out a mix of artifacts, including a process flow in the form of Business Process Modeling Notation (BPMN). In some ways, BPMN is a competing notation to BPEL; however, Oracle Fusion Middleware uses the same core engine to support both notations at runtime.

**TIP**

More complete coverage of the mechanics and usage of SOA technologies in Oracle Fusion Middleware is available in the sister volume for this book mentioned in the Introduction: Oracle SOA Suite 11g Handbook (McGraw-Hill Professional/Oracle Press, 2010).

How to Get It Wrong with SOA

As you have seen, many technologies sit under the SOA umbrella, not just web services. Taken as a whole, the platform is very attractive; services and processes can be mixed and matched at will without having to be explicitly designed to work with each other. Likewise, integration with external systems becomes much easier if both sides utilize SOA principles. However, those that adopt SOA have a tendency to make the following two fundamental mistakes, particularly when using SOA in database-backed systems.

Everything Is a Service? Architects unseasoned in SOA have a tendency to take the SOA message too literally and assume that absolutely every data access or process should be defined using SOA architecture and protocols. With a typical database application, such as an OLTP system, this is almost always the wrong thing to do. Relational databases already have a universal standardized and abstract protocol for interacting with the database—SQL. In a typical data entry form there is nothing to be gained by hiding data access behind a web service interface; doing so will reduce performance because of the additional cost of marshaling and un-marshaling data between Java and XML at either end of the service interface in addition to the extra overhead in shipping messages through different layers. Hiding data access behind a web service will also hide useful capabilities of the SQL interface from the client, for example, features such as array fetching, transactions, and key relationships. When data is needed for remote access as part of a genuine process orchestration, then it is reasonable to expose just that required data through web service protocols; however, the remaining 98 percent of your data access should not pay the penalty just to satisfy this corner case. The mantra for SOA designers needs to be “everything can be a service” as opposed to “everything is a service.”

Service Granularity One of the hardest jobs for a service architect is that of defining the correct level of granularity for a service. Many beginners who fall at the “everything must be a service” hurdle will often also end up defining those services at too fine a level. As an example, consider an HR-related function—promote an employee. This function represents a single, expressible use case with a basic set of inputs, for example, employee ID, new job grade, and new salary. However, encapsulated within that larger process may be multiple steps, including approving, updating the employee record with the new grade information, and integrating with the payroll system. So this is a great example of a use-case-driven service that is hiding complexity from the consumer and can react to changes in the business process if required. An overly granular approach to building in the same functionality might define separate services for updating the EMPLOYEES table and notifying the payroll system. Architecting the application in this way would force the developer to carry out the service orchestration in the calling program, thus hard-coding parts of the high-level use case into the application rather than encapsulating it behind the service façade.

The rules of thumb for service granularity are to ask the questions: “Does this service make any sense when used on its own?” “Could this whole service be relocated?” and “Is this service reusable in a variety of situations?” If there are dependencies and other steps in a process that must be invoked for this service to be valid, it may be defined at too low a level.

There may well be functions that operate at a finer level of functionality as part of a larger process orchestration, but these would be for internal consumption only.

Is Thick Database the Way Forward?

The so-called thick database architecture is occasionally held up as an alternative to SOA and Fusion development for the Oracle community. The idea behind *thick database architecture* is that everything from validation to workflow to complex business processes and even user interface should be defined in the database using a combination of PL/SQL code, views, and metadata. Thick database architecture is defined as having the following advantages:

- **Code is colocated with data** Code that works with data is colocated with that data and there is no network overhead used to transport data and SQL statements.
- **Business rules can be highly dynamic** Every system encodes a series of rules or validations for the data and processing. Some of these will be simple, hard-coded validations, but many are more complex, contain variable data, or change over time. If the metadata that configures those rules is held in the database, then the system becomes very agile, as that metadata can be easily changed on the running system.
- **UI independence is achieved** Using thick database architecture, you can simplify your user interface development and become independent of the technology that you are using. User interface technologies change more rapidly than core database technologies, and this architecture allows you to switch between technologies much more cheaply. You can also bridge between older UI technologies such as Oracle Forms and newer technologies such as JSF using this common layer.
- **Familiarity with PL/SQL is rewarded** Coding is done in a language that Oracle developers understand—PL/SQL.

On the surface, thick database architecture does seem like an attractive proposition for incumbent PL/SQL developers, as it provides a coexistence model that does not require them to venture too far out of their comfort zone. However, as with SOA, taken to extremes, the thick database architecture can be misused in each of those categories and does not necessarily offer more than the SOA and Fusion technology stack can:

- **Code is colocated with data** This is certainly a very good thing when that particular piece of code has to crunch through thousands of records to perform some calculation. For any such requirement, the database is certainly the place to put the code. However, most data access for an OLTP system is not of this nature, and in most modern systems it is not going to be acceptable for the user to have to post a transaction to the database just to perform a simple validation check. Among other problems this also introduces a long running stateful transaction into the database context which can seriously affect the scalability of the system. Users also want much richer UIs than the block-mode solution of a thick database can provide. Therefore, regardless of your use of thick database principles, you will inevitably end up duplicating code in the middle tier and the database tier. (This may not be a bad thing in any case.)

It is also a fallacy to assume that a core validation such as checking for a value in a list is always going to be more efficient when conducted in the database. When performing this kind of work in the middle tier, the frameworks will be caching data for later reuse and the caches can be explicitly shared among the entire user population. (This is something

we'll look at in Chapter 7.) As with all received wisdom, there is no substitute for prototyping and proving to yourself one way or another that the performance you receive using your preferred method falls within acceptable parameters.

- **Business rules can be highly dynamic** This is indeed a key requirement of modern systems, and although you can take the decision to build your own metadata-based infrastructure in the database to provide such flexibility, you can also just use off-the-shelf capabilities. Features used by Fusion applications such as Business Logic Units (covered in Chapter 6) and the Rules Engine provide much of the desired flexibility. Also note that the SOA-based architecture is designed to make business processes as a whole more dynamic, not just the micro-rules within them. SOA-based systems also can have the advantage that processes can be easily versioned at runtime. This increases the safety factor as new processes are brought on-stream, as only new transactions will pick up the changes and existing transactions can continue under the old rules. Although runtime versioning is possible with database-based business rules systems, it requires a less standard, homegrown solution.
- **UI independence is achieved** Fusion Architecture is based on a Model-View-Controller design pattern (discussed in Chapter 4) where the coupling between the business service (Model) layer and the UI (View) layer is already a loose one. The thick database architecture proposes a second layer of abstraction through database views and PL/SQL packages, allowing the reuse of a common table API between legacy and Java applications. In this case, the abstraction pattern within the database can be a very useful tool even when exposing that data through a Fusion middle tier. Since the Fusion Architecture already factors UI code away from business services code, the difference between the way Fusion Architecture and the thick database approach provide separate UI and business services layers is minimal.
- **Familiarity with PL/SQL is rewarded** You cannot argue against PL/SQL; it is a great language and is actively evolving. However, to a large extent, learning a new language is not a big barrier to overcome. Any programmer should be familiar with the basics of writing procedural code; for example, the exact semantics of writing a loop are of secondary importance. This is particularly true when you are using an advanced code editor such as JDeveloper, which will almost write the code for you if you take advantage of its templating and auto-completion features. In addition, as we describe in Chapter 4 in the section “Level of Knowledge”, developers do not need a high level of mastery with a new language to be immediately productive.

Let's preview some of the key features of the Fusion Architecture that provide support for putting more of an application infrastructure in the middle tier.

- **Declarative Development** One of the prime objectives of Fusion Architecture for the development team that created the underlying frameworks and tools was to enable application creation with as little coding as possible. Less code to write means less code to go wrong and less code to maintain. So most of a Fusion application, like an iceberg, exists below the surface in the form of declarative metadata. If you just concentrate on validation and basic business logic as areas of overlap with a thick database approach, the contrast becomes evident. From a declarative perspective, the database can implement simple check constraints and validate the relationships between tables, but pretty much everything else requires custom code.

Using the ADF Business Components framework within JDeveloper, the declarative palette is much richer than before, in terms of both the types of declarative check style validations and of advanced features like Business Logic Groups, which allow multiple sets of business rules to apply conditionally to a row, depending on some deciding factor such as the value of a particular column. The semantics of the declarative coding within ADF Business Components are also richer, allowing the developer to traverse relationships as part of validations and so forth.

- **Agility** In the subset of applications that have significant integration with other services, the loose coupling within a SOA composite provides a great deal of flexibility for making changes to the processes quickly and safely; for example, as mentioned earlier, multiple versions of a process can be in flight at any one time.
- **Customization** This is a key part of the Fusion Architecture feature set. Runtime metadata such as business logic, process flow and screen UI designs is managed through a service called Metadata Services (MDS). *MDS* provides an engine for running customizations on top of a core application that can provide multiple versions of the application, implementing different business logic, processes or UI. These versions may be differentiated by job function, department, or company. We will discuss MDS in more detail later in the chapter.
- **Caching** As mentioned earlier, the middleware engine and the frameworks that run on it provide caching by default. The Fusion Architecture frameworks allow developers to define the caches rather than caching being a DBA tuning exercise.
- **Security** Although the database effectively manages its own security, the scope is constrained to the database. When dealing with a modern SOA-based application, security needs to extend beyond the boundaries of the local process; for example, although a user executing a business function is locally authenticated and authorized, what happens if that process has to call out to some remote service to perform some work? The SOA infrastructure already has the protocols in place to manage the trust and service-level agreements between systems.
- **Scalability** This is a feature part of n-tier systems; connections are managed and the database is only used when it is needed. For example, a few hundred concurrent database connections may be required to support a concurrent user population of tens of thousands. This contrasts with a model that involves storing user state within the database session, which implies either a one-to-one client to database connection relationship (as is true in Oracle Forms), or at the very least, programming work by the PL/SQL programmer to manage the persistence and reconstitution of state from the session.

Of course, the success of scalability depends on how you write the application. It is perfectly possible to write a bad application that uses middleware and consumes more resources than the equivalent two-tier application. Fortunately, JDeveloper helps the developer in this area; for example, it provides auditing to warn the developer when something that they have done will prevent the application from migrating or failing over between nodes on a middleware cluster.

In summary, when it comes to how much work and logic you put into the database tier, it is a matter of being pragmatic. As you have seen, Fusion Middleware provides many capabilities out of the box that would be a lot of work to implement manually in PL/SQL database code. On the other hand, the database is highly capable and is there to be utilized. Also remember that the database should be a gatekeeper of the data that is stored within it, so relational integrity and basic validation such as check constraints should always be implemented no matter where the bulk of the business logic ends up, even if this seems like an apparent duplication of effort.

The Fusion Architecture

We've established that taking either SOA or the thick database architecture to the extreme is probably the wrong idea. The fact remains that in most applications, the core task is to move data in and out of the database. This use case remains unchanged from the traditional model that Figure 1-1 illustrates. Fusion Architecture as shown in Figure 1-2 reflects this reality; it is really an evolution of the traditional model that does not ignore direct database access for OLTP but adds in extra dimensions.

The top half of Figure 1-2 is similar to the mode used in the tradition model, although as you will see in later chapters, there has been a significant shift in the capabilities of the user interface and in the way that information is presented. Where things change is the area below the dashed line where parts of the Fusion Middleware suite are used. In this next section, we'll examine how these new components are used.

Assuming that the basic business problems have not changed, why is there suddenly a need to involve much more technology? One of the key factors is that, although the basic functions of our systems have not changed, the technology allows them to be implemented in a much better way. A traditional OLTP system includes an inline portion, the basic data entry and reporting functions; it also almost always includes more code in the form of batch jobs and integration code. In the traditional model of Oracle development you would often see logical transactions, which are invoked by some action in the UI, but not completed until some overnight job has run. Once you break down the functionality of these batch systems, you will find that their existence is largely explained by integration needs, whether it is in the form of generating some data file to transfer to a partner—for example payroll information—or to generate emails in some kind of workflow.

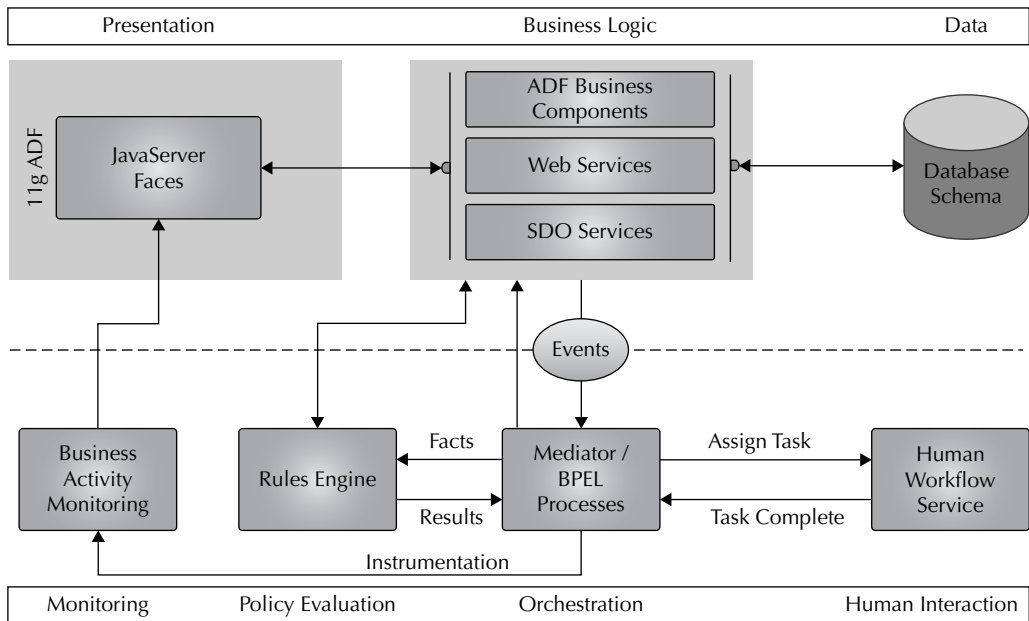


FIGURE 1-2. Fusion architecture as an evolution of the traditional model

The change is that the underlying infrastructure now provides all of the standards-based engines to manage these conversations with external systems in-line, rather than as batch, and there is no need for the build-your-own infrastructure approach of the past.

Another use case for batch processing, apart from integration and human workflow, is complexity. Often you would be unable to run a function or calculation inline because it would take too long. Thankfully SOA principles can also help out here as well—you can invoke asynchronous processes as easily as synchronous ones. Of significance here is the fact that BPEL has the concept of compensation.

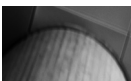
Compensation

Compensation is the process of recovering from errors, similar in concept to a rollback for the database, but potentially much more complex. Imagine a simple process for increasing an employee's salary. From the database perspective this is a simple update to a column in a row in the table. From perspective of the larger process, it would include a series of approvals and communication between the HR system and the payroll system (which might have been outsourced). So, if there is some downstream failure such as the departmental budget being exceeded, the transaction as a whole will need to be undone and the implications of this will extend beyond the core database that the UI is working with. "Compensation" is the name given to this recovery process, and it allows the developer to design the remediation process in just as much detail as the original process. An implication for the designers of SOA services is that they must think not only about the service design for carrying out a particular action, but also about the corresponding interfaces for use by compensating transactions.

Breaking Down the Architecture

Let's look at each of the components in Figure 1-2 and examine what they are used for, starting with the top part of the figure, which defines the core OLTP part of the architecture.

Presentation (User Interface) We spend a lot of time in this book focusing on exactly how you build user interfaces for this platform. However, the key feature of the view or user interface is that the user experience is very rich (highly interactive) even though these applications are delivered through a normal web browser. The core UI is delivered by a technology called JavaServer Faces (JSF), which is a core part of the Java EE standard. Chapter 9 discusses the basic principles of JSF. JSF provides a component-based UI development model and is only as good as the components that you have available. Fusion Architecture uses a set of components provided by Oracle called ADF Faces Rich Client (described further in Chapter 12). These components provide an end-user experience that approaches a desktop-like level of interactivity. Significantly, the range of components is huge, including not only the core data entry widgets that you would expect, but also a vast number of charts and gauges, and more exotic visualizations such as maps, hierarchy viewers, a pivot table, a Calendar, and even a full Gantt chart component.

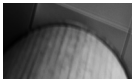


NOTE

Throughout this book references to ADF Faces refer to ADF Faces Rich Client 11g rather than the ADF Faces components shipped as part of JDeveloper 10.1.3. Any mention of the older component set will explicitly mention ADF Faces 10g. This older component set is still available in ADF 11g but is now packaged as Apache Trinidad. Trinidad is an Open Source project created from the donation of ADF Faces 10g to the Apache Foundation.

If there is one feature that you will immediately notice from the applications that Oracle produces using this technology, it is the way that these components are used to provide a visually attractive and engaging user interface. Another aspect of the user interfaces is the inclusion of features that would seem more at home on a social networking site such as Facebook. You can think of these social features as adding a dimensional context to data within an application. For example, you look at a record, and you can see who last changed it. This is not new, but you might now also be able to see that that person is currently online; in addition, you might actually be able to establish an instant messenger session with them, or click to call them over Voice over IP. Again, these are part of the middleware capabilities, which can add significant value to the users of your application.

A significant part of this social instrumentation requires infrastructure to back it up in terms of services and storage. Oracle's WebCenter product adds this value on top of the ADF core UI capabilities.



TIP

The Oracle WebCenter product is described in depth in another Oracle Press book (mentioned in the Introduction): Oracle WebCenter 11g Handbook: Build Rich, Customizable Enterprise 2.0 Applications (McGraw-Hill Professional/Oracle Press, 2010).

Business Logic We have already discussed the placement of business logic to a certain degree, and we describe the actual technology in some detail in later chapters. However, at this stage it is important to grasp the following key Fusion Architecture concepts:

- All data access to the business service layer from the user interface flows through an intermediate data binding abstraction layer. As you will see, interacting with this binding layer is one of the key skills of a Fusion developer.
- Most database access is still directly to the database rather than via some proxy such as a web service and is defined declaratively or in SQL.
- The core business service provider (ADF Business Components) is used to service SOA processes as well as user interfaces.
- A strong direct linkage exists between ADF Business Components and the Mediator for invoking SOA processes as part of a normal business transaction. This linkage utilizes a capability called *Service Data Objects (SDO)*, which is used to efficiently pass state between the business service and any associated BPEL process. The why and the how of this capability are beyond the scope of this book but are discussed in the online documentation and the *Oracle SOA Suite 11g Handbook*.

Data Let us not forget the database. At the core of all of these applications is still the requirement for solid relational design.

SOA Tier The bottom half of Figure 1-2 is getting into newer territory, working right to left:

- **Human Interaction** The reality of many systems is that they have the concept of approvals at some stage. In simple applications, approvals could just be part of the functional user interfaces, provided that all of the approvers in the system actually have access to and use the user interfaces. In more complex cases, something a bit more robust is needed.

Capabilities such as approval by email, approval chains, approval delegation and escalation, and so forth are all baked into the Human Workflow Service. Additionally, the work lists that are used within the service are all built using the same technology as the user interfaces you will be building for the rest of the application, so customization and integration of those interfaces is no great challenge.

- **Orchestration** The orchestration space largely belongs to two technologies: the Mediator (or its bigger brother the Enterprise Service Bus) and the Business Process Execution Language Process Manager (BPEL PM or just BPEL for short). The *Mediator*, as its name suggests, is basically a routing engine to take events and distribute them to the relevant orchestration engine, which is usually BPEL. In the process, the Mediator may transform or map the payload of the event, although this will be less common within an application, where you own all of the interfaces and can dictate the required data structures. The key task of the Mediator then is to abstract away the details of the location and nature of the process that will be processing the event.
BPEL PM itself is the main SOA engine. It is driven by the BPEL language, an XML syntax that provides a way of describing the process flows that are required. In the context of the JDeveloper IDE, this flow language is written through the medium of a BPEL diagram. This diagram allows the developer to draw the flow that will integrate the various services, which need to be coordinated as part of a transaction.
- **Policy Evaluation** Much like Human Interaction, many applications have the need for this capability and in the past have largely implemented it as custom code. The major focus in this space is the Rules Engine, although you can also consider policy evaluation in terms of governance of the running application—key performance indicators, and so forth. (This then overlaps with Monitoring.) Sticking to the Rules Engine, we are concentrating on the types of rules defined within an expert system, rather than the fixed business logic that would be defined with a simple validation on the business entity. A good example would be an insurance application. When an application for insurance coverage is made, you will need to consider many logical factors to calculate the level of risk associated with the request. What's more, those rules may change between deployments (if this application is being re-sold) or even depend on factors such as time. The Rules Engine is driven by rule sets that can be defined dynamically by expert business users, rather than developers, and can be easily customized after deployment when the application is live. Many Oracle customer systems today have hand-created exactly this kind of functionality in custom PL/SQL code and fact tables. Now the capability is available, out of the box, in Oracle Fusion Middleware.
- **Monitoring** Knowing exactly what is going on is one of the bigger problems of managing composite SOA applications. Multiple services may be consumed by an application, so visibility into how those services are performing is needed to measure the health of the application as a whole. If you cast your mind back to the traditional approach with batch engines, generally the batch job succeeds or it fails. If it fails, you find out in the morning, try and fix the issue, and just re-run the job. A batch job is asynchronous anyway, so that is a workable solution. In the more modern architecture things are much more fluid and it is much more important to have a real-time view of how the various moving parts are performing, in terms of being alerted both when things are not working at all, for example when losing connectivity to a partner's service, and when things are simply slowing down at some link in the chain.

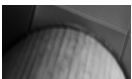
Oracle Fusion Middleware provides an engine for managing all of this information in the Business Activity Monitoring (BAM) service. *BAM* allows applications to be instrumented to output key performance data and correlates that information for presentation in the relevant management dashboards. Detailed discussion of BAM is out of scope for this book, but it is important to the Fusion Architecture overview to know that the data the BAM server makes available is just viewed as another service by the ADF framework that Fusion uses. Therefore, with an application instrumented with BAM the developer can easily expose performance metrics within the application. Information that was previously only available to administrators through specialized dashboards or through output logs can now be exposed within the core application user interface. In addition, BAM provides a data push capability referred to as *active data*. When a Fusion application user interface is bound to an active data service, its display will update dynamically as the underlying data changes without the user having to refresh the screen. This allows for the creation of real-time dashboards.

Now that we've discussed the high-level view of what moving parts may go into a Fusion application, it is time to drill down into an overview of the framework that makes much of this possible, the Oracle Application Development Framework (ADF).

What Is a Framework?

ADF is the development framework used by Fusion applications that we have already alluded to. Let us start out, however, by examining exactly what we mean by a framework before moving on to the technical pieces of ADF specifically.

When confronted with a set of low-level APIs, such as the Java EE APIs, programmers naturally start to develop common patterns and convenience methods for fulfilling requirements that come up again and again. After a while, most developers will build up their own personal toolkit of handy code and techniques that they can reuse whenever they encounter a familiar use case. Sometimes, problems are so common across the industry that formal recipes or design patterns are recognized for handling with the scenario.



TIP

There are many sources for more information about design patterns.

The seminal treatment is the classic Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley Professional, 1994), also referred to as the "Gang of Four book" after the four authors. This treatment of the subject is not lightweight, so we'd also recommend Head First Design Patterns (O'Reilly Media, Inc., 2004) as a more approachable treatment of the subject that focuses on Java in its examples and Design Patterns in Java (Addison-Wesley Professional, 2006).

Frameworks evolve as concrete implementations of such patterns, factoring out the repeated portion of the task, and leaving programmers with a more limited exercise of configuring the parameters of the framework to their needs and much less coding overall although coding is never completely eliminated. The scope of a framework can vary. Some frameworks address specific parts of the application development process. For example, *object-relational mapping* (O/R mapping), used to convert between object-oriented and relational structures (such as class instances and

database tables), is an important but constrained task. Many O/R mapping frameworks have evolved and compete; some, such as Hibernate, are from the open-source community; others, such as ADF Business Components or Oracle TopLink (EclipseLink), are from the commercial world. As you will see, using one of these existing frameworks for O/R mapping is a sensible thing; the skill comes in choosing a framework that will be best for implementing a particular project profile.

Frameworks are not confined to point solutions (addressing one and only one task), such as O/R mapping. Some have evolved to supply a much larger range of the functionality needed to provide the infrastructure for a complete application. For instance, the Apache Struts framework handles user interface creation, page flow control, and a certain amount of security. Inevitably, this has led to the development of larger frameworks, such as the Oracle Application Development Framework (ADF), JBoss Seam, and Spring, which directly provide infrastructure for some tasks as well as inheriting functionally by aggregating and integrating smaller frameworks, such as the O/R mapping solutions or JavaServer Faces. These conglomerates are referred to as meta-frameworks, described further in the sidebar “Meta-Frameworks.”

The Anatomy of a Framework

We explained how frameworks evolve out of standardized solutions to common tasks, in the process implementing the best practice for solving that particular problem. However, what transforms a programmer’s toolkit of simple APIs into a framework? Part of this, of course, is determined by the scope of the problem; beyond that, the following attributes imply framework status:

- **Configured, not coded** By definition, a framework performs most boilerplate and plumbing tasks for you. In order for that to happen, the framework must offer a way to define configuration data that provides the information it needs. This configuration data, or *metadata*, is sometimes injected through code or code annotations, but more often through some sort of configuration file. XML is typically used for such metadata files in modern frameworks, although this does not have to be the case.

Meta-Frameworks

We’ve just started to define a framework, and already we’ve introduced a twist in the form of meta-frameworks. What’s this all about? A *meta-framework* is an end-to-end application development framework that encompasses a wide range of functionality. The meta-framework not only provides functionality, but it may also encapsulate or subsume multiple single-solution frameworks. If a single-solution framework is a screwdriver, a meta-framework is the whole toolbox.

Furthermore, the definition of meta-frameworks includes the idea that they offer choice and pluggability for particular tasks. For the O/R mapping example, meta-frameworks, such as Oracle ADF or Spring, allow the developer to choose one out of a whole range of O/R mapping solutions to implement the data access function. The important distinction is that the actual choice of implementation will not have an effect on the rest of the application. The user interface, for instance, will be unaware of the actual O/R mapping mechanism being used. This allows for much more flexibility in the development of the application as well as ease in changing the underlying technologies later should the need arise.

- **Runtime component** Applications written using frameworks rely on the framework infrastructure code being available at runtime. This infrastructure code may be deployed with the application as libraries, or it may exist as some kind of runtime engine that the application runs on.
- **Design-time component** Since frameworks need to be configured with your business domain's profile, they need to have some way of helping you create that configuration. This facility may take the form of complete IDE support, including graphical editors and syntax checkers; or it may be a more manual process, for instance, a *Document Type Definition (DTD)* containing XML element and attribute names used to verify that an XML configuration file is valid.

If you review the many frameworks available within the Java EE universe, you'll see a vast spectrum of pretenders to the framework title. Many open-source frameworks, for instance, are weak on the design-time support or concentrate too much on coded configuration rather than on metadata-driven configuration. (See the sidebar "The Importance of Metadata" for details.) So let's look at what turns "just another framework" into something that you might actually want to use.

The Importance of Metadata

Although you can usually configure frameworks using both code and metadata, metadata has some advantages. First, you achieve a clean separation of framework configuration from application logic; second, you have the potential for customization without recompiling.

The first point is perhaps obvious. It is easy to understand the sense in keeping code that configures the basic framework operation separate from the code that implements the business logic so that updates are easier in case either layer changes. The second point, however, has implications that are more profound. The ability to customize an application through metadata can make the initial installation and setup of an application easier. For example, you can develop code while pointing the application database connection to a development database. Then, when you need to install the application in a production situation, you change a configuration file that points to the production database. The change is small and contained, and requires minimal testing.

Further, the metadata may be manipulated at runtime to customize an application. For example, your code can change metadata for security rules at runtime if the application has to adapt based on the credentials of the connected user.

The management and storage of such complex metadata is a problem. To this end, Oracle ADF has a complete set of services for managing and storing metadata called *Metadata Services (MDS)*. The presence of MDS is largely transparent to a developer using ADF, but its services underlie all of the framework metadata handling in the JDeveloper IDE and the WebCenter runtime environment. MDS is of particular importance to organizations building applications for resale, although it can also have a place in applications used by multiple user communities within an organization. It allows a core application to be developed and then customized in a nondestructive way by its consumers. Such customizations are recorded as differences to the core application and do not physically change the base source code. This process simplifies the entire upgrade and patch process for the product using MDS technology.

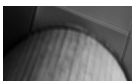
What Characterizes a Good Framework?

Although we've defined a framework as a best-practice implementation that includes runtime, design time, and a little configuration, there is much more to a framework than that. Let's consider the factors that make a framework truly useful:

- **Functional depth** The framework must provide all features needed for a particular functional area. This is the one area when a well-thought-out commercial or open source framework can outshine an in-house toolkit. The framework must be able to handle every eventuality in the problem domain. Take, for example, an O/R mapping framework that handles basic query syntax but cannot map outer joins. Such a limitation may not have mattered to the original developer, but if you suddenly need that capability in your application, what do you do? You certainly don't want to employ a second framework just to fulfill that niche requirement.
- **Functional scope** The functional depth argument works the other way, too. A framework can be too specialized to a particular domain. An O/R mapping framework that specializes in only the query of data and that does not manage updates is of limited use in most applications, no matter how deep its query functionality. Learning to use a framework well takes time; you don't want to have to repeat that exercise for every piece of point functionality in a system.
- **Being as declarative as possible** You would expect that a framework will relieve you of the burden of coding, and this implies that most of the interaction with the framework will be in configuring it with the relevant metadata, rather than writing lower-level code.
- **Clean APIs** Frameworks must provide a clean separation between the underlying code and the programming interface that the framework exposes to the developer. The framework's authors should be able to change its implementation without affecting the code in existing business applications. This problem is particularly difficult in the Java EE world, as the frameworks and code written using them are generally both written in Java. In Java, it is fairly simple for a developer to inadvertently call an internal function by mistake or to override an internal-only class accidentally (or even deliberately to obtain some extra functionality). An update to the framework can break that type of code.
- **Extensible APIs** It is a fact of life that you can't please 100 percent of the people 100 percent of the time. Frameworks have a similar problem here. By necessity a framework has to be a compromise and has to dictate certain ways of carrying out a task; after all, it's doing most of the work for you. A good framework will, however, give the programmer the hooks to extend or add functionality at the ground level to the framework, to customize it in a nondestructive way to more closely meet the needs of a particular task. It is often through the provision of such mechanisms that frameworks evolve: as one particular extension becomes more and more popular, it becomes a candidate for rolling into the core framework itself.
- **Tooling** Many frameworks attempt to make your life easier by allowing you to configure them using metadata. However, if this involves having to manually write XML files, for example, you may well lose many of the productivity gains that the framework promises, because XML coding is prone to syntax errors that are difficult to debug. A framework does not need associated IDE support to make it successful, but such support will certainly boost adoption by flattening the learning curve. A good example of this is the Apache Struts framework. The Struts framework was popular before any kind of visual

Struts tool was available, but with the introduction of diagrammatic representations of Struts page flows in IDEs, such as JDeveloper and Eclipse, Struts use grew immensely. Developers using Struts in such graphical environments need never learn the syntax of the Struts XML configuration file or, indeed, the in-depth mechanics of how Struts works.

- **Scalability** There is a world of difference in writing an application to track membership of a local soccer club and building out an enterprise HR system. One of the key abstractions that a good framework makes is to prevent the programmer from having to design and code the application around the need for scalability. The framework should take care of this as part of the plumbing and make those optimizations for scalability, such as supporting clusters of servers, transparent to the application developer if at all possible.
- **Community and Acceptance** Frameworks evolve in a Darwinian environment; that is, widespread adoption is more a key to survival than technical brilliance. Particularly with open-source frameworks, the framework will survive and evolve only as long as people feel that it's worthwhile and are prepared to give their time to both using and maintaining it. Successful frameworks tend to have a snowball effect: they are discussed more on the Web; more books are written about them; and because more information is available, the framework becomes even more popular. An active user community is another natural result of wide acceptance. An active user community can offer support and experience in real solutions that are not available from the framework's author. Therefore, a user community adds to the snowball.
- **Support** The most expensive part of using a framework is never the purchase price (if there is one); it's the investment in time to learn and use the framework and the maintenance of the system you create with that framework down the line. With open-source frameworks, you have access to the source code, but in many cases, you may not have the expertise or desire to understand that code well enough to fix bugs yourself. As soon as you make a change, you need to either contribute that change back to the community (assuming that there still is a community at that point), with the possible legal issues that this may cause in many organizations, so that the change can become part of the framework. Alternatively, you can keep the change to yourself, but then you're stuck with a custom implementation of the framework, and if and when the framework evolves to the next level, your change may not be relevant or supported. The fashionable business models in the open-source world recognize this issue. Much open-source software is now directly funded by companies looking to profit from support or consulting services in those frameworks. This is the ultimate loss-leader. Of course, this provides a loose coupling in terms of support, because those vendors can just walk away if something better comes along. Frameworks from stable companies like Oracle Forms and ADF at least offer a degree of service level, with commitments to error correction and so on. This is what gives them such tremendous longevity (for example, more than 20 years and counting for the Oracle Forms framework) and makes them a surer bet for typically long-lived commercial applications.



TIP

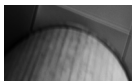
The source code for the ADF framework is available to supported Oracle customers on request. If you have an Oracle Support contract, contact Oracle Support Services to ask for a copy.

- Documentation** The learning curve for a framework is greatly reduced if it offers complete documentation in several categories: a reference for the APIs and framework classes, development guides with examples, and IDE documentation with task-oriented explanations. The more documentation the framework offers, the better the perception of its ease of use. This adds to its popularity as well as to its usefulness.

What Is Oracle ADF?

The *Oracle Application Development Framework* (Oracle ADF or just ADF) is a meta-framework that fulfills the core requirements for a framework as outlined in the preceding section. ADF integrates a mix of subframeworks to provide the key functions for object-relational mapping and other forms of service access, data bindings, and user interface, along with the functional glue to hold it all together. Nearly everything that a core OLTP application needs is already encapsulated within ADF. However, if something is not available, you can add external packages and libraries to extend the meta-framework or push the work into the SOA layer using the available hooks.

Figure 1-3 shows the core technologies available within the overall Model-View-Controller (MVC) design pattern used by ADF.



NOTE

Chapter 4 discusses the MVC pattern along with much of the core technology that is used by ADF.

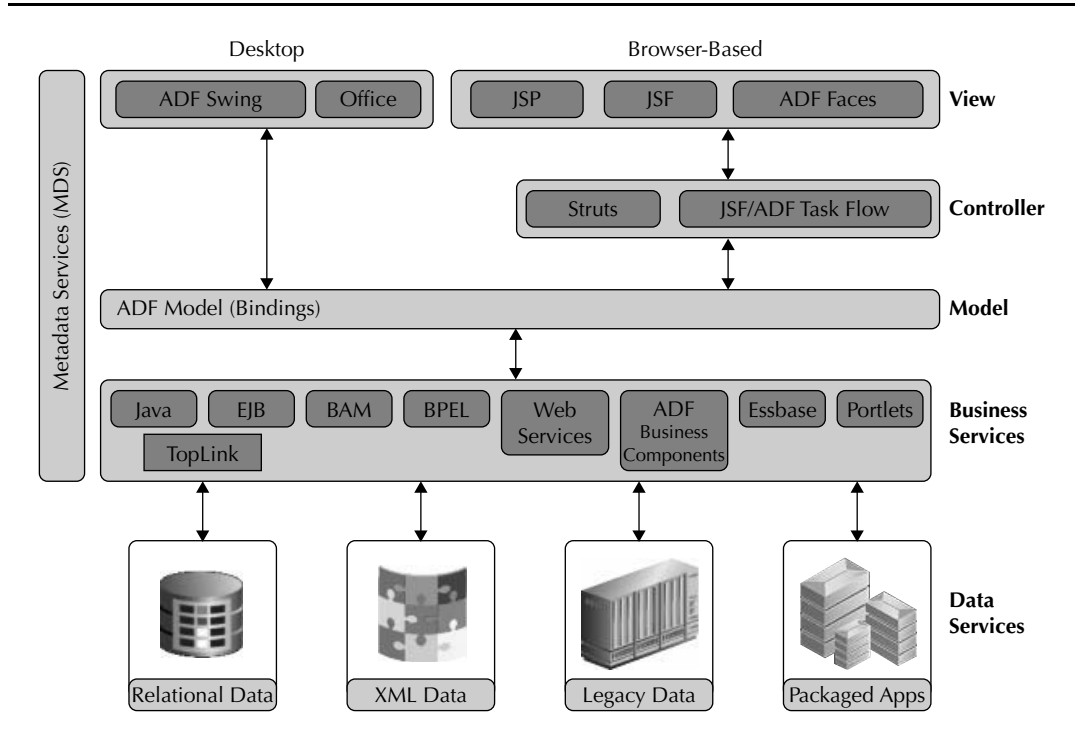


FIGURE 1-3. *The Oracle ADF technologies*

Since this book focuses on the key ADF technologies used by Fusion (ADF Faces, ADF Controller, and ADF Business Components), we will not discuss all of the possible technologies in the ADF architecture. However, Figure 1-3 highlights the pluggability of ADF as a meta-framework. ADF coordinates your selections from a range of technologies that fulfill various logical functions, such as interacting with the database and generating user interfaces. The key component is a layer provided by the ADF framework: ADF Model (or ADFm). The *ADF Model* layer (depicted as the Model section of Figure 1-3) acts as the glue between the various business service providers, such as Enterprise JavaBeans (EJBs) or web services, and the consumers of those services, generally the user interface that you are building. We discuss the ADF Model layer in detail in Chapter 13.

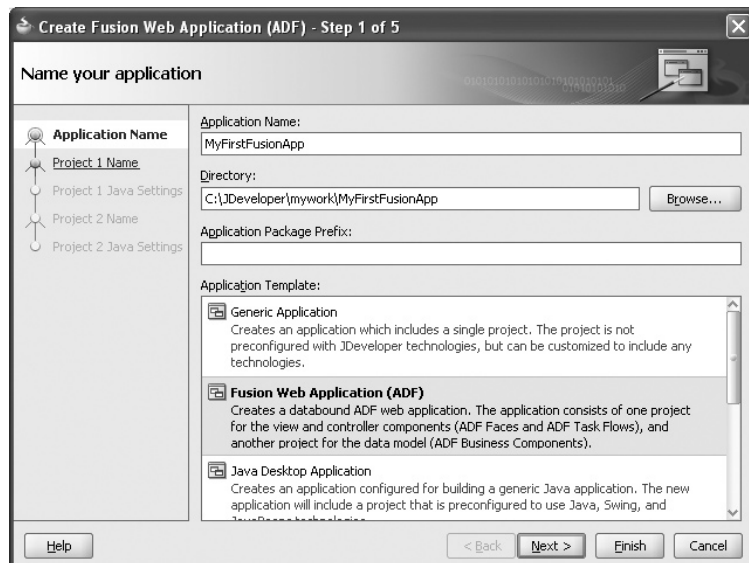
**NOTE**

The ADF Model binding layer is being considered in the Java Community Process (JCP) as an extension to the J2EE standard. The proposal is covered in the Java Specification Request (JSR) 227, which you can find using the JSR search feature at www.jcp.org.

How Do I Choose Technologies?

ADF is a great offering, but the capability to plug in several different solutions into each functional area gives rise to a problem—there is almost too much choice. If the technologies to be used for an application have not been predefined for you, what slice through the available technology stack should you use?

When starting out on your first Java EE project using ADF, you are going to come up against this technology question right away. The first step when creating an application presents the Create Application dialog, which asks you to select an application template, as shown next. Each template consists of a different combination of technologies.



If you make a wrong choice, you can always change your mind later on, but it's better to get it right from the start. Reading the descriptions will help you make the decision.

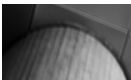
When building a traditional data entry–style application of the type you would build with Oracle Forms, PeopleTools, or the PL/SQL Web Toolkit, you need to make a technology choice for two main functional areas: user interface and database integration.

User Interface Technology

This book concentrates on building web applications for delivery through a standard web browser, without the use of plug-ins or Java applets. This means we concentrate on UI technologies that can generate standard HTML, and maybe some JavaScript to create the screens. A large number of technologies available in the Java EE world address this task, for example, JavaServer Pages (JSP), servlets, Velocity, Tapestry, Wicket, and JavaServer Faces (JSF). However, JDeveloper best supports JSP and JSF.

Before the advent of JSF, the user interface was a difficult area in which to make a decision; JSP technology has the benefits of being widely used, but it provides a rather low level of UI capabilities and is hugely fragmented in terms of the libraries and techniques to use even within that single technology stack.

JSF has changed this picture considerably, and because of the way it is architected, it brings some key capabilities to the table; we discuss some of these later in the chapter.



NOTE

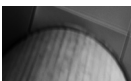
Chapter 9 describes JSF in more detail.

But What Is a Modern User Interface?

Over the last two or three years there has been a fundamental shift in user expectation when it comes to web-delivered user interfaces. Public applications such as Yahoo Mail and Google Maps have reset expectations of what a web user interface can do, and in the process the bar has been raised considerably for the average web developer.

Examples of the kinds of interactivity that users have come to expect are drag and drop, instant validation, screens, which can be manipulated with the mouse to resize or reorder, and so on.

The sad reality is that these modern, highly interactive web user interfaces are really hard to build without a framework to help. Developers need to combine skills in Java, or some other server-side language, JavaScript, and Cascading Style Sheets just to get started. Then they have to enter the mysterious world of Asynchronous JavaScript and XML (AJAX) programming, and finally they are confronted by the distinctly nonstandard ways in which different browsers behave in the area of building dynamic applications in particular. The sidebar “AJAX in a Nutshell” explains a bit more about AJAX.



NOTE

Don't worry about the details of these languages at the moment. Chapter 4 explains these terms in a bit more detail.

Why JSF?

JavaServer Faces has really changed the playing field for web user interface development. Published as a standard in March 2004, JSF is now, like JSP, a formal part of the new Java platform specification (Java EE 5), and Java enterprise container engines, such as Oracle's WebLogic, implement a JSF service in order to conform to this standard.

Ajax in a Nutshell

AJAX (or “Ajax,” described in Chapter 4) stands for Asynchronous JavaScript and XML, and describes a series of techniques for allowing components on the page to communicate with the server in the background. Fortunately, you don’t have to worry about the mechanics of all of this when using ADF; it’s all taken care of. So when you see the term “AJAX,” just think “rich user interface.”

JSF technology is attractive because of the following characteristics:

- **Component-based architecture** All UI elements within JSF are components and are written to a standard specification. These components can be complex and can consist of many sub-elements. For example, a tree control consists of images to connect the nodes, hyperlinks for the nodes, buttons to expand and collapse all, and a lot of dedicated JavaScript. Because components are standardized, you can pick and choose from multiple open-source or commercial sets of widgets and create your own, if required. The components will (in theory) all work together and be consistent to use.
- **Programming model** Each JSF component is able to register action handlers and listeners for events. Put simply, for something such as carrying out an action in reaction to a button press, the programmer just has to write a piece of what we’ll call “trigger” code that is associated with the button definition in the UI and automatically executed by JSF when that button is pressed. Contrast this to JSP or servlet development, where the programmer needs to decode the request object to determine if a button was pressed and, if so, which one was pressed. Therefore, JSF makes the whole process of wiring up code to UI events much simpler for the programmer.
- **Page layout** In the same way that each UI element in JSF is a component, screens collect and organize those components within specialized layout components. This removes the requirement from the developer to code low-level screen layout tasks using HTML tables and style sheets. It also requires much less code to define a single screen, since the boilerplate HTML elements used to generate a particular layout are created only at runtime. Some JSF component sets, such as *ADF Face Rich Client* and *Apache Trinidad*, also support the concept of multiple look and feel—custom skins, templates, and other definition files that enforce a common look and feel within an application. These capabilities allow the whole graphical aspect of an application to be changed after creation without having to change each and every page.
- **Component pool** The Fusion developer is fortunate in having around 150 different components available for use from within the ADF Faces Rich Client component set. This pool of components includes the usual elements for creating data entry forms. However, also included are components such as the following:
 - Charts and graphs
 - Gauges and dials for status display
 - Geographical maps
 - Pivot tables
 - Hierarchy viewer

In addition, ADF Faces offers some specialized nonvisual components to perform tasks such as file download, accessibility support, data export, dialog windows, and so on. Therefore, the JSF developer immediately has a much richer palette to work from, and what is most important, each component works with every other component in the same way. This blurs many of the old lines between specialized developer categories such as Graphical Information Systems specialists and Business Intelligence specialists. With a JSF application using an advanced component set such as ADF Faces, you can see all of these categories blended into one.

- **Device-independent rendering** Through the use of components and layout containers, the programmer is actually coding an abstracted definition of a page rather than hard-coding the actual HTML tags that the web browser displays. JSF includes a mechanism called *render kits*—code that can enable a single page definition to be rendered in a device-specific way at runtime. For example, render kits allow a JSF page to run unchanged in a normal browser or on a handheld device. The JSF component and its render kit, rather than the developer, is responsible for creating the correct markup tags for the target device. This point is key when developing really rich user interfaces. Components that need a lot of scripting support to create client-side interactivity can take care of rendering the correct scripting for the particular browser that the end user is using. This saves the developer a huge headache. (Also see the sidebar “JSF and AJAX—A Rocky Road.”)
- **Security** One of the secret problems of a modern AJAX (interactive) web application is that because there is a backchannel continually transmitting information between the browser and the server, the so-called attack surface for hackers is greatly increased. Because JSF enables the components to encapsulate the client-side scripting, they can be fully tested and hardened to remove any risk. If you hand-build the equivalent code for this kind of client-side interactivity, then that code becomes a security problem for you rather than one for Oracle.

Therefore, with the advent of JSF, the problem of choosing the correct UI technology has been greatly simplified. There is no doubt that with the framework, and the component sets provided by ADF, applications that have a similar level of interactivity to thick-client desktop applications are a possibility. What’s more, you can write them without the in-depth knowledge of browser nuances or JavaScript.

JSF and AJAX—A Rocky Road

Although the component-based model of JSF promises seamless interactivity from components from different places, the reality does not quite match the vision. The problem here is how those components manage AJAX transactions. The techniques now used for these very rich web user interfaces were popularized only after the core JSF component specification was laid down. The JSF component vendors such as Oracle all wanted to enable their components with this rich functionality. However, because the JSF standards do not define how this should be done, the industry now has several competing implementations, which are not interoperable. As a result you are unlikely to be able to mix components from different component libraries on the same page. Fortunately, the ADF Faces Rich Client and Data Visualization Tools (DVT) component sets provide a pretty wide palette to choose from.

This entire AJAX compatibility issue is currently being addressed as part of the JCP's JSF Expert Group's discussion for the next release of the JSF specification. So, with luck, this will cease to be an issue in the future.

Database Integration

In a database-centric language such as PL/SQL, you may take it for granted that you can just embed a SQL statement without thinking about how it will actually be issued or how the data will be handled. However, in the Java EE world, things are different. You cannot just embed SQL within the code; you need to use specific APIs to handle the SQL. Low-level Java Database Connectivity (JDBC) APIs (explained a bit more in Chapter 4) are available for accessing the database. However, by taking this approach, you are continually repeating code and introducing more places for bugs to creep in, not something you really want to be doing. Therefore, with Java, you need to use one of the higher-level O/R mapping frameworks supported by ADF.

Your Choices for Database Integration

ADF offers two primary choices for directly mapping database objects to Java code. Each choice exhibits a slightly different focus and suits different communities of developers. The choices follow:

- ADF Business Components
- Enterprise JavaBeans (EJBs) using the Java Persistence API (JPA)

ADF Business Components ADF Business Components (ADF BC) is a powerful and rich framework for mapping database objects into Java. It forms one of the core frameworks of the ADF stack of technologies (as shown in Figure 1-3). It is widely used in the Oracle community and is the key object-relational mapping tool for Fusion Applications. Consequently it's the technology that we concentrate on in the rest of this book (for reasons further discussed later, in the section "Selecting the O/R Mapping Tool"). It is well suited to what we call the *relational viewpoint*, where you approach the design process after creating a well-formed relational database design. This, after all, is the way that most database developments are run—with the database designer, DBA, and coders working closely together.

Enterprise JavaBeans (EJB) and JPA The Enterprise JavaBeans standard defines a server-side component architecture for Java Platform, Enterprise Edition (Java EE) loosely based (conceptually, at least) on the JavaBeans standard used for Java GUI components. The EJB standard defines a series of services for handling database persistence and transactions using an EJB container, which is usually, although not always, provided by the application server's Java EE container. EJBs have had such bad press in the past that they've become somewhat of a cliché. However, things have changed in the EJB world with the latest revision of the specification—EJB 3.0. The Java Community Process expert group driving this latest revision of the standard (JSR 220) received the message that EJB was unnecessarily complex, so EJB 3.0 is focused on simplification. EJB 3.0 also defines entity beans as *Plain Old Java Objects (POJOs)*—standard Java class files—without having to implement all of the interfaces and artifacts required by earlier standards. The EJB standard learned from its past mistakes and is turning into a usable way to handle data. As part of this revamp of EJB, the *Java Persistence API* was introduced as a core part of the implementation. Unlike traditional EJB, JPA is not exclusively tied to a full Java EE container and can be used to access databases from within Java SE as well as Java EE.

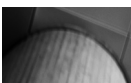
**NOTE**

For more information about the JPA in particular, you can refer to the FAQ at java.sun.com/javaee/overview/faq/persistence.jsp.

On the positive side, using EJB does have some advantages: all of the major development IDEs can help you build them, and every vendor's EJB container provides all of the EJB services required for persistence and transactional support at runtime.

However, to a traditional Oracle developer, EJBs are still going to be a less attractive option for the following reasons:

- **You do not write “normal” SQL** Instead, you write in a slightly different dialect using the EJB Query Language.
- **The EJB container generally handles persistence and querying** Embedded hints in the form of code annotations define the management of data and relationships in data, such as sequence-number generation and master-detail queries. None of this is a bad thing but traditional Oracle developers may find this a little hard to adapt to and feel that EJBs lack the degree of control that they require.
- **EJB is a bare-bones framework** You need to write your own code to add capabilities, such as validation in the EJB layer, although the Java standards are being enhanced as this book is being written to add a declarative data capability to JavaBeans generally. Note also that ADF actually provides a way to decorate bindings to EJBs with some basic metadata based validation as well. Neither of these techniques provides the same power or flexibility as the built-in validation capabilities of ADF BC.
- **It is not based on metadata** With an EJB application there is much more Java code to write, and every line of that code is one more line to debug and maintain, and one line less that can be customized at runtime.

**NOTE**

Oracle's implementation of EJB 3.0 within Fusion Middleware uses Oracle TopLink (an O/R mapping framework) under the covers.

This foundation opens the way for users of Oracle's particular implementation to use native SQL, and to gain all of the benefits in performance and resource usage that TopLink provides. TopLink also adds additional features above being an implementation of the JPA, providing, among other things, Object-to-XML mapping and database web services.

Selecting the O/R Mapping Tool

This book is biased to approaching Java Enterprise Edition development from a relational perspective, which is the normal perspective for existing users of the Oracle database. Accordingly, we don't hesitate to recommend that you use ADF Business Components. This recommendation is not intended to minimize the effectiveness of other options, but experience has shown that traditional Oracle developers, in particular, adapt to ADF Business Components fairly quickly.

We'll look at some of the reasons for that next. However, one of the benefits of using the ADF framework is that, should you choose to use any of the other options, such a decision will not really affect the user interface of the project; it will just affect the mechanics of the model or database integration layer and the amount of code that you have to write.

ADF Business Components is attractive for the following reasons:

- **It emphasizes declarative definition** It's possible to build a relatively sophisticated database integration layer without a single line of handwritten code. You can generate default mappings to database tables using a wizard. JDeveloper will *introspect* (automatically examine properties of) the database schema and not only generate the table mappings but also put in place all of the artifacts and rules required to enforce referential integrity. It is worth noting that declarative definition also provides a cleaner upgrade path for the future and opens up the possibility of customization using the metadata customization capabilities of the framework.
- **It provides the ability to define basic validation rules** It does so in a declarative fashion, just as with declarative O/R mapping. This type of declaration includes validating an attribute based on a database lookup—something that is relatively easy in PL/SQL and tools like Oracle Forms but that is generally not an off-the-shelf function in O/R mapping frameworks.
- **It exploits the power of the database** ADF Business Components will run against non-Oracle databases, but Oracle is its core competency; this focus allows it to support functionality such as *interMedia* (now known as Oracle Multimedia) types and bulk (array) operations.
- **It provides a rich event model** This allows developer much more control over the internal processing order and other features.

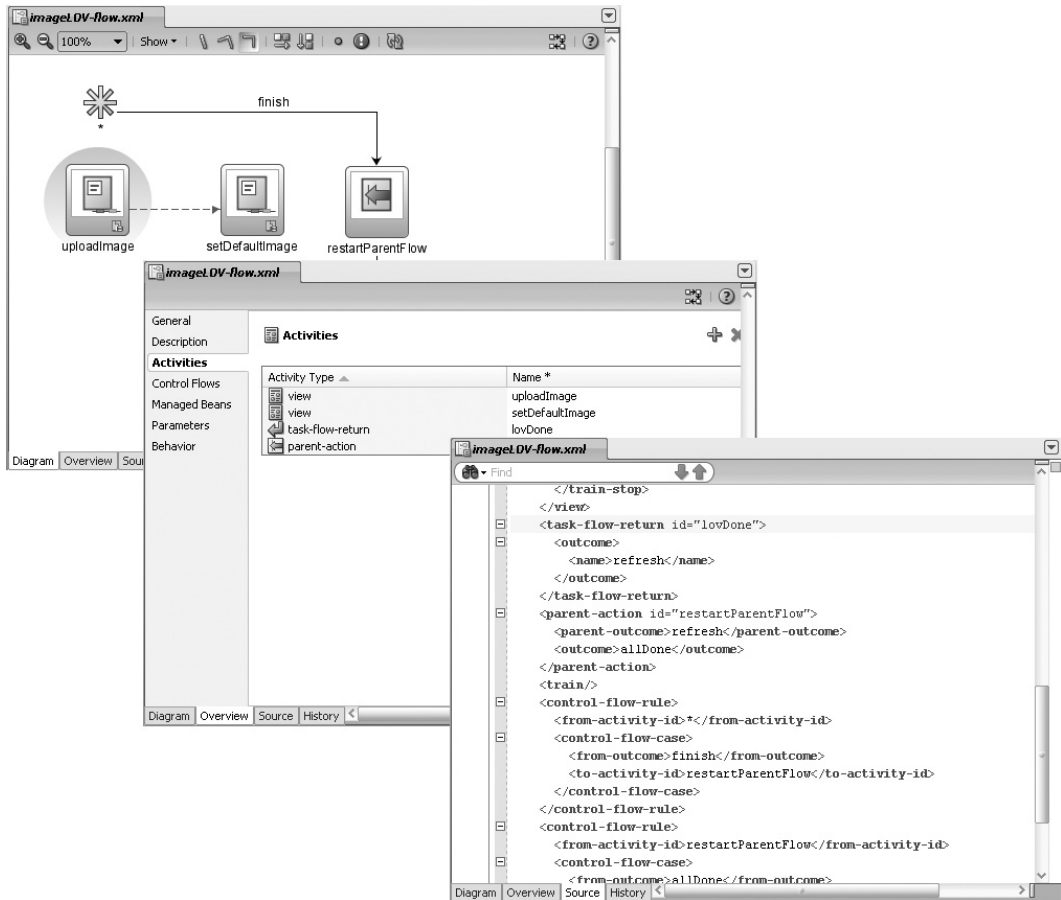
We'll spend a lot more time with ADF Business Components in detail in Part II of this book.

Why Should I Use JDeveloper?

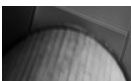
We've been making oblique references to the JDeveloper IDE throughout this chapter without really touching on what it is. JDeveloper is a free tool that Oracle supplies to help developers build applications for deployment onto both the middleware platform and the database. It provides a wide range of capabilities as we will show, but most important, it is the tool that is used to build the Fusion Applications within Oracle itself.

At its heart, JDeveloper is a development environment engineered to make the end user developer (you) as productive as possible. It manages the delicate balancing act of providing very visual abstractions for tasks such as page flow design or page layout, without preventing access to the underlying source. Thus, developers are able to work in a mode that suits them for the task on hand. This is really important as developers cycle through the learning process with a new API or framework. The visual editors help them to get up to speed quickly and produce working code, even when they are not entirely sure how it works and what it is doing. However, once the developer's use of the technology matures, that same visual editor does not have to get in the way. There is always the option to drop down into the underlying source code and make

the changes there—an approach which is often quicker. The following illustration shows the multiple views available to the developer for editing a page flow. We can see (from top to bottom) the diagram view, the structured overview view, and of course the XML source code.



Any of these views, in combination with the Structure Window and the Property Inspector, can be used to manipulate the object. JDeveloper takes care of keeping everything in sync. This approach of the IDE is also crucial when teams of developers are working together on the same artifacts. They can all use their favored views (or even a different IDE all together), and everything will be kept synchronized.



NOTE

Chapter 2 discusses the key structural features of the IDE such as the Structure window and the Property Inspector.

Of key importance is the range of technologies available to the developer. JDeveloper is a *platform IDE*; that is, it serves the development needs of the entire Oracle platform, from working with SQL and PL/SQL in the database, to creating Java EE web UIs, to SOA orchestration, to customizing Oracle's packaged applications. It is literally the Swiss Army Knife of IDEs and is best of its breed in many of those core tasks. One area of particular focus, of course, is the support for the ADF development framework. To use ADF in another IDE such as Eclipse would mean a lot of editing of raw XML files. In JDeveloper, the developer uses a drag-and-drop environment instead; all in all, it is a lot more attractive for building Fusion-style applications.

JDeveloper is also a very adaptable IDE that can be both customized and extended to meet your requirements. Some of the aspects of customization such as the basic environment, layout, keyboard shortcuts, code templates, and so forth can be altered from within the tool. Other customizations require the creation of plug-ins for the product. We don't have space to discuss that topic in this book, but it's a really interesting process to get into and really not as difficult as you might expect. Head over to OTN (www.oracle.com/technology) if you have an interest in that.

Of course, there is much, much more to talk about in relation to JDeveloper, so much so in fact that we devote the next two chapters to it.