



I D E A B L A D E

# DEVFORCE

## BUSINESS OBJECT SERVER

Version 1.5

The DevForce Business Object Server (BOS) enables a DevForce application to be deployed as a secure, scalable, multi-tier internet application.

You make an important business and technical decision when you choose to use – or not use – the BOS in your application. This paper explains what the BOS is, what it does, and when it matters. It poses and answers some of the most frequently asked questions.

## Summary

Most client/server applications communicate directly with their data sources.

The client program runs on a PC where it interacts with end users. When the program needs data, it establishes a networked connection to a database on a different machine and sends a query to the database; the database returns the requested data over the connection; the program translates the received data into business object form and continues working.

Following the recent fashion in Service Oriented Architectures (SOA), we are starting to see client/server applications that access remote web services. Clients establish their own direct connections, implement their own security (if any), and exchange data with those services autonomously – that is, without any scrutiny from the business.

Such arrangements may be satisfactory for simple enterprise applications supporting small numbers of users who are continuously connected within a secure, high-bandwidth, local area network (LAN) and interacting exclusively with internal databases and web services.

The client’s unmediated connection to a host server is its Achilles heel. There is no place outside the physical client tier to satisfy the security, scalability, reliability, and functionality requirements in more demanding scenarios.

Failure to address such requirements can increase business risk leading to financial losses. Ad hoc client-side workarounds are typically expensive, misleading, and ineffective.

The problem is fundamental. We should not burden the client with responsibilities that are better executed on the host. We shouldn’t be asking a database to perform functions unrelated to its mission of managing data. We need an intermediate layer of software – running on a host server – that provides core services including a secure gateway to ancillary services.

This is the role of a middle-tier layer such as the DevForce Business Object Server (BOS). The BOS is application server software that executes in the host environment. Client programs channel their requests for business objects and services through an instance of the BOS which acts as a mediator between those clients and the requested servers.

Here are some of the reasons why businesses choose to implement the BOS:

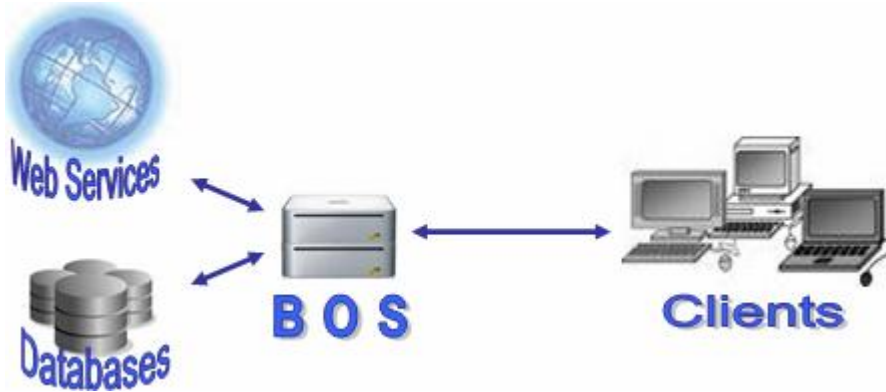
Need	Reason
Security	<p>We must ensure confidentiality of transmitted data, authenticate each communication, and impose application-specific authorization logic that is invulnerable to client-side hacking.</p> <p>We want a consolidated security solution for all requests whether directed at a database or a web service.</p>
Performance over WAN	<p>We need to support users who connect over a wide area network (WAN) or who connect to our LAN via VPN.</p>
Access Everywhere	<p>We have business partners outside the firewall who are not authorized to access our network; we cannot give them VPN access. Such clients should be able to acquire data and services over port 80 (HTTP) or 443 (HTTPS).</p>

Need	Reason
Limited Network Bandwidth	Some clients have low-bandwidth connections or lose their connection. We must limit the number of transmissions, compress the data transmitted, and be able to continue to operate when we lose the connection.
Central Monitoring	We need to track server and client application activity including data source operations such as queries and saves.
Web Services	<p>We need centralized control over client access to web services so we know what clients are doing and that they are authorized to do it.</p> <p>We want to publish a subset of our business object model as a web service to both .NET and non-.NET clients.</p>
Hosted Services	While much of the application runs best on the client, some business logic and services must run on the server either because the logic is proprietary or because the resources are scarce and cannot be distributed to clients.
Software-As-A-Service and Software Trial Options	<p>We could reach a larger market if we could “rent” our product to customers who cannot or do not want to host it themselves.</p> <p>We could increase sales and shorten our sales cycle if prospects could experience it hands-on, online.</p>
Server Push	Certain data can change at any moment and we want the server to tell our clients about those changes immediately.

The DevForce Business Object Server is the key software component in a multi-tier application that can address all of these needs.

## BOS Technology

The Business Object Server (BOS) is a middle-tier mediator between clients and networked resources such as databases and web services.



It is a software component that executes on a server within a hosting application. The host can be any application that can execute managed code on the .NET 2.0 Framework<sup>1</sup>.

The BOS' primary job is to provide clients with business objects. It can also serve as a gateway to other kinds of services and resources that, for security or performance reasons, should execute on a server rather than on a client.

For example, a client might initiate a Reporting Services request that sifted through a terabyte-sized data warehouse before emailing the finished PDF report. The client could transmit its request over its channel to the BOS; a receiving class running on the BOS could ascertain the client's right to make that request and then delegate it to another process (perhaps on a another tier) for processing.

Of course the application architect can by-pass the BOS and open its own channel to the service. But why duplicate what the BOS already does well? If each client logs-in to the BOS at the start of a session and then channels all of its requests for networked resources through the BOS, we can

- simplify authentication and authorization,
- reduce the "surface area" subject to attack,
- audit and monitor requests,
- consolidate data access,
- lower our development, test, and maintenance costs.

---

<sup>1</sup> The BOS ships with three hosting alternatives: an IIS application, a Windows Service, and a console application.

## BOS as a Data Access Layer (DAL)

Enterprise applications request data more often than any other networked resource. The BOS' primary role is to mediate between permanent data stores and clients.

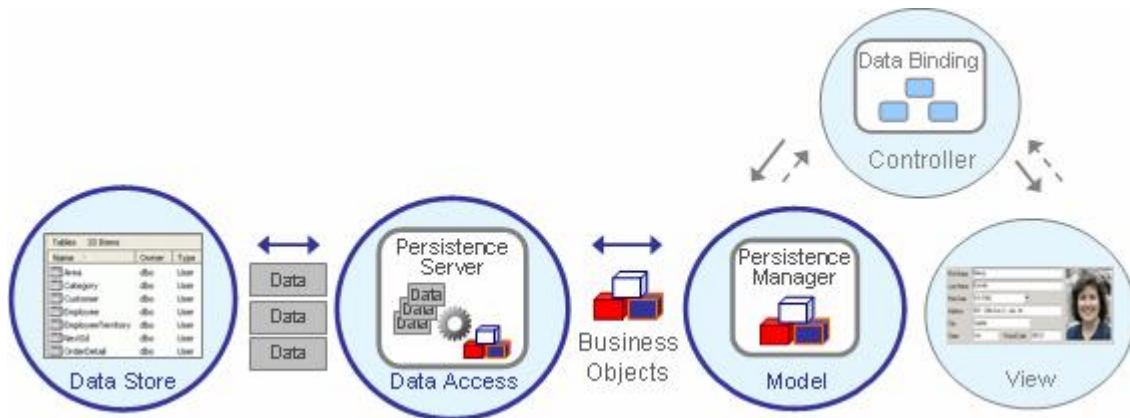
A data store is typically a SQL relational database but it doesn't have to be. The data could reside in a flat file or within some other legacy data manager. The variety can be daunting. Fortunately, most shops have developed web service adapters that provide bidirectional access to these data stores. The DevForce object mapping and persistence technologies can work with both relational and web service access methods.

The client should be insulated from the storage and access details as much as possible. There should be a clean separation of the data access code from the developer's representation of those data as business objects and from their implementation of the business logic with those objects.

While this advice is often given, it is rarely observed because it is advice that is difficult to follow. DevForce is one of a small number of products that make the job easier.

DevForce supports a five layer application architecture, as shown below, consisting of

1. **Data store**
2. **Data Access**
3. **Business Object Model**
4. Presentation View
5. Presentation Controller



The first three layers are the most pertinent to the present discussion.

Notice that the DevForce "Persistence Server" is responsible for interacting with the data store. It translates the data into business objects before handing them on to the DevForce "Persistence Manager". The business object state and logic execute primarily within the model where they engage with the rest of the application as it is experienced by the end user. The "Persistence Manager" sends new and modified business objects back to the "Persistence Server" which translates them into data before updating the data store.

The Model layer never interacts directly with the data store. The Model establishes a secure connection to the "Persistence Server" and henceforth relies upon that server to perform all "persistence operations" necessary to satisfy its requests. This "Persistence Server", as you may have guessed, is a key component of the BOS.

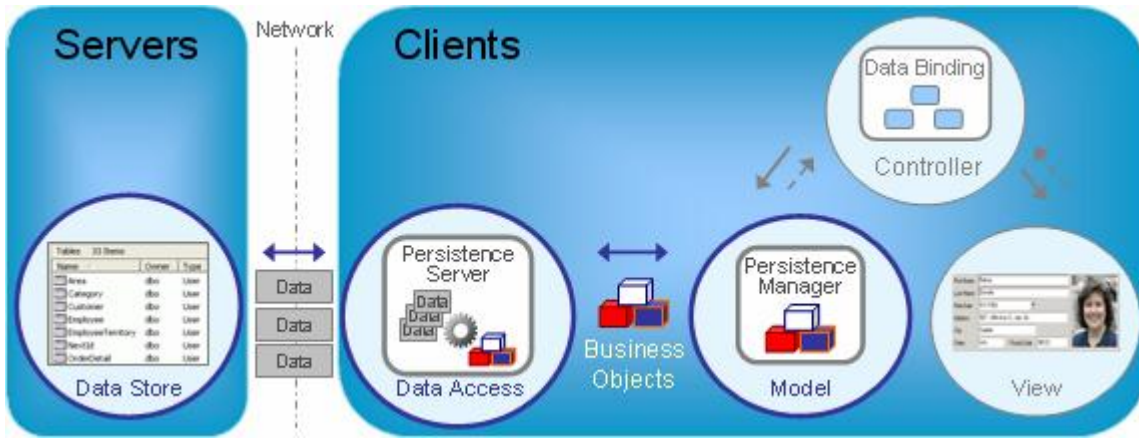
This is a logical architectural layering that is manifested physically as separate class libraries (dlls) with their separate responsibilities.

## Client / Server versus Multi-Tier

The logical layering does not determine the physical deployment of the application. Far from it.

Programmers often find it convenient to host all five layers on the same PC, freeing them from dependence on (or interference with) their developer colleagues.

In a client/server deployment, four layers remain on the client; the data store(s) are re-located to server(s) across a network as seen here:



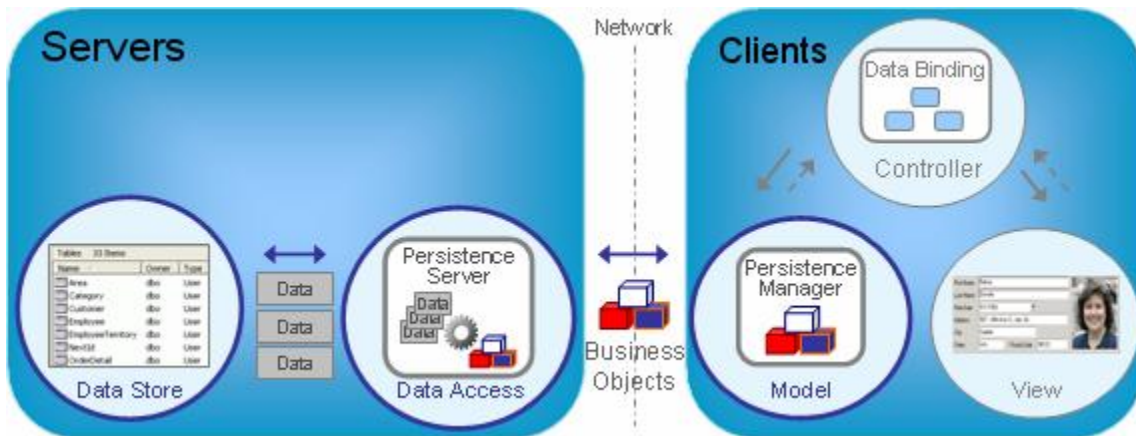
The “Persistence Server” is still in close proximity to the “Model”. In fact, it ought to execute in the same process as the “Model” to minimize the performance cost of exchanges across a process boundary; such is the case with the DevForce “Persistence Server” in this client/server or “2-tier” topology.

This also means that there is no “intelligence” on the server side of the network. The core client application – represented by the Model-View-Controller complex – may be ignorant of the server connections. But the “Persistence Server” must know and must have the privileges necessary to do its work with the data store. If there are multiple data stores – and there will be in a web service world – there will be multiple connections, each a potential security risk.

Moreover, raw database commands and raw data are flowing across the network; we may need to secure these – a difficult challenge over an open network. If there is significant network latency – either low bandwidth or long round trips – there could be performance issues.

## Multi-Tier with the BOS

These and other considerations motivate us to move the “Persistence Server” to the server side of the network.



There is a comprehensive review of these “other considerations” below.

A layered architecture that separates the Data Access Layer from the business object model is certainly essential to this transition. It would be impossible without a clear separation of responsibilities. If the client application could continue to read and write the data store directly, we would have little confidence in achieving our purpose.

While there are a number of persistence-oriented products that can justly claim to offer a distinct Data Access Layer, only a select few can relocate that layer to a separate physical tier on the other side of a network. It is one thing to isolate data access code; it is considerably more challenging to write something like the BOS.

## The Middle Tier Challenge

“Isn’t the BOS just fancy re-packaging of the DevForce client / server persistence layer?”

We hear this question regularly and it’s a reasonable question. Folks want to know that there is significant added value when they license the BOS. We obscure that value whenever we describe our two-tier persistence layer as a “mini-BOS”<sup>2</sup>.

We hasten to emphasize that the BOS is much more than a Data Access Layer (DAL). But even considered narrowly in this aspect alone, a middle-tier DAL, supporting a distributed client base, differs dramatically from a DAL on the client.

A DAL on a server must address several fundamental issues including:

- Transport of commands and data across a process boundary and over a network.
- Security of the connection.
- Support for multiple clients.

A client-side DAL is free from these concerns.

Client code calling ADO.NET does the work of connecting to the database, forwarding commands, and moving data. There is no need to ship data over a process boundary – or even a thread boundary. Database operations occur synchronously on the client UI thread.

The database manager performs the authentication, making use of information in the connection string. The company network is responsible for confidentiality of the content. The DAL developer typically has little to offer. In practice, what passes for security in a client / server application is almost always implemented outside the DAL and integrates with it at few touch points around connection initialization.

The client DAL serves one customer – the client application. If the DAL crashes, only one client is harmed. If the client application crashes, it takes the DAL down with it – but only one client is harmed.

We can’t talk about the DAL without also talking about the consumer of DAL services, the business model.

Here too the proximity and simplicity of the DAL reduces the strain on the model developer. We can move data between the DAL and the model simply by handing object references from one layer to another. We don’t need the complicated proxies and serialization that are essential when transmitting data across a process boundary.

The model need not worry about losing or recovering its connection with the DAL because both layers are running within the same thread, application, and machine. DAL / Model security is a non-issue for the same reason.

All of this changes when the DAL migrates across a network and begins to serve multiple users. Both the DAL and the Model have to “grow up”.

---

<sup>2</sup> “Mini-BOS” is a catchy sound bite conveying the point that many BOS features are available (if scaled down) in the client / server mode.

The DevForce BOS, considered simply as a middle-tier-ready DAL,

- supports multiple, simultaneous clients on separate threads,
- insulates the server as a whole from exceptions thrown in a client thread,
- marshals business objects across processes and over multiple network transports,
- caches prepared SQL statements for performance,
- exploits connection pooling for scalability,
- affords tight security at connection and during every persistence operation,
- is stateless for maximal reliability and minimal footprint,
- compresses data transmitted to clients,
- can be hosted in a variety of server environments.

These features are meaningless for a client-side DAL ... and essential for a middle-tier DAL.

Their implementation requires expertise in multi-threaded programming and intimate knowledge of arcane .NET technologies which are themselves evolving as Redmond ships new products and patches.

Failure at the middle tier could crash hundreds or thousands of client sessions at great cost to the business; defensive coding techniques and extensive testing are a must.

The expectations for reliability and capability elevate the BOS to an entirely different league from its seemingly similar client-side DAL cousin.

## BOS Data Access

With few exceptions, the DevForce “Persistence Service” should handle all communications between the client application and database or web service data sources. Client programs make no SQL or web service calls. They issue structured query and save requests to DevForce “Persistence Manager” interfaces; the “Persistence Manager” pre-processes those requests before forwarding them to the “Persistence Service” which is solely responsible for communication with the applicable data source managers.

When the BOS is in play, the client machine is completely insulated from the data stores. No code executing on the client can reach a data source. In fact, the client typically knows nothing more about its data sources than their general type (database or web service).

The client never sees the connection string; it refers to data sources symbolically – by a “DatasourceKey” name. For example, client code might compose a request for business objects from a data source called “blue”. The BOS resolves “blue” to a concrete database that it alone can access.

The BOS and the client always exchange “business objects”, never data in the raw form returned by the data source managers. The BOS converts client requests into the commands required to store and extract data and translates between the data source representation of the data and their business object representation. The BOS serializes<sup>3</sup> the business objects so they can be transmitted over any network; DevForce reconstitutes the serialized stream as business objects back on the client.

We tend to say that business objects flow over the wire. In fact, it is the persistent data of those business objects that flow over the wire. We get the effect of full business object mobility – including consistency of business object behavior – by ensuring that the business object types are the same on both server and client. After an exchange between client and the BOS, for example, an instance of an employee object on the server is essentially the same as that employee instance on the client.

---

<sup>3</sup> DevForce provides its own serialization logic to compensate for errors and deficiencies in native .NET serialization.

The BOS intercepts data source operation exceptions and forwards them to the client.

The BOS handles transactions – including distributed, cross data source transactions. Client-side save requests are transactional by default and typically require no conscious effort by the developer. The “Persistence Manager” collects all entities that are ready to save into a single list<sup>4</sup>; the list may contain entities of many types and may span multiple data sources. The “Persistence Manager” forwards this list to the BOS which attempts to save them as a single transaction (a distributed transaction if need be).

This approach is easy on the developer and ensures that transactions are both opened and closed efficiently.

## Performance and Scalability

*Performance* is measured by how quickly the applications responds to the user. *Scalability* is measured by the change in performance as the application works harder (e.g., as we add users).

We say the application *performs well* under a given load if every active user gets what she perceives as an immediate response from her every gesture. The application performs well relative to an alternative if users can complete a set of routine tasks faster than with the alternative.

We say the application *scales well* if its performance is mostly constant as the number of simultaneous users increases.

There are no absolute values in either metric but we can measure them and compare those measurements to our anticipated needs.

If we want to replace a system, we should identify routine tasks (AKA “use cases”) and time them so that we have a benchmark against which to compare the performance of its replacement. We may be unhappy with the benchmark and insist on better performance. But at least we will know what we mean by “performance” and can demonstrate that, at minimum, we’ve done no worse.

The benchmark may also protect us from the common vice of paying for performance that doesn’t matter. This isn’t an excuse for needlessly squandering performance with poor technique. But a 400% performance advantage that translates to a quarter second response is rarely worth cleverness we can neither understand nor maintain.

We need a similarly rational approach to scalability. Do we know how many simultaneous users the application must support? Where will those users be and how will they be equipped? Go ahead, add some SWAG, and then make decisions that are relevant for that number, not some fantasy number.

## Add a Tier, Lose Performance

It is that simple ... when all other factors are constant.

Adding a middle tier bleeds performance because we must:

- establish cross-tier connections,
- serialize and de-serialize data each way,
- authenticate and authorize each transfer.

How much do you lose? It always “depends”. A typical rule of thumb is that a server trip takes about twice as long when we introduce a middle tier<sup>5</sup>.

Will this matter? It might. Then again, it might not.

---

<sup>4</sup> The developer can manipulate this list prior to the actual save.

<sup>5</sup> See Lhotka, *Expert C# 2005 Business Objects*, p.16.

Consider the application that caches its business objects and only crosses tier boundaries when it needs new data or is saving changed data. The measured routine task

- takes ten minutes
- relies on cached objects throughout
- makes a single half-second trip to fetch the data in two-tier
- makes a single half-second trip to save in two-tier.

This task will take one second longer when configured for a middle tier. Spread over the ten minutes that's a task performance loss of 0.16%.

Let's admit that the psychological impact is far greater. The user will feel minor annoyance at the hiccups at the start and finish. Maybe that's worth programming around with some asynchronous fetch or save cleverness. It probably isn't.

The important observation is that what seemed a significant performance degradation in the abstract amounted to nothing when assessed in context.

We shouldn't burn any performance needlessly; if all conditions are equal, we'd be better off two-tier than n-tier with the BOS. Of course circumstances are never equal. The human and capital overhead of introducing a middle tier is a far better reason to resist a middle tier. And a host of factors – we'll get to them shortly – can sway the balance far in favor of the BOS.

## The Effect of the Network

We can't beat the performance of not going to the data source at all. Clearly the best way to improve performance is to minimize trips to the database.

The next best thing is a database on the client itself. This is actually a viable approach for many applications with large amounts of mostly static data. DevForce applications can work with business object models sourced simultaneously from remote and local data stores; it may be feasible to rely on remote sources for volatile data and local sources for "invariant" data<sup>6</sup>.

However, most enterprise applications draw upon networked data stores such as the relational databases that we'll consider in this discussion. Let's turn our attention to this more usual case.

Both two-tier and multi-tier deployments move data across the network between server and client. That's why we omitted "network latency" from our inventory of factors in the previous analysis. After all, if the clients are using the same network, latency is a wash, right?

Well, ... not right.

In the two-tier, client / server mode, the application issues database requests directly to the database and receives raw binary database data in return. The DevForce client-side "Persistence Service" translates those data into business objects and passes references to those objects on to the application.

Serialized business objects flow over the wire, not database data, when the BOS gets involved. Although BOS binary serialization is more compact than native .NET serialization, the business object data are somewhat larger than the equivalent raw database data. This difference is overwhelmed by the benefits of data compression. The BOS compresses the serialized data before transmission, resulting in shipments that are typically about a tenth of their original size.<sup>7</sup>

---

<sup>6</sup> Off-the-shelf replication can update the local "invariant" store during off-peak hours.

<sup>7</sup> We bleed a little performance performing the compression. The amount appears to be small measured on today's hardware.

Data compression is not available today in a two-tier model using the leading commercial database products and standard ADO.NET. Therefore, on balance, the average BOS payload is somewhere between a fifth and an eighth of the size of the two-tier payload.

Does this matter? It depends upon your network.

### *Speed Tests*

We ran some tests to determine if variations in network latency make a difference. The specific numbers are not important but the trends are<sup>8</sup>.

We looked at two query scenarios

The first fetched an average of three objects per query; we were interested in how many of these queries would complete in a second. The second repeatedly fetched a little over 800 objects at a time; we were interested in how many objects we could fetch in per second.

The objects in each scenario were of typical size, mapping to approximately twenty data columns.

We executed the queries in four operating contexts. We ran each query both as a two-tier application and as a BOS application hosted in IIS. We ran them over the company local area network (LAN) at a nominal 100Mbps. We ran them over a typical wide area network (WAN) – a commercial cable connection between the tester’s home and the office (yes, he used a VPN).

The **small query** yielded the following results

Queries per second	LAN	WAN
<b>2-Tier</b>	60	13
<b>BOS</b>	34 (56%)	8 (62%)

The WAN is about 20% as fast as the LAN. The BOS query is a bit better than half as fast as the two-tier query as predicted by our rule of thumb. There appears to be LAN / WAN distinction has little effect on the relative performance of the two deployment modes. We are inclined to conclude that the amount of data transmitted in each query benefits hardly at all from compression.

The **larger query** yielded the following results

Objects per second	LAN	WAN
<b>2-Tier</b>	8760	388
<b>BOS</b>	4620 (53%)	839 (216%)

The LAN / WAN impact is dramatic. The relative speed of BOS to two-tier remains the same on the LAN (56% for the small query versus 53% the larger query). But the BOS is better than twice as fast as two-tier over the WAN.

---

<sup>8</sup> Details of the tests including the hardware, software, and network characteristics are available by request.

## *Interpreting the Tests*

Clearly network latency must be factored into our performance predictions.

Considering raw, single-user performance speed alone – a dangerous metric:

- Two-tier always wins on a LAN
- The BOS wins on the WAN for medium to large queries
- Two-tier wins on the WAN for frequent small queries

## *About the Queries*

There's an important performance tuning story in the choice of these test queries.

**Small query performance doesn't matter.** The fastest small query case (2-tier, LAN) completed 60 queries per second; in the worst case (BOS, WAN) it was 8. That's a 7x difference! But does it matter? The worst case query completes in ~ 1/10 of a second. Can the user tell the difference between 1/10<sup>th</sup> and 1/70<sup>th</sup> of a second?

We don't have to worry about one small query. We should worry about firing lots of small queries in a row.

**Response time always suffers when we issue many small queries**, regardless of deployment mode or network. The small query returned three objects on average; in the best case (two-tier on the LAN), we could complete 60 queries per second; that's a mere 120 objects returned per second. The worst performing larger query (two-tier on the WAN) delivered 390.

Application users can't click a button fast enough to intentionally post a flurry of small queries. User's deliberate searches will never be the problem. The application itself, on the other hand, is more than capable of launching a storm of small queries. See the sidebar for an example.

### Query Storms

Binding grid columns to business object relation properties is a vivid example.

Imagine a grid of a hundred **Order** objects that displays ten rows at a time. In our naïve implementation, we query for the orders alone.

Each order row displays the customer name and the total price. These columns are bound to the **Order**'s **Customer.Name** property and its custom **TotalPrice** property. The **Customer.Name** property gets the name from the related **Customer** object; **TotalPrice** sums the prices of the **Order**'s detail objects. That's two related-object reads per row.

Imagine that the user scrolls down the grid. Every time the row changes, data binding reads the two properties. Initially the properties fail to find the related **Customer** or **OrderDetail** objects in the cache so DevForce fires off two smallish queries. This happens every row.

We hardly notice at 2/70ths of a second per row. We will surely notice at 2/10ths of a second per row because it takes a full second to cover five rows. It probably took two seconds to load the grid at the start.

Suppose the user clicks the column header to sort by customer name. The grid freezes. Most grids fire every property on every row while they sort. That single click yields roughly two hundred queries. It will take over a second to sort the grid even in 2-tier on the LAN. Fortunately, subsequent scrolling performs "instantly" because the cache now holds all of the related objects..

This kind of problem starts as an annoyance on a 2-tier LAN application, lurking somewhere near the bottom of the defect priority queue. It rockets right to the top when we move to the WAN.

The solution is to pre-cache the related **Customer** and **OrderDetail** objects at the same time that we fetch the initial **Order** objects. This is easy to do with DevForce span queries.

**Medium to large queries are the norm.** In a typical task sequence, a user searches for a list of main entities, picks one, and drills in to reveal its rich detail.

The list query is probably of medium size much like our “larger” test query. Of course we cache it so that subsequent visits to the list cost nothing. When we drill in we could issue a bigger query. For example, if the main entity is a sales rep, we should probably fetch the sale rep’s orders, the related customer records, the details for those orders, and the products for each detail. That’s a healthy slug of data.

**Minimize trips to the server.** You probably noticed that my drill-in example consisted of many separate smaller queries. Isn’t that the problem we identified earlier? It sure would be if we posted each component query independently. But with the DevForce “span query” we can fetch all of them in a single trip to the server.

Further research has confirmed our experience that a query for a single type performs about as well as a span query returning multiple types. This makes sense in all scenarios. The network is the bottleneck, not the database. The benefit is greater still in the BOS-over-WAN case because data compression squeezes the combined results of all the queries into a single package; there’s far less to compress if we transmit each query result individually.

**Consider Async Queries.** Performance is mostly perception. The user wants the screen to snap into place. The data she can’t see won’t matter in the first few seconds after the screen loads. What if the screen filled immediately with the “first impression” query results and you fetched the balance in background while she was still thinking? That’s easy to do with a combination of DevForce asynchronous and paged queries.

**Look at your scenarios.** Don’t you find that users are more effective when presented with a comfortable load of data, an amount that they can peruse visually? Don’t they hate an application that feeds them a spoonful at a time? Aren’t they overwhelmed by too many query results?

Enterprise applications are about rich human interaction with medium volumes of data. We’re not talking about optimizing transmission of terabytes; that’s a problem more appropriate for a weather model or arbitrage system. DevForce is more attuned to the needs of a “line of business” client application like CRM, patient records, or inventory management wherein knowledge workers search for and enter business information.

**Give yourself time to go n-Tier.** DevForce makes it easy to redeploy your two-tier application as a BOS-enabled multi-tier application. You may be tempted to hold off while all of your users are on a LAN in hopes of switching quickly when the time to connect widely arrives.

Unfortunately, the LAN’s superior performance can disguise the many inefficiencies hiding in your application, especially the undetected places where relation property databinding produces a flurry of tiny queries. These inefficiencies will bite you when you turn on the BOS.

You may want to rethink a few screens that worked great on the LAN but are impossibly slow on the WAN and can’t be improved by query consolidation or async queries. Give yourself enough time to find and fix these “non-functional gotchas”.

## Performance of Web Services

We’ve been comparing two-tier and BOS applications as they fetch data from a database. We did not mention web services. How do web services affect the analysis?

First, web services are another species of middle tier software because they always stand between clients and a backing data store. If a web service mediates the client’s access to data residing in a relational database, the web service is sure to perform no better than the BOS. It faces the same challenges as the BOS and there is no magic that makes those challenges easier to overcome for a web service.

In practice, web services never perform as well the BOS because they must sacrifice performance for portability. They are verbose. They can’t use data compression. They can assume almost nothing about their clients and therefore cannot benefit from proprietary technologies such as .NET remoting.

There may be good reasons to use web services in your applications; performance is never one of them.

## Performance Summary

We've looked at the drivers of pure performance.

It's clear that if you consider raw speed alone and every user is on the LAN, the two-tier approach is the winner. If all of your users will be hard-wired to a high speed LAN in one facility, you could be better without the BOS.

Raw speed over the LAN is rarely the only consideration. We've already seen that performance over a WAN is usually better with the BOS than without.

Will you have wireless users in your facility? Remember that even 802.11b with its theoretical speed of 54 Mbs delivers no better than half that speed in the most favorable circumstances. It's real performance is much more like the WAN we tested. The BOS can help.

Have you considered the productivity gains when users work remotely? Can you attach a cost to the productivity lost if they cannot or if the queries run half as fast in two-tier as they would under the BOS?

Do you have security requirements?

Will there be more than fifty simultaneous users?

Do you have business partners who should have access to the application but not to your network?

Do you need to monitor the health and activity of the client applications?

If you answer "yes" to any of these question, you should consider the BOS.

### *It doesn't have to be "either / or"*

What if you have a combination of remote and LAN-only users?

You can deploy the same application – the exact same executables – in both two-tier and multi-tier modes. The difference is purely a matter of runtime configuration. The machine plugged into the LAN can run two-tier; when it's remote it can run multi-tier.

### *Measure performance against real scenarios*

We mentioned it earlier; it bears repeating.

Performance is a measure of real work accomplished over a period of time. The speed of an individual query could be irrelevant and you can probably optimize a particularly bad one.

DevForce caching enables you to radically reduce the number of trips to the server; that makes a huge difference in task performance, especially as the cache fills up with frequently referenced data.

Asynchronous queries, run in background, can refresh stale cached objects and improve the apparent responsiveness of the application.

You can retrieve an extended business object graph in a single trip to the server with span queries.

These techniques will virtually eliminate the two-tier LAN speed advantage when we measure the overall performance of real business scenarios.

The point: you don't have to sacrifice performance to get the benefits of the BOS.

## Scalability

Fault tolerance, load balancing, and scaling are simple and inexpensive.

The BOS is stateless. It retains no information about client sessions. Client 'A' can request data from one BOS instance, alter the data, and save the changes back through a second BOS instance. Each exchange is atomic and the choice of one server or another is transparent to both client and server.

The BOS is focused and efficient. It is devoted primarily to managing object and data flows. It doesn't construct user-interface html or manage session state and it rarely executes business logic. Therefore it can support more active users than an alternative application server with wider responsibilities.

Greater efficiency over a given user load means fewer servers to buy and maintain. It means greater resilience in the face of changing user load. Having fewer *stateless* servers simplifies load balancing. Fewer moving parts and fewer interdependencies reduce overall server costs and improve fault tolerance.

## Security

A distributed application that uses the BOS can be much more secure than a client/server application.

Confidentiality of data flowing over the network is often a key concern especially over wireless networks. SSL encryption is available when deploying under IIS. Custom encryption can be inserted into to the communication stream.

Both client and server must authenticate each other. Clients log in to the BOS via the developer's custom implementation of a DevForce `LoginManager`. The `ILoginManager` interface lends itself to virtually any authentication mechanism imaginable. DevForce offers a number of ready-to-go example implementations.

The BOS constructs an encrypted "Session Bundle" that identifies the user during the session. This bundle must accompany every exchange between BOS and client, ensuring the authenticity of these exchanges. Inside the bundle is an instance of the application user's `IPrincipal` object, created during login. The bundle also carries information necessary to reconstruct the login on other BOS servers.

The rules governing what each user is allowed to do tend to be application-specific. Logic within the client-side executable is the first line of defense. Some applications require a second, tamper-proof line on the server. The BOS executes a `QuerySecurityCheck` method on every client query and a `SaveSecurityCheck` for every client save. The developer can customize these methods, inspecting every query and save to ensure that the current user is authorized to perform these operations. These methods will execute on the server automatically, every time. No mistake or malfeasance on the client can thwart these checks.

Applications require sensitive information to connect to their data sources. The database connection string, for example, may contain a username and password. Usernames and passwords should never reside on client machines: not in a configuration file, not in the body of the code, not in an obscure location, not even in obfuscated or encrypted form. We must assume that a determined individual can discover any data if in possession of the client machine (perhaps a stolen laptop).

A properly constructed DevForce application does not keep connection information on the client. The client application submits data requests to the BOS using a symbolic `DataSourceKeyName`. The BOS translates that name into a real connection to an actual data source based on information in a secure place on the server.

## HTTP Connectivity and Firewalls

Client / Server applications communicate directly with their databases. Such communications are either impossible or impractical over HTTP (port 80).

DevForce distributed applications communicate business objects and related requests over HTTP. Any .NET client that can access the internet with a browser will be able to access the DevForce application. We need only configure one or two xml files that identify the ports and addresses and deploy them, along with the BOS and business object assemblies, behind a firewall and NAT (network address translation).

## Remoting

DevForce uses .NET remoting for exchanges between clients and the BOS. Remoting is the technology that best combines performance and extensibility for .NET applications today.

Many developers do not realize that remoted applications can communicate over HTTP. The DevForce implementation uses binary serialization to format the business object data and transports these data over the .NET remoting HTTP channel which makes them as accessible as output from a web service.

Remoting itself is a forbidding technical challenge. Fortunately, the DevForce implementation makes the process seamless to you and your application.

DevForce will incorporate Microsoft Windows Communication Foundation (WCF, formerly known as Indigo) when it is released. The change will be transparent, requiring no modification to your existing application.

## Central Monitoring

Enterprise customers with demanding requirements for application availability need mechanisms to detect and respond to application anomalies. They need to examine the record of application activity for usage patterns and performance bottlenecks.

DevForce applications routinely record application activity to an XML file on the client called the “debug log”. DevForce records data source queries and saves by default; we can log our own messages as well.

When deployed as a distributed application, the client no longer accesses the data source(s) directly. The client-side record of such access stops although we can continue to capture user activity and extraordinary events that may help us diagnose application failures or misadventure.

The BOS, meanwhile, maintains its own, server-side log, where it records its own data source accesses on behalf of its clients. We can supplement that log with custom entries via the RPC and security check methods.

Both the BOS and the client applications publish their log entries as they write them. We can watch remotely for changes to all logs, as they happen, by means of the DevForce Trace Viewer. We can subscribe to log events with our own auxiliary programs that detect unusual conditions and alert external systems as appropriate.

## Server-Side Application Logic (RPC)

We may have application logic that must execute on the server. Perhaps that logic involves proprietary code that must reside and run only inside the firewall. Perhaps the application launches a process that engages expensive resources only available in the hosting environment.

A DevForce client application can make a “Remote Procedure Call” (RPC) to the BOS that invokes code resident on the server. The BOS services the call in the same manner as it does business object requests. It applies the same security constraints and marshals the data back and forth using the same protocols.

## Managing Limited Bandwidth

Applications that depend upon low bandwidth connections such as phone and cellular networks can be painfully slow. A distributed DevForce application conserves bandwidth in two ways:

1. DevForce clients cache their business objects. They only make a trip to the server when saving or requesting fresh data. DevForce only sends additions and changes during a save.
2. All exchanges between the BOS and clients are compressed, yielding 10:1 reductions in payload size for a typical database data.

## Deployment Options

A distributed application deployment involves four ingredients:

1. The Business Object Server executable.
2. The business model assemblies (we exclude the UI assemblies).
3. Configuration file.
4. Optional helper assemblies such as those invoked by a remote procedure call (RPC).

We place these elements together in a single directory on the target server machine, configured to run in one of three forms: the console application, the windows service, or the IIS application.

Console server deployment is the quick approach most suitable during development. Windows service deployment is a good choice for production deployments that do not require multi-processor or SSL support. Both console and windows service deployments can use port 80 if IIS is not also active on the host machine. An IIS-hosted server application is appropriate for multi-processor applications and SSL support.

## Conclusion

Many applications are deployed initially as 2-tier, “client / server” applications. The application grows and scalability, security, connectivity, or monitoring concerns move to the fore.

A 2-tier DevForce application can quickly address these concerns with a simple configuration change and the addition of the Business Object Server. We’ve seen the transition take as little as half an hour. Please contact [sales@IdeaBlade.com](mailto:sales@IdeaBlade.com) to learn more and acquire a license to deploy your application as a distributed, n-tier application.