Part 4

Broader topics

In Arkansas, we gaze at the rain-swollen Little Missouri River. We look forward to running this Ozark jewel, which is rarely this high. Our egos, too, are swelled. This river, now merely the promise of a fun diversion, would have been well beyond our skill level a mere two years ago. As we suit up, we plan our run, discussing strategy and safety issues. I notice a partner frantically scrambling through our gear and realize that we've forgotten to pack a spray skirt. The function of a skirt is to seal water out of the kayak. After eight hours of driving, we'll have no run today.

In Arkansas, we painfully learned that issues like packing and strategizing can be as important as fundamental skills like paddling. The same holds true of EJB development. Part 4 of *Bitter EJB* addresses secondary issues like tuning and packaging.

In chapter 9, we discuss the importance of good performance tuning techniques. We emphasize the need for an automated test suite and the importance of testing before making assumptions about performance. In chapter 10, we discuss the issues of building, testing, and packaging an application. We look into tools like XDoclet and Ant that make the build process easier to automate. And we underscore the importance of running automated tests. In chapter 11, we peek into the future of EJB, pointing to technologies that may play a crucial role in the future of EJB.

9 Bitter tunes

This chapter covers

- Definitions of performance
- Antipatterns related to the EJB performance tuning process
- A JUnitPerf tutorial
- Tuning an example EJB application using JUnitPerf tests
- A step-by-step performance testing methodology
- Techniques for automating performance testing

It's early in the morning, and I'm locked in tightly to my new snowboard, staring anxiously down the impossibly steep slope. I'm a skier who's grown increasingly addicted to the freedom of snowboarding, and I've learned quickly. But I'm having a tough time getting to the next level—the confident level of the elite boarder. With a twist of the hips, I accelerate downhill. I mechanically hammer through a couple of turns, reacting to each tiny groove and bump in the ungroomed morning snow. My brain gradually falls behind, and my body only barely keeps up with the descent. I'm in a purely reactionary mode now, with my eyes tracking the terrain only inches in front of me. I fear that I may be unable to stop, and I certainly can't keep up this reckless pace. I wonder if I will even see the crash come.

In this chapter, we'll tour a few common pitfalls related to the EJB performance tuning process. We'll focus on developing a disciplined performance testing methodology driven, not by irrational fears or wild speculation, but by automated tests whose objective results aren't distracted by emotion. By continually measuring the performance of our code—and the impact of our changes to it—these tests will help us stay ahead of the pain endured when undetected performance problems sneak into our code.

Ah, but tuning isn't a development activity, you say. Configuring the application for its operational environment is a job suited for those other geeks—the operations folks strolling safely around the lodge—not those of us still on the mountain. Well, we could pass the buck that way, but letting performance tuning roll too far downhill is an incredibly inefficient way to develop software. At best, it introduces a costly delay in the feedback cycle between making a change intended to improve performance and seeing whether that change actually did any good. At worst, failure to start measuring performance early invites the danger that significant problems will crop up later, when redesigns are no longer economical. Instead, to maximize our time and ensure a successful rollout of our application, we must obtain immediate results on early performance testing.

In this chapter, we will consider an EJB application that suffers from poor performance. The application will employ a familiar antipattern that will serve as a crash test dummy for our performance testing methodology, letting us focus on tuning the application and measuring the impact of that tuning. Each time we ratchet the performance gear a notch, we'll receive immediate data that indicates unambiguously whether we've truly improved performance. By taking the guesswork out of the tuning process, we'll increase our confidence, allowing us to tackle new performance requirements without fear.

9.1 Measures of success

Before we shift into high gear, let's first nail down a definition of performance as a measurement. In general, two ways exist for viewing performance: response time and throughput. We tune and test applications differently, depending on the aspect on which we're focusing our improvements.

9.1.1 Response time

The *response time* of our application refers to the speed at which the application is able to service a given request, such as a user requesting a web page through a browser. The request may be serviced by any number of resources in our application, including servlets, EJB, a database, or a legacy system. We can manage certain types of resources by placing a limit on the maximum number of concurrent requests each resource can safely handle. That is, managed resources are control valves that help us throttle the application for consistent performance and stability. Consequently, each time a request requires the use of a managed resource, it may need to wait in a queue until the resource is available.

Take, for example, a limited resource familiar to most enterprise developers—database connections. Figure 9.1 (a) shows a database connection servicing a request for data. In this case, the database connection pool is sized with ample available connections capable of servicing requests without queuing. So no cost is incurred in waiting for a database connection to become available. In contrast, figure 9.1 (b) shows a queue of active requests waiting to be serviced by a single database connection. In this case, the size of the database connection pool is not able to keep up with the number of new requests without queuing. Step right up… and take a number!

From figure 9.1, we can infer that the response time of a request will include any time spent waiting in the request queue for an available database connection. The response time may also include any network latency in obtaining a database connection via a remote call. Furthermore, as concurrent requests for a database

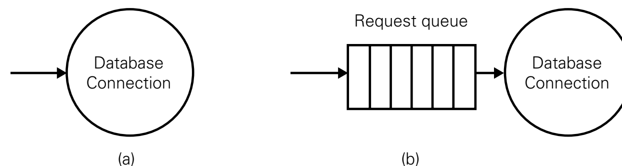

Figure 9.1 Database connections are examples of managed resources that may cause incoming requests to be queued before being serviced. Queuing incurs additional response time overhead.

connection increase, more requests will be queued, waiting for the connection. Therefore, to characterize the response time of our application accurately, we must take two essential measurements: the response time for a single request and the response time for the same request under a load of concurrent requests.

In section 9.7.2, we'll roll up our sleeves and write automated tests that measure the response time of a use case from our application. By continually running these tests, we should gain confidence, knowing that any optimizations we make have indeed improved response time. For now, let's begin by considering the possible measurements of such tests.

9.1.2 Throughput

While response time focuses on the speed of a specific request, *throughput* measures the number of requests our application can service in a given amount of time. For EJB applications, throughput is typically measured as the number of business transactions per second (tps). What constitutes an average business transaction is certainly application-specific, so throughput metrics always must be taken in appropriate context. For example, our application might be capable of processing 10 product catalog queries per second, with each query returning an average of five products.

From a slightly different angle, we use throughput as an indicator of our application's potential to scale. *Scalability* is a measure of a load's affect on our application's performance. For example, if we say that our application can scale to handle five concurrent users, then we're referring to the application's ability to maintain a linear (not exponential) average response time for each user, while under the stress of a five-user load.

Applications that scale well can deliver increasingly higher levels of throughput by adding resources, such as more hardware or more connections within a pool. When the average response time of a business transaction becomes intolerable under load, the application has reached its *maximum effective throughput*. Stressing the application beyond this point by piling on a heavier load will further degrade its responsiveness.

Revisiting our example of database connections as managed resources, figure 9.2 shows how a database connection pool can be used to work off requests efficiently. As the number of concurrent requests increases, the single database connection in figure 9.2 (a) will eventually hit a wall. Try as it might, the connection won't be able to keep up with the number of pending requests in the queue. Consequently, the request queue will continue to grow, adding to the response time of all waiting requests. By tuning the size of the database connection pool to include two

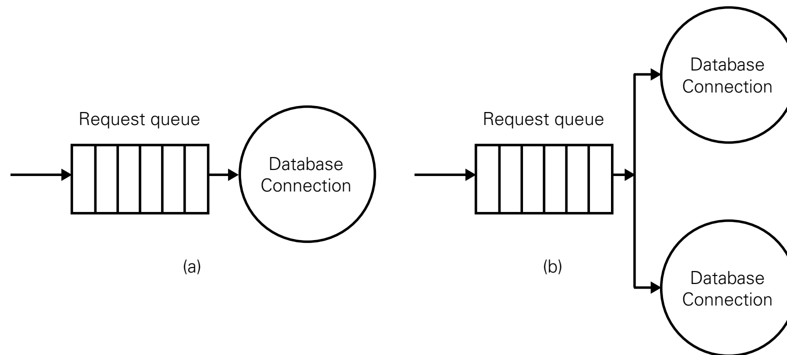

Figure 9.2 Pooling limited resources, such as database connections, is a common technique for improving an application's throughput. By increasing the size of the connection pool, a scalable application can take advantage of these additional resources to deliver better performance.

available connections, as shown in figure 9.2 (b), more requests can be serviced in a given amount of time. All other things being equal, figure 9.2 (b)'s connection pool will double the application's throughput depicted in figure 9.2 (a).

When attempts to get an application to scale prove frustrating, or even downright impossible, bottlenecks are the usual suspects. In general, a bottleneck is any chokepoint that restricts throughput. When a bottleneck is suspected, load testing tools are called to the rescue. Load testing tools are invaluable. They first put a load on an application, then shine light on any bottlenecks that rear their ugly heads. In section 9.7.5, we'll capture a nasty little bottleneck in the wild by writing an automated load test. Before we do, however, we must discuss an antipattern everybody knows but nobody likes to talk about: Premature Optimization.

9.2 Antipattern: Premature Optimization

Making a chunk of seemingly slow code faster can be quite satisfying. You can get a thrill from strutting your programming prowess and seeing immediate performance improvements. However, that thrill can cloud your judgment. Time passes in a flash as performance tweaking takes over, often resulting in an overly complex tangle of code that might get run only once in a blue moon. But when that code does run—man, is it fast!

Nobody is immune to the allure of Premature Optimization. We've been both victims of and witnesses to its use. Premature Optimization can take many forms: a speculative architecture decision, the choice of a particular design, a change in the runtime environment, or convoluted code.

Low-level code optimizations tend to attract your attention first. The trouble is, in most cases, the code paths you decide to optimize aren't called frequently enough to justify the time spent tuning the code. To make matters worse, the risk of wasting time optimizing arbitrary code paths increases with the code base size. Sure, you can always find a chunk of code that can be made faster, but it's usually the wrong one. Although you want your code to be reasonably efficient at all times, you gamble each time you blindly optimize code at the expense of code clarity and precious development time. Indeed, code clarity is often compromised as a result of optimization.

Premature Optimization has another insidious side effect. Tweaking code to make it faster tends to break something that's already working. That is, as you contort the code to squeeze out the last little bit of performance, you inevitably make compromises that can come back to bite you big later. Alas, your code won't win any awards for producing the wrong result quickly.

9.2.1 Tuning EJB applications blindfolded

EJB applications are especially unforgiving when you tune them in the dark. They inherently use at least one other resource, such as a database. As such, any arbitrary EJB method may spend more time blocked, waiting on a resource, than actually using the CPU. In that case, optimizing code won't show any significant performance improvement.

The runtime environment of an EJB application is also particularly fertile ground for the Premature Optimization antipattern. You must consider the virtual machine, database, and application server versions, as well as a dizzying array of internal tuning parameters for each version. If you try to get all these parameters adjusted for optimal performance before understanding their effect on your application, you incur at least two costs. First, you divert time away from those activities that really pay the bills. Second, you can complicate deployment unnecessarily by assuming that certain configuration parameters must be used.

Another reason EJB applications are prone to Premature Optimization is the myriad design decisions that must be weighed against performance. Indeed, it's easy to speculate on possible optimizations from the design perspective. Once you swerve onto that path, you may waste a significant amount of development time before you see any possible gain. Table 9.1 describes a few premature high-level design optimizations prevalent in EJB applications, along with their potential consequences.

At the end of the day, time spent tuning one area of our application is time not spent tuning another. It's a game of opportunity cost. Without a deep understanding of your application and the behaviors of its users, arbitrary

Table 9.1 Premature design optimizations in EJB applications may degrade performance and increase complexity. Deferring these types of optimizations until deemed necessary and beneficial is a better use of our time and resources.

Premature Optimization	Potential consequences
Entity beans	If your business object doesn't require concurrent read and write access while retaining stringent transactional integrity, then the use of an entity bean may incur unnecessary complexity and performance overhead. A servlet or session EJB using JDBC is often sufficient.
Stored procedures	Although stored procedures allow your database to do the heavy lifting, the business logic they encapsulate is tightly coupled to the database schema and may be written in a proprietary language that's difficult to maintain. Designing a business logic layer in your middle tier generally is easier to develop and maintain.
Bean-managed persistence	BMP entity beans may suffer from hard-coded SQL, difficult-to-maintain database logic, and $n + 1$ database calls to load n bean instances. Entity beans using container-managed persistence generally are more efficient and easier to develop, if used properly.
Custom primary key generator	If not designed carefully, custom primary key generators may require synchronization that becomes a scalability bottleneck. Better scalability, with less work, may be realized by using automatic primary key generators already provided by your database. To help, JDBC 3.0 includes new methods to facilitate the retrieval of automatically generated fields.
Caches	If the data in your cache is changing more often than it's being used, then the number of cache hits may not justify the complexity of caching while preserving data integrity.
Custom resource pools	The use of custom resource pools in the name of better performance may prevent your application server from managing resources effectively. Stability may deteriorate unless the pools already provided by your server are used.

optimizations are pure speculation. However, by first identifying the most valuable optimizations, whether at a high or low level, you can concentrate your efforts where they're most needed.

9.2.2 *Solution 1: Plan, but don't act (yet)*

Performance requirements are the solution for Premature Optimization. Without well-defined goals, you'll try forever to optimize every line of code you write to mitigate a performance backlash. However, by defining measurable goals for performance-critical use cases, you can optimize pragmatically, based on patterns in user behavior and data usage. Your energy is focused on solving the most critical performance issues first.

In our experience, the best performance gains are realized when following the advice given in the simple motto, "make it run, make it right, make it fast." Notice

that this advice speaks to the order in which you take action, not necessarily the order in which you consider the necessity of those actions. In other words, you should take action to improve performance only when not doing so would preclude you from delivering a successful application. The earlier you know what determines success, the better.

Knowing when to take action isn't always clear-cut. We're constantly trying to strike a delicate balance between optimizing the code we write today and building an application that can achieve expected levels of performance. On one hand, you want to keep the code efficient without racking up too much time tuning. On the other hand, you need to consider the performance requirements of your application early and often. If you don't keep in mind how your decisions might impact performance, chances are when you finally do look up, you'll be aimed straight for a tree. Nevertheless, you can avoid possible disasters by expending effort to improve performance only when you've gathered sufficient evidence to let you prioritize and focus on optimizations that will truly make a difference.

If you defer performance tuning until it's proven to be a high-yield investment, you'll have a chance to validate your design with working code and tests. At this point in the development cycle, you will be able to understand the design well enough to consider the potential benefits of global and local optimizations toward meeting performance goals. Better yet, with a solid foundation of tests, you will be able to tune safely, knowing that the tests will fail if tweaking code causes existing functionality to break. In the meantime, writing well-factored, modular code puts you in a position to tune economically down the road, if necessary.

9.2.3 Solution 2: Write well-factored, modular code

Until performance improvements are necessary, write code that is as simple and clean as possible. The time you'll save if you write the simplest and cleanest code as a matter of course can be used later to optimize those few places where code accidentally gets complex and laden.

If you find opportunities to improve performance, remember that simple designs that use well-factored, modular code are more amenable to performance tuning than more complicated designs. In general, well-factored code is easier to change. And code that's easy to change is easier to tune. By encapsulating implementation details, modular components can respond to change, allowing us to change their underlying code without breaking their clients. Moreover, well-factored, modular code exposes succinct methods that serve as excellent starting points for optimizing a particular code path. Take, for example, the method in

listing 9.1 responsible for withdrawing an amount from a bank account, designed as an entity EJB.

Listing 9.1 Code that is not well factored is also difficult to tune

```
public double withdraw(int accountId, double amount)
  throws Exception {

   InitialContext context = new InitialContext();

   Object homeRef = context.lookup("AccountHome");

   AccountHome accountHome =  (AccountHome)PortableRemoteObject.
    narrow(homeRef, AccountHome.class);

   Account account = accountHome.findByPrimaryKey(accountId);

   account.setBalance(account.getBalance() - amount);

   return account.getBalance();
 }
```

Find the specified account by its ID

Withdraw the specified amount of money

Notice how all the logic is inlined in the method. If we were to run a code profiler on this code, we would find it difficult to ascertain which piece of logic—finding the appropriate account or withdrawing the specified amount—consumes the most time. The profiler would merely decompose the overall method time into the individual execution times of each method invoked. However, we'd like to know which coarse-grained code path could benefit most from tuning. To make this monolithic method easier to read, let's refactor it a bit, as shown in listing 9.2.

Listing 9.2 Well-factored code enables a code profiler to help you tune effectively

```
double withdraw(int accountId, double amount) {

  Account account = findAccount(accountId);

  return account.withdraw(amount);
}
```

Find the specified account by its ID

Withdraw the specified amount of money

Our refactoring organized the inlined code into two distinct methods: `findAccount()` and `withdraw()`. In applying this refactoring, not only have we made the code simpler and more modular, we've also enabled the code profiler to help us. The code profiler can now quantify the individual cost of each code path and point us directly to the starting point of the most expensive path. And, as an added bonus, once we optimize a particular method, that method can be used by other components in our application, providing better performance many times over.

Now, let's consider what would happen if, instead of building performance into our design, we attempt to bolt on performance after the design is finished.

9.3 Antipattern: Performance Afterthoughts

Developing applications that perform well requires prior intent. If performance is important, it must be baked in to the application, not bolted on afterwards. We learned this lesson the hard way a few years back. (And one of us has the hairline to prove it!) The database we were using had a serious bottleneck in its locking strategy. When used with applications that read more data than they wrote, the database was lightning fast. Yet whenever multiple concurrent users attempted frequent database updates, this particular database was clearly the wrong tool for the job. Unfortunately, we didn't know the bottleneck existed until it was too late. Although we knew from the beginning that the application needed to scale in order to be successful, we didn't plan for scalability early enough. We assumed that the application would scale, and if it didn't, we figured we would have time later to refactor the application's design to be more scalable. Building a prototype that demonstrated a few performance-critical use cases under load would have alerted us earlier to the impending doom.

Just because you're using EJB technology doesn't mean you can disregard performance concerns. It's true that any worthy EJB container will help you manage resources for the best possible performance. However, perfuming a poorly performing application with the scent of EJB won't keep the flies away.

We can increase our application's probability of success by using simple designs tactically with well-factored, modular code, but that, too, is no substitute for strategically planning for performance. This presents a conundrum. If we delay considering performance until right before the application goes live, it's usually too little too late. Then again, we don't want to speculate on performance at the expense of rapidly delivering valuable software. The answer to this problem lies in continuous planning and measurement.

9.3.1 Solution: Plan early and often

To counterbalance premature optimization, we need to plan proactively for performance. That's not to say we should attempt to predict future performance demands and carve a plan in stone. We'll be sorely disappointed when, as often happens, things don't go according to that plan. Indeed, our perspective inevitably will alter as we learn more about our application and its users. Consequently, the performance plan is subject to frequent change. Planning for performance

requires that we constantly consider the current state and goals of our application, by taking measurements and making course corrections throughout the project.

As the delivery date approaches, no doubt we'll know which aspects of our application suffer from poor performance. Furthermore, we'll have more accurate estimates of the production load on our application and the respective hardware necessary to handle that load. Performance plans may change as a result of this information, and we should consider this a good thing.

In the meantime, the performance planning process will help us head off potential problems at the earliest opportunity. If we continually plan for performance, we'll be able to see obstacles in the terrain ahead and react in time to avoid a crash. Let's consider the following guidelines for performance planning:

1. ***Understand the application's usage patterns*** Users generally expect different levels of service, depending on the feature of the application they are using at the time. Users expect some use cases to respond rapidly and understand when others are slower. A web user, for example, expects to navigate a product catalog quickly. Yet, when an online order is placed, the same user will accept a delayed order confirmation via email. Understanding patterns in user behavior, and the data and resources required to support that behavior, provides invaluable input into the performance planning process.
2. ***Prioritize performance requirements*** To maximize your time and dollars, you should satisfy the performance requirement with the highest business value first. For example, optimizing a product catalog for maximum responsiveness when browsed under load is arguably a better investment than optimizing your email server for faster order confirmation. Once the top performance requirement has been demonstrated successfully, you can work on the next highest priority requirement. Rinse and repeat.
3. ***Write automated performance tests*** Performance tests that unambiguously define and validate the performance requirements of your application are essential in helping you meet desired performance goals. Without a target, you'll never know when you've hit the mark. Good performance tests express objective exit criteria in an executable format. In other words, running these tests will help you decide if tuning is necessary, and, if so, when tuning should stop. Tests also prevent tendencies to overoptimize based on speculation or commit too early to designs and infrastructure that seem to promise improved performance.
4. ***Build modular components*** Components that hide their implementation insulate the rest of the application from changes made to improve

performance. Using these components, you can start with a simple algorithm that works, even if it may incur a few extra seconds of overhead. As you learn more about your application and its uses, you can easily swap in a new, blazingly fast algorithm or data structure, for example.

5. ***Revise plans based on feedback*** Once performance goals have been identified and prioritized, you must demonstrate performance as early as possible to get feedback. You'll want to know sooner rather than later if you're making design decisions that may prohibit your application from meeting user demands. If you feed this information back into the planning process, you can steer the design to meet your performance goals continually. You can respond more readily to change, rather than dutifully marching to an inflexible master plan.
6. ***Understand your EJB server's configuration options*** In your haste to tear the wrapper off your new EJB server, a few finer features may go unnoticed. Server vendors differentiate themselves from their competitors, providing different knobs and levers you can twist and pull to improve performance. If you understand the available options, you'll know when to leverage rather than build for successful performance. Study and investigate the contents of the box. Then experiment and see what happens. Your tests will announce impending danger if your application starts to sputter.
7. ***Schedule the availability of production hardware*** Your plan should include testing on production-quality hardware as soon as possible. Indeed, how your application performs for real users is what matters most ultimately. Everything else is just preparation.

Remember, it's not the plan that's important, but the planning. With that in mind, let's get down in the trenches with a real, live application.

9.4 Grist for the tuning mill

Let's say we've accepted a mission to develop yet another online product catalog. Our customer group—those folks defining the requirements of our application—has decided that users want to browse a list of all products for a particular category within a product catalog. The initial user interface will be an HTML browser, but it's imperative that the catalog browsing service be available to other types of distributed clients. This requirement is not unlike the others we've been delivering, from which a service-oriented architecture has emerged.

We conclude that the simplest approach would be to publish a remote façade that encapsulates the business logic of querying a product catalog. We dub it the catalog service. A stateless session bean seems like the logical choice given our architecture and experience, so let's make it the centerpiece of our design.

9.4.1 *Putting an EJB to the test*

Before diving into the implementation of our catalog service, let's start by writing a test. Why? Well, how else will we know what code to write? If we write a test first, we'll have an example of the catalog service's intended use. In addition to demonstrating the intent of the catalog service, the test can validate automatically that the catalog service returns the correct results. Once the test passes, we're done!

Listing 9.3 shows the JUnit test, which queries for all products in the snowboard category of the product catalog. (Note: a full tutorial on JUnit goes beyond the scope of this chapter. The *JUnit Primer*[1] will help get you up and running quickly.)

Listing 9.3 Unit testing the product catalog service

```
public class CatalogTest extends TestCase {

  public CatalogTest(String name) {
    super(name);
  }

  public void testGetProducts() throws Exception {

    String snowboardCategory = "Snowboard";

    Catalog catalog = (Catalog)getCatalogHome().create();

    Collection products =
       catalog.getProductsByCategory(snowboardCategory);

    assertEquals(25, products.size());

    Iterator productIter = products.iterator();
    while (productIter.hasNext()) {
      ProductDetails product = (ProductDetails) productIter.next();
      assertEquals(snowboardCategory, product.getCategory());
    }

    catalog.remove();
  }
}
```

1 http://www.clarkware.com/articles/JUnitPrimer.html

The test case has a single test method, `testGetProducts()`, that starts by using the `Catalog` home interface to create a remote reference to a `Catalog` session bean instance. The `Catalog` remote interface represents a façade—a black box from the client's perspective—that finds products in a catalog. Using the remote `Catalog` reference, the test then queries for all products in the snowboard category. We expect exactly 25 products in this category because before running the test we created 25 example products in the snowboard category of the product catalog database. Iterating over the resulting collection of products, the test validates that the catalog service only returns the products in the snowboard category.

Excellent! Now there's just one problem: We have to get the test to pass.

9.4.2 Passing the test

To get the test to pass, we have to write the code for the `Catalog` EJB. All EJB components require a remote interface, home interface, and bean class. We won't actually show the code here because, frankly, the implementation doesn't matter. Instead, we'll just show the design of one possible solution. Our priority should be to begin by writing clean and simple code. We won't worry about performance here. We just want to validate that our design is usable and our code produces the correct results, thus avoiding the risk of overspending on performance too early. Running the test from the remote client's perspective gives us confidence that the catalog service is working as we expect. We can change and tune the underlying code without the fear that existing functionality might silently break. If it does, the test will surely let out a scream.

Under the hood in the `Catalog` bean class, we code the bean to use JDBC to query our product table directly through a database connection. The values in our database table are then packaged and returned to the client in a collection of lightweight `ProductDetails` DTOs. In future use cases, administrative users may update the products in a catalog. This updating might require a more complex persistence mechanism, for example, the type of persistence afforded by an entity bean. We'll cross that bridge when we get there. Right now we're concerned only with retrieving read-only product information, so a simple stateless session bean wired up to the database will do just fine. Figure 9.3 shows a UML sequence diagram illustrating the interaction of our recently built components.

Notice that all business logic occurs on the server side, behind the façade of the `Catalog` interface. As such, our design is modular. If performance becomes an issue, we can tune the code behind the `Catalog` interface without adversely affecting its remote clients. That's reassuring because we have a sneaky suspicion that tuning may be in our immediate future.

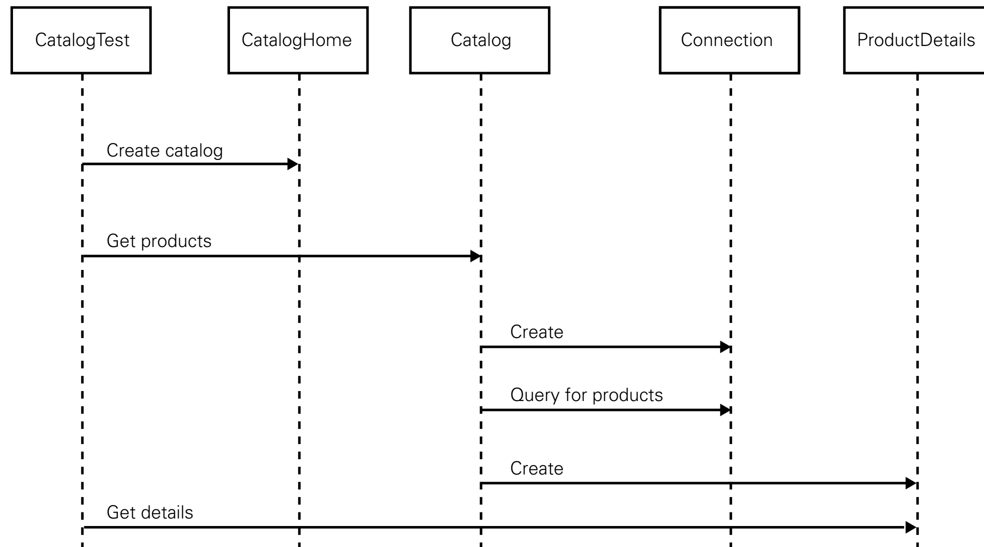

Figure 9.3 The Catalog stateless session bean, wired directly to the database, is used to query the catalog for all products in a product category. The test stands in for a client, demonstrating a usage scenario and validating that expected results are returned.

We've written just enough code to get the test to pass. We've made it work and made it right. Now, let's make it fast.

9.4.3 *Specifying response time as a measure of success*

Up until now, we've written simple code so that the test would pass. In doing so, we certainly didn't forget the hard-earned wisdom we gained designing distributed systems over the years. Rather, we made a concerted effort to determine the workability of our design and code first, before speculating on performance.

With much pride, we demonstrate the new catalog service to our customer, who is delighted. After kicking the tires a bit, our customer hunkers down at our test machine and proceeds to press the button that lists the snowboard category products over and over again. Our customer likes the fact that the new catalog service is working this soon in the schedule, so now we're ready to start polishing. The customer consensus is that the service just isn't fast enough; we need to improve performance.

The average response time of the web page to query and list 25 products is approximately 1.4 seconds. We think we can do better than that with a little tuning, but we need a way to measure success. So we ask our customer group to write

a new performance requirement. We watch as they scratch their heads and mumble a bit, but finally they draft a performance:

The average response time of the catalog web page listing up to 25 products should not exceed 1 second.

Great, now we have a goal!

9.4.4 Seeing light at the end of the tuning tunnel

To satisfy our new performance requirement, we need to shave off about a half-second of the response time, then stop tuning before we hit the point of diminishing returns. However, we're not sure whether to start chiseling in the presentation layer or the business logic layer. We want to focus our attention on tuning the sections of code that contribute the most overhead to the overall response time. Guessing would be a fool's game, so we put our trusty code profiler on the case to hunt down the busiest code. Table 9.2 shows the results:

Table 9.2 Using a code profiler takes the guesswork out of tuning by identifying hot spots worth optimizing. Always seek the advice of a code profiler before attempting to tune code.

Method	Total time (ms)
`CatalogEJB.getProductsByCategory()`	1327.0
`CatalogServlet.service()`	10.0
`CatalogEJB.getConnection()`	10.0
`CatalogEJB.runQuery()`	15.0
	1362.0

The code profiler identifies the `CatalogEJB.getProductsByCategory()` method as the major contributor to the total response time. Well, that rules out tuning the presentation layer for any observable gain in performance. The top contributor is the EJB method, not the servlet method that presents its results. The profiler results also rule out tuning the database interaction. The time it takes to obtain a database connection and execute the SQL query is insignificant compared to the business logic in the EJB.

We have no question that the culprit is the code in the EJB that transforms database rows into products. But when we're done tuning, how will we know we've made good progress? What if our tuning activities cause performance to take a step backward? Fear begins to envelop us as we're reminded of endless hours spent with the subject of our next antipattern—Thrash-tuning.

9.5 Antipattern: Thrash-tuning

I am barely a third of the way down the hill, contemplating disaster, when something finally clicks. Instead of focusing on every slight ripple in the snow, I concentrate only on obstacles that might impact my balance or my course down the mountain. I'm able to keep my eyes further ahead, improving my ability to plan and react. With this improvement I husband my waning strength better, saving it for the biggest challenges—like that tree just ahead! I've entered the zone that my mentors so often describe.

If you're always looking at your feet, you can't anticipate what's up ahead. And if you can't anticipate, you can't determine whether your microadjustments are making any progress toward the ultimate goal. Before long, you're bound to diverge hopelessly off course.

Thrash-tuning is a nasty habit born out of undisciplined performance tuning. You know the drill: change a tuning parameter here, tweak some code there, then run the application. *Is it faster? No, it actually seems to be slower! Interesting. Now which change caused that to happen?* Lacking clear knowledge, you repeat the cycle, over and over again.

Without a baseline to measure against, Thrash-tuning is entirely unpredictable. You may spend days tuning in circles with only a minor improvement to the application's overall performance. Sometimes you get lucky and increase performance by a single order of magnitude in a quick round of thrashing, but you'll soon find that's the exception. As a general rule, Thrash-tuning can consume hours or days of your life, without amounting to much good.

The following are common ways to invoke the curse of Thrash-tuning:

- **Changing more than one thing at a time** In the rush to get the most bang for your tuning buck, you change multiple things at once, but doing so makes the individual contributions of those changes indistinguishable from each other. Multiple changes also makes backing out a change that degrades performance difficult.
- **Forgetting to measure between changes** Without quantifiable evidence that a particular change improves performance, you can't clearly determine its impact. Performance goals always seem to be just beyond your grasp.
- **Not knowing when to stop** Remember, Thrash-tuning is a habit and a particularly hard one to kick. You may find yourself tuning endlessly. Performance tests are the cure, telling you when enough is enough.

If these scenarios sound all too familiar, you're not alone. Both novice and veterans alike have suffered similar fates at the hands of EJB applications. The sheer number of opportunities for improving EJB performance makes falling into a vicious tuning cycle deceivingly easy. At one end of the spectrum, you can change deployment descriptors quickly to dynamically influence performance. At the opposite end, you can apply high-level design patterns, using general knowledge gained from distributed computing.

The difficulty lies in choosing an approach and measuring its impact in isolation. A solution applied in hopes of improving performance for one aspect of our application may cause unwanted secondary effects in other areas. Consequently, performance tuning quickly turns into a delicate balancing act. We become painfully aware that multiple controls often exist for performance with complex interactions. Each new interaction increases the odds that making a change—one that theoretically should improve performance—may not make a difference.

Besides being incredibly frustrating, when thrash-tuning runs rampant, it has the potential to rob us of enormous amounts of development time. The more you scratch it, the more it itches. A sure-fire way to stop this irritation is to use a sound methodology with information derived from automated performance testing.

9.5.1 Solution: Use a performance testing methodology

The only defense we've found against falling prey to Thrash-tuning is the use of tests to gather evidence first, before making an attempt to improve performance. We've tried predicting whether an optimization would improve performance, and yet, after hours of navel gazing, we remained undecided. Although not as therapeutic as navel gazing, writing tests that measure the impact of changes has given us much better success.

To ensure that you're always ratcheting forward toward optimal performance, current performance must be automatically compared to a baseline. Doing so keeps the performance of your application from going off the rails. If you use a methodology like the following, your performance stays on track, never more than one change away from the last baseline:

1. Begin with clean and modular code that's easy to understand and modify, and driven by tests that express its intentions and expectations.
2. Choose quantifiable performance goals for the code.
3. Profile the code to identify hot spots with the highest return on investment.
4. Write and run an automated test that baselines current performance.

5. Make a change intended to improve performance.
6. Run the automated test again to measure the gain (or loss) in performance.
7. Repeat as often as necessary until the application meets its performance goals.

Notice that we use a test to measure both before and after a tuning change is made. The test tells us if the tuning did any good. If not, we can back out the change to arrive safely at the last good baseline. Bear in mind that we might need to give caches, pools, and other performance-enhancing mechanisms a chance to warm up before taking a measurement. Otherwise, the observed performance may be thrown off by a cold start. In other words, the test environment has to be predictable, and the tests must be repeatable.

This easy-to-use formula really shines when applied incrementally to solve the most pressing performance problem at any given time. Once performance has improved in one area, and a test is in place to keep that problem continually in check, you can repeat the formula with the next most important performance problem.

We'll use this methodology to confidently tune our catalog service throughout the remainder of this chapter. In fact, we've already taken the first steps toward our goal. We wrote clean and modular code to make our test pass. Demonstrating our results to our customer prompted them to draft a realistic performance requirement we can measure. Before beginning to hack and slash, we used a profiler to find high-yield tuning opportunities. Now we're ready to begin tuning so that we can deliver on the goal our customer has given us. First, let's make sure we get started on the right foot by avoiding the tedium of manual testing.

9.6 *Mini-antipattern: Manual Performance Testing*

When we first turned over our catalog service to our customer, we watched as they poked and prodded, and we noticed that they grew weary of manually hitting the web page to assess performance. A couple times, they had to redo tests because the manual process wasn't followed consistently. Clearly, we need a more efficient and less error-prone testing strategy. As the suite of performance tests grows, running them all manually just doesn't scale. So many tests to run, so little time. Our team has earned a reputation for cranking out high-quality features like clockwork. To live up to that great reputation, we can't drag ourselves to the test lab to play the role of simulated web users every time we change something that impacts performance. That's what computers are for!

When push comes to shove and deadlines loom, manual tests are always the second thing (right after documentation, of course) that gets selectively ignored. Peril usually isn't too far behind. The next time we tune something without the safety of automated tests, our confidence will wane, and we'll fall back into a Thrash-tuning cycle again. Our stress goes up, the number of defects increases, and pretty soon we're burnt out.

9.6.1 Solution: Automate performance testing

Automated performance tests are like canaries in a coal mine. If we keep them running, they'll continue to measure whether performance goals are being met in the face of change. If a change causes performance to backslide, well, we'll know it at the poor birdie's expense.

Good performance tests offer many advantages, including

- automatically checking their own results
- providing immediate feedback in the form of a simple pass or fail status
- retaining their value over time through repeated testing of expectations
- running continuously without manual intervention
- instilling confidence to change code with impunity

At the least, we should run our performance tests once a day as a sanity check. If we're actively tuning code or changing the runtime environment, we should run them more often. The repeatability of the tests will prove our application's readiness for production.

A plethora of tools already exists for performance testing automation. Apache JMeter,[2] for example, is an open source desktop application that measures the performance of an application's behavior under load. Traditionally used to test web applications, over time JMeter has been made more extensible. You can now write custom extensions to put almost any server, network, or object under load. JMeter is also highly configurable; it includes a collection of test listeners for graphically visualizing performance. It can also be configured to include test assertions. For example, you can assert that a request for a web page returns within a specified amount of time. Yet, although JMeter is a valuable tool for performance testing, it falls short of our goals for automation. Specifically, the test results must be manually inspected each time the tests are run. While we could

2 http://jakarta.apache.org/jmeter/

probably extend JMeter to satisfy our desire for automation, instead we opt to use a complementary tool.

In this chapter we'll use JUnitPerf, an open source performance-testing tool that wraps existing functional tests written in JUnit. We choose JUnitPerf because it allows us to write tests that automatically check their own results and provide immediate feedback with an unambiguous pass or fail status. JUnitPerf is also tightly integrated with JUnit, so our performance tests can be run automatically alongside our functional JUnit tests.

9.7 Automated performance testing with JUnitPerf

We've already validated the feasibility of our EJB design with working code and a functional test. Making it fast enough to please our customer is our next order of business. However, we don't want to risk breaking functionality by complicating our code with performance optimizations. To be useful, the catalog service has to work right and perform well at the same time. Let's explore how JUnitPerf can keep both these interests in check.

9.7.1 JUnitPerf overview

JUnitPerf[3] is an open source set of JUnit extensions for automated performance testing. JUnitPerf tests transparently wrap standard JUnit tests and measure their performance. In other words, we can build upon our existing functional test to make sure the code continues to work right. The JUnitPerf tests tell us if the code is fast enough. If a performance test doesn't meet expectations, the whole test fails. If the functional test fails, the performance test fails. Conversely, if the performance test passes, then we have confidence that tuning didn't cause existing functionality to break. Table 9.3 describes the major JUnitPerf classes and interfaces.

Because JUnitPerf tests can run any class that implements JUnit's `Test` interface, we could use JUnitPerf to measure the performance of any test conforming to this interface. In this section, we use it to wrap the JUnit test we wrote earlier for our catalog service. We could also use JUnitPerf to run HttpUnit tests and measure the performance of our entire web application, for example. Another option might be to use JUnitPerf to run Cactus tests to validate our catalog service's business logic from within the EJB container.

3 http://www.clarkware.com/software/JUnitPerf.html

Table 9.3 JUnitPerf is a collection of classes and interfaces for performance testing JUnit tests.

Class/Interface	Description
`TimedTest`	Runs a JUnit test and measures its elapsed time. A `TimedTest` is constructed with a specified maximum elapsed time. By default, a `TimedTest` will wait for the completion of its JUnit test and then fail if the maximum elapsed time was exceeded. Alternatively, a `TimedTest` can be constructed to immediately fail when the maximum elapsed time of its JUnit test is exceeded.
`LoadTest`	Runs a JUnit test with a simulated number of concurrent users and iterations. The load can be incrementally ramped by registering a `Timer` instance to control the delay between the additions of each concurrent user.
`Timer`	An interface implemented by classes that define timing strategies to optionally control the delay between additions of users in a `LoadTest`.
`ConstantTimer`	A `Timer` with a constant delay.
`RandomTimer`	A `Timer` with a random delay and a uniformly distributed variation.

But that's another day. Right now, our customer is getting nervous. Let's put our money where our mouth is with an automated response time test for the catalog service.

9.7.2 Testing response time

Recall that we're staring down the barrel of a response time of approximately 1.4 seconds to display 25 products on a web page. We won't sleep well until the response time is under 1 second. Luckily, we know what to do. The code profiler indicated earlier that optimizing the `CatalogEJB.getProductsByCategory()` method would be the smart move. How will we know when we're done? Well, when a performance test passes, of course.

We want to write a test that will fail if the response time of our use case exceeds 1 second. To do that, we create a JUnitPerf `TimedTest` instance that wraps our existing `CatalogTest.testGetProducts()` test case method. Listing 9.4 shows the JUnitPerf test used to validate our performance expectations.

Listing 9.4 Testing the response time of the catalog service

```
public class CatalogResponseTimeTest {

  public static Test suite() {

    long maxTimeInMillis = 1000;

    Test test = new CatalogTest("testGetProducts");
    Test timedTest = new TimedTest(test, maxTimeInMillis);
    return timedTest;
```

```
  }

  public static void main(String args[]) {
    junit.textui.TestRunner.run(suite());
  }
}
```

As a convenient way to run our test, our test defines a `suite()` method called by the JUnit test runner in the `main()` method. We run the `CatalogResponseTime-Test`, and it fails with the following output:

```
.TimedTest (WAITING):
  testGetProducts(com.bitterejb.catalog.ejb.CatalogTest): 1352 ms
F
Time: 1.352
There was 1 failure:
1)testGetProducts(com.bitterejb.catalog.ejb.CatalogTest)
Maximum elapsed time exceeded! Expected 1000ms, but was 1352ms.

FAILURES!!!
Tests run: 1,  Failures: 1,  Errors: 0
```

All right, we knew that would happen. We just wanted to see if the test was really measuring anything. The test expected the response time to be less than 1 second, but sure enough, it measured the same response time as observed by our customer—1.4 seconds. Now we have a solid baseline from which to work. We should be able to optimize code, improving the response time until the test passes. If the test doesn't eventually pass, we'll need to start turning the performance knob in the other direction or look for another knob to turn.

Be aware of a subtle "gotcha!" when writing JUnitPerf tests. The response time measured by a `TimedTest` includes the elapsed time of the `testXXX()` method and its test fixture—the `setUp()` and `tearDown()` methods. Therefore, the maximum elapsed time specified in the `TimedTest` should be adjusted accordingly to take into account any cost of the existing test's fixture.

9.7.3 Tweaking code

To get the test to pass, we follow the code profiler's advice and optimize the logic that created `ProductDetails` objects from the rows in our database. The SQL query is sufficiently fast, according to the profiler, so nothing is gained barking up that tree. After optimizing, we run the `CatalogResponseTimeTest` again, and it gives us the following output:

```
.
```

```
TimedTest (WAITING):
testGetProducts(com.bitterejb.catalog.ejb.CatalogTest): 751 ms

Time: 0.751

OK (1 test)
```

Hey, that did the trick! The test tells us that we made good progress. Had the test failed, we could have continued to optimize until it passed.

After showing off the improved catalog service to our customer, we add this test to our suite of performance tests. As we go forward, the automated test will continue to keep the response time of this use case in check.

9.7.4 Specifying scalability as a measure of success

At this point we know how long it takes for one user to get a list of 25 products using the catalog service. How long will the same process take if our application is under the stress of multiple concurrent users? Until now we haven't thought much about scalability, but we're confident that our simple design won't let us down. Here's where the rubber meets the road.

Our customer is impressed with our track record of meeting goals, and now is ready to hand us a new challenge. Performance planning early and often has enabled the customer to accurately estimate the expected load on the production system. The customer now wants to improve upon our performance success by writing a new performance requirement, which states:

The response time of the catalog web page listing up to 25 products should not exceed 1 second under a load of five concurrent users.

In other words, the catalog service should scale to handle five concurrent users while consistently maintaining the single-user response time we demonstrated earlier. That's a tall order. Let's use JUnitPerf to see how far off the mark our application currently is.

9.7.5 Testing response time under load

We have an automated JUnitPerf test that measures single-user response time. We'd like to use a similar testing technique to put this test under a load of five concurrent users while measuring each user's response time. We want the test to fail if any user's response time exceeds one second. Then we can follow our performance testing methodology to tune until the test passes.

To do so, we write a JUnitPerf test that creates a `LoadTest` instance passing in a `TimedTest` instance and a number of concurrent users. The `TimedTest` in turn

wraps our existing `CatalogTest.testGetProducts()` test case method. Listing 9.5 shows the JUnitPerf test used to validate our scalability expectations.

Listing 9.5 Testing the scalability of the catalog service

```
public class CatalogLoadTest {

  public static Test suite() {

    long maxTimeInMillis = 1000;
    int concurrentUsers = 5;

    Test test = new CatalogTest("testGetProducts");
    Test timedTest = new TimedTest(test, maxTimeInMillis);
    Test loadTest = new LoadTest(timedTest, concurrentUsers);
    return loadTest;
  }

  public static void main(String args[]) {
    junit.textui.TestRunner.run(suite());
  }
}
```

We run the `CatalogLoadTest` , which fails with the following output:

```
..........
TimedTest (WAITING):
testGetProducts(com.bitterejb.catalog.ejb.CatalogTest): 771 ms
TimedTest (WAITING): testGetProducts(com.bitterejb.catalog.ejb.CatalogTest):
   1372 ms
F
TimedTest (WAITING): testGetProducts(com.bitterejb.catalog.ejb.CatalogTest):
   1963 ms
F
TimedTest (WAITING): testGetProducts(com.bitterejb.catalog.ejb.CatalogTest):
   2584 ms
F
TimedTest (WAITING): testGetProducts(com.bitterejb.catalog.ejb.CatalogTest):
   3255 ms
F
Time: 3.40
There were 4 failures:

1)testGetProducts(com.bitterejb.catalog.ejb.CatalogTest)
  Maximum elapsed time exceeded! Expected 1000ms, but was 1372ms.
2)testGetProducts(com.bitterejb.catalog.ejb.CatalogTest)
  Maximum elapsed time exceeded! Expected 1000ms, but was 1963ms.
3)testGetProducts(com.bitterejb.catalog.ejb.CatalogTest)
  Maximum elapsed time exceeded! Expected 1000ms, but was 2584ms.
4)testGetProducts(com.bitterejb.catalog.ejb.CatalogTest)
  Maximum elapsed time exceeded! Expected 1000ms, but was 3255ms.
```

```
. . .
FAILURES!!!
Tests run: 5,  Failures: 4,  Errors: 0
```

Ouch! Our application can't scale beyond one user. Notice that the first user's response time is within the 1-second limit, but the other users' response times bust the threshold. Worse yet, the response times increased for each successive user, indicating that our application has a bottleneck restricting its ability to scale.

So we fire up the code profiler and run the `CatalogLoadTest` to obtain clues. The code profiler doesn't let us down. Table 9.4 shows what the profiler finds when the catalog service is under load.

Table 9.4 Running a profiler on the catalog service under load reveals contention for a database connection. Load testing tools help illuminate scalability bottlenecks.

Method	Average time (ms)
`CatalogEJB.getProductsByCategory()`	716.0
`CatalogServlet.service()`	10.0
`CatalogEJB.getConnection()`	1248.0
`CatalogEJB.runQuery()`	15.0
	1989.0

The `CatalogEJB.getConnection()` method that was only taking around 10 milliseconds in our initial run of the code profiler is now taking up the majority of the overall response time. Let's tune that method while continuing to test the single-user response time.

9.7.6 *Using a connection pool to increase throughput*

Based on the evidence provided by the code profiler, we conclude that a single database connection is the limiting factor to scaling our application. Consequently, requests for a connection are being queued. Each successive user's response time in turn rises above the desired threshold.

In this case, pooling database connections is a low cost, high reward change sure to improve scalability. Before you start rolling your eyes over yet another example demonstrating the virtues of connection pooling, allow us to explain. We realize connection pooling is the poster child for many discussions on performance. Indeed, it's a well-known performance problem, and that's exactly why we're using it here. We want the problem—and the solution—to be familiar so

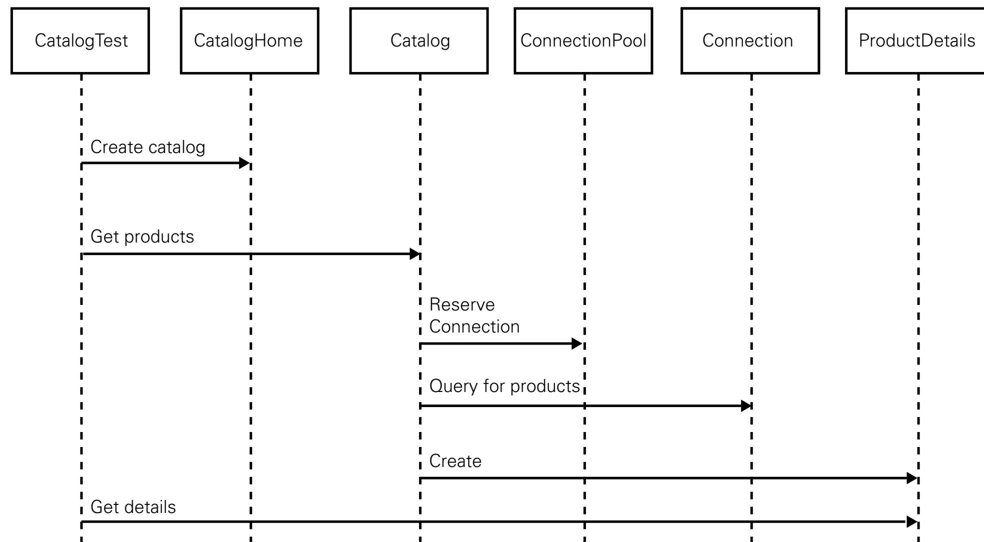

Figure 9.4 Refactoring the catalog service to use a database connection pool improves the scalability without sacrificing code complexity.

you can focus on how to test it. It's not about the connection pool; it's about the technique to discover it.

It's also worth noting that we've run across more than one improperly sized connection pool. Worse yet, we've seen custom connection pools that were implemented incorrectly. (Yet another victim of the Not Invented Here antipattern.) In other words, just because you're using a connection pool doesn't necessarily mean you get instant scalability. You have to test for that, so let's get back to the technique.

Instead of synchronizing access to a single database connection shared by multiple users, we refactor our `Catalog` EJB to use a database connection pool. We then configure the pool size to five active connections to improve scalability. Figure 9.4 shows a UML sequence diagram illustrating the use of the database connection pool.

Now we run the `CatalogLoadTest` again, and it passes with the following output:

```
.....
TimedTest (WAITING): testGetProducts(com.bitterejb.catalog.ejb.CatalogTest):
   751 ms
TimedTest (WAITING): testGetProducts(com.bitterejb.catalog.ejb.CatalogTest):
   812 ms
```

```
TimedTest (WAITING): testGetProducts(com.bitterejb.catalog.ejb.CatalogTest):
   822 ms
TimedTest (WAITING): testGetProducts(com.bitterejb.catalog.ejb.CatalogTest):
   831 ms
TimedTest (WAITING): testGetProducts(com.bitterejb.catalog.ejb.CatalogTest):
   811 ms

Time: 0.972

OK (5 tests)
```

Outstanding! Our scalability test is passing, and the underlying functional test continues to pass. This tells us that refactoring to use a database connection pool didn't break anything. As we expected, the refactoring actually improved scalability. Because requests don't need to be queued before being serviced, the response times are fairly consistent for each concurrent user. The test validates our design as able to handle five concurrent users without any specific user experiencing a delayed response time. If, in the future, the response time of any user increases beyond the limit set in our load test, the test will fail.

9.7.7 *Testing throughput*

We may end up with performance requirements expressed as throughput rather than as response time under load. For example, we might want to write an automated test to measure the total amount of time elapsed while servicing all five concurrent users. Using JUnitPerf, we simply reverse the order in which we create the tests, this time wrapping the `LoadTest` in a `TimedTest`, as indicated in listing 9.6.

Listing 9.6 Testing the throughput of the catalog service

```
public class CatalogThroughputTest {

  public static Test suite() {

    long maxTimeInMillis = 1000;
    int concurrentUsers = 5;

    Test test = new CatalogTest("testGetProducts");
    Test loadTest = new LoadTest(test, concurrentUsers);
    Test timedTest = new TimedTest(loadTest, maxTimeInMillis);
    return timedTest;
  }

  public static void main(String args[]) {
    junit.textui.TestRunner.run(suite());
  }
}
```

The `CatalogThroughputTest` will fail if the catalog service is unable to process at least five catalog queries per second. After refactoring to use a database connection pool, the `CatalogThroughputTest` passes with the following output:

```
.....
TimedTest (WAITING): LoadTest (NON-ATOMIC): ThreadedTest:
  testGetProducts(com.bitterejb.catalog.ejb.CatalogTest)(repeated): 972 ms

Time: 0.972

OK (5 tests)
```

Now that we've written JUnitPerf tests to measure both response time and throughput, let's put all the numbers together into a performance model.

9.8 Modeling performance

Using our existing JUnitPerf tests, we can ramp up the user load to sketch out a model that represents our application's overall performance. Doing so will answer questions in the performance planning process like, "Will our application scale to meet the demands of 10, 100, or 1,000 concurrent users?"

As an example, figure 9.5 shows the average response time as a function of the number of concurrent users. The figure example compares the use of a database connection pool with 10 active connections to that of a single shared database connection.

Notice that with a single database connection the application cannot maintain a linear response time as the number of concurrent requests increases. That is, as more users attempt to use the application, their observed response times are elongated. In contrast, using a database connection pool allows the application to service requests to up to 25 users at a relatively constant response rate.

Figure 9.6 shows the throughput as a function of the number of concurrent users. This figure also compares the use of a database connection pool with 10 active connections to that of a single shared database connection.

Notice that, regardless of the number of concurrent users, the bottleneck caused by a single database connection limits the throughput to one catalog query per second—the application's maximum effective throughput. In contrast, by configuring the connection pool with 10 active connections, the application is able to consistently process almost 10 catalog queries per second. The application can scale to handle at least 25 concurrent users with only 10 shared connections.

Models such as these are great information radiators. You can look at them quickly and know how your application performs. Many performance testing

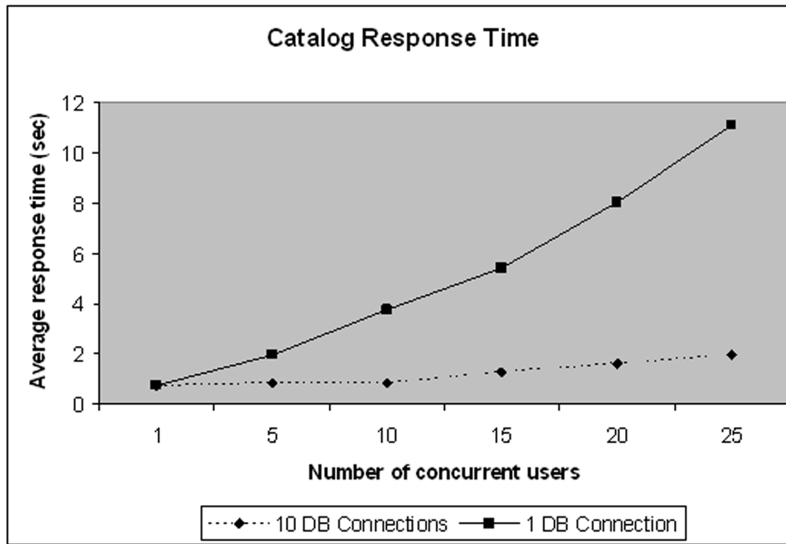

Figure 9.5 The use of a database connection pool, as indicated by the dotted line, yields a fairly constant response time for up to 25 concurrent users. With a single shared connection, as indicated by the solid line, the response time curve is exponential.

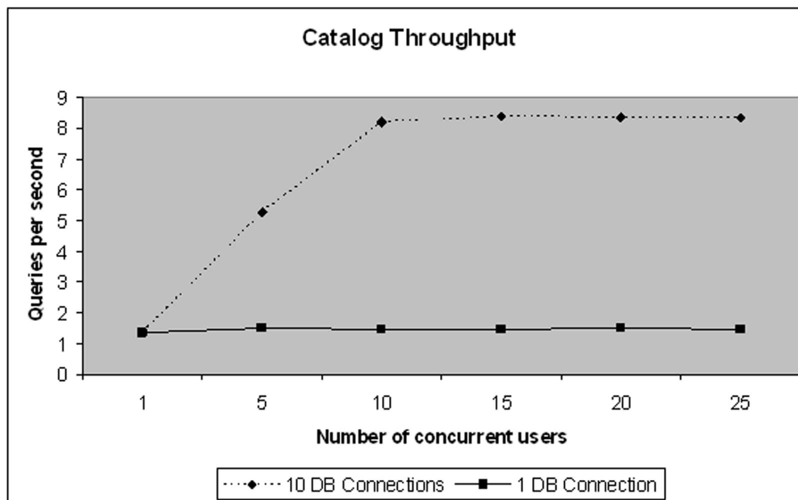

Figure 9.6 The use of a database connection pool, as indicated by the dotted line, delivers a throughput roughly equivalent to the number of active connections in the pool. With a single shared connection, as indicated by the solid line, the throughput bottoms out at one query per second.

tools, including JMeter, will automatically generate charts of this sort. Use them to your advantage.

9.9 *Mini-antipattern: Stage Fright*

If we don't test our application's performance early and often in a production-like environment, when the curtain goes up, the application may fall down in front of its live audience. Often we assume that, when an application meets its performance goals in development, it will perform equally well for its intended audience. We're usually disappointed.

To simulate realistic production traffic and usage patterns, we need to test our application's performance with representative data, tool versions, workloads, network latency, and hardware capacity. Tests that merely simulate users continually buying Chihuahuas from our online pet store won't cut it. We'll be in for an unwelcome surprise when a real user tries to buy a furry friend not already cached in the middle tier.

9.9.1 *Solution: Practice on stage*

To alleviate the fear and risk of embarrassment on stage, practice is our only recourse. Running performance tests in a production-staging environment as soon as possible, and keeping those tests running, will give us the confidence we need. The best approach to practicing for a production setting is to write tests that address our worst fears. What will happen when 10 users log in at the same time? We won't know until it happens, but we do know it's better to have it happen when we're practicing. Writing a passing login test under a 10-user load goes a long way toward boosting our confidence for the big show. In other words, tests let us safely play "what if" games with performance. By simulating a load, they can help us determine the amount of hardware we'll need to support our expected user load.

We need to give our application a dress rehearsal by testing under realistic scenarios. We'll use the same version of the virtual machine, application server, database, and other tools that will be deployed in the production environment. If caching and pooling is used to boost performance, then we can let the application warm up before running the performance tests. In other words, our tests should measure the actual performance, as observed by real users, to the maximum extent possible.

Once we've done our best to design and tune for performance under realistic loads, no substitute exists for tuning an application in production. Don't

underestimate the value of a tool that can monitor a live performance. Usage patterns in a live system may behave differently than expected.

9.10 Summary: Tuning with confidence

In this chapter, we looked at several antipatterns that commonly plague the EJB performance tuning process. In our quiet moments, when we're sure nobody is listening, we've probably all admitted to ourselves that we've been bitten by the need for speed. However, now that we've resolved to put speed in its place, we want to avoid these antipatterns by adopting an approach that's best summarized in the carpenter's motto: measure twice, cut once.

Before tuning to improve performance, we profiled our code to find hot spots. We then measured the performance again with a failing performance test written using JUnitPerf. Only after reviewing this evidence did we attempt to change code or the runtime environment to improve performance. We also put our trust in our gauges—automated tests that specify the performance requirements set by our customer. We leveraged these tests, using them as the qualifying measure of success, to ensure that our application's performance continually improved.

The methodology proposed for side-stepping the pitfalls encountered in this chapter is applicable to any type of performance tuning activity—EJB or otherwise. The bottom line: When making a case for or against applying changes in the name of performance, don't assume facts not already in evidence. Test first, then tune with confidence.

9.11 Antipatterns in this chapter

This section covers the Premature Optimization, Performance Afterthoughts, Thrash-tuning, Manual Performance Testing, and Stage Fright antipatterns.

PREMATURE OPTIMIZATION

DESCRIPTION

Good programmers frequently try to optimize every line of code and speculate in the name of performance, without considering which code/design elements are actually performance problems.

MOST FREQUENT SCALE

Application

REFACTORED SOLUTION NAME

Performance test automation

REFACTORED SOLUTION TYPE

Process, technology

REFACTORED SOLUTION DESCRIPTION

Use the simplest code/design that will work. Establish concrete criteria and run automated performance tests against the criteria to establish the need for performance tuning. Tune only problem areas. Write well-factored and modular code that's easy to tune later, if necessary.

ANECDOTAL EVIDENCE

"It works fine, but I suspected future performance problems so I spent the afternoon making it fast." "All of my code is a little tough to read, but it's very fast." "That design/technology is going to be too slow."

SYMPTOMS, CONSEQUENCES

Fewer development cycles are left for customer requirements or meaningful optimization when unforeseen problems arise. Design and code becomes unnecessarily complex and difficult to maintain. Functionality breaks when it's tweaked to be faster.

PERFORMANCE AFTERTHOUGHTS

DESCRIPTION

Attempting to bolt performance on to an application at the end of the development cycle rather than bake it in from the beginning

MOST FREQUENT SCALE

Application

REFACTORED SOLUTION NAME

Continuous performance planning

REFACTORED SOLUTION TYPE

Process

REFACTORED SOLUTION DESCRIPTION

Gather performance requirements early and often. Build automated performance tests that continuously validate performance criteria. Performance tests help to define exactly which areas do not meet criteria to focus testing efforts. Make any necessary course corrections throughout the project based on quantifiable measurements.

TYPICAL CAUSES

Poor planning

ANECDOTAL EVIDENCE

"We will have plenty of time to performance tune at the end of the development cycle." "It's a good design. We do not need to tune for performance." "We'll let our QA department measure performance." "We're using Enterprise JavaBeans, so it should scale well."

SYMPTOMS, CONSEQUENCES

Repeated delivery of poorly performing software, redesign of critical use cases late in the development cycle, and last-minute tuning activities that are ineffective

THRASH-TUNING

DESCRIPTION

Performance tuning is difficult without a solid baseline or when multiple configuration parameters are changed at once between measurements. Attempting performance tuning in these conditions makes it difficult to gauge progress and correct problems, lengthening the overall cycle time and giving the appearance of thrashing.

MOST FREQUENT SCALE

Application

REFACTORED SOLUTION NAME

Good performance methodology

REFACTORED SOLUTION TYPE

Process

REFACTORED SOLUTION DESCRIPTION

A sound performance testing methodology and a good testing environment are the primary keys. Baseline measurements are mandatory to gauge progress. All tests should start from a common configuration and changes should be made one at a time. Focus on performance problems demonstrated by failed tests.

ROOT CAUSES

Haste, inadequate performance testing tools

ANECDOTAL EVIDENCE

"It feels faster, don't you think?" "When are we done tuning?" "What did we change to make it slower?"

SYMPTOMS, CONSEQUENCES

Inefficient performance testing and tuning, longer than expected performance tuning cycles, and unclear results of performance improvements

MANUAL PERFORMANCE TESTING

DESCRIPTION

Manually running performance tests every time something is changed doesn't scale and the tests aren't easily repeatable

MOST FREQUENT SCALE

Organization

REFACTORED SOLUTION NAME

Automated performance testing

REFACTORED SOLUTION TYPE

Process, technology

REFACTORED SOLUTION DESCRIPTION

Use a performance testing tool like JUnitPerf to build automated tests. Let a computer run the tests continuously and consistently.

TYPICAL CAUSES

Inadequate performance testing tools

ANECDOTAL EVIDENCE

"I don't have time to run that test." "I can't repeat the results of the test from run to run."

SYMPTOMS, CONSEQUENCES

The time it takes to run tests manually increases the pressure to fall back into a thrash-tuning cycle. Performance problems aren't detected consistently.

STAGE FRIGHT

DESCRIPTION

Failure to test software in its production environment with representative data, tool versions, workloads, network latency, and hardware capacity

MOST FREQUENT SCALE

Application

REFACTORED SOLUTION NAME

Production environment testing

REFACTORED SOLUTION TYPE

Process

REFACTORED SOLUTION DESCRIPTION

Test application performance in settings as close as possible to the production environment.

TYPICAL CAUSES

Pride, ignorance

ANECDOTAL EVIDENCE

"We don't have time to test in production. The system is going live tomorrow!" "Don't worry. This is good code. It should work fine in the production environment." "It was fast on my development machine."

SYMPTOMS, CONSEQUENCES

Software that performs well in development environments, but fails miserably in production settings