Proven Solutions to Intractable Problems

J2EE™ Design Patterns

O'REILLY®

William C. R. Crawford & Jonathan Kaplan

J2EE™ Design Patterns

Other Java™ resources from O'Reilly

Related titles

- Enterprise JavaBeans™
- Java™ & XML
- Java™ Cookbook
- Java™ Enterprise in a Nutshell
- Java™ I/O
- Java™ in a Nutshell
- Java™ Performance Tuning
- Java™ Programming with Oracle SQLJ
- Java™ Security
- JavaServer™ Pages
- Java™ Swing
- Learning Java™

Java Books Resource Center

java.oreilly.com is a complete catalog of O'Reilly's books on Java and related technologies, including sample chapters and code examples.

OnJava.com is a one-stop resource for enterprise Java developers, featuring news, code recipes, interviews, weblogs, and more.

Conferences

O'Reilly & Associates bring diverse innovators together to nurture the ideas that spark revolutionary industries. We specialize in documenting the latest tools and systems, translating the innovator's knowledge into useful skills for those in the trenches. Visit *conferences.oreilly.com* for our upcoming events.

Safari Bookshelf (*safari.oreilly.com*) is the premier online reference library for programmers and IT professionals. Conduct searches across more than 1,000 books. Subscribers can zero in on answers to time-critical questions in a matter of seconds. Read the books on your Bookshelf from cover to cover or simply flip to the page you need. Try it today with a free trial.

J2EE™ Design Patterns

William Crawford and Jonathan Kaplan

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

CHAPTER 5

Presentation Tier Scalability

Many developers believe design patterns and scalability do not go hand in hand. They argue that patterns add layers to an application, so the server must perform more operations and use more memory to handle each request. The extra operations slow response time down, and the increase in memory means fewer clients can be supported per server. By itself, this is a fair assessment, and if no two requests were alike, it might be the end of the story.

In an enterprise application, however, many clients need to access similar data. On a site that publishes stock quotes, for example, the server may respond to thousands of requests a minute for the same stock. If the price of the stock changes every five minutes, it would be massively inefficient to contact the stock market for each request. Even in an online bank, where every user wants to view personal data, resources such as database connections do not need to be recreated for every request.

Often, we can sacrifice some speed up front for better performance in the average case. While the first request for a particular stock quote or the first connection to a particular database might require a lot of work, subsequent requests will be much faster. It is fair to say the system's scalability will increase: we can support more requests in the same amount of time.

In this chapter, we look at three patterns that increase the scalability of the presentation tier using variations of this concept:

Asynchronous Page
: Shows how to cache data, such as stock prices, and use it to generate dynamic pages.

Caching Filter
: Can be used to cache entire dynamic pages after they are generated.

Resource Pool
: Describes how to create a "pool" of large or expensive objects that can be loaned out as needed, saving instantiation costs.

Scalability and Bottlenecks

Before we jump into the patterns, let's take a minute to discuss what we mean by a *scalable* system. Think of a web-based system as a request processor. Requests come in from the clients, and the clients wait until results are generated. Everything in between—whether it's simply returning the contents of a static file or generating a fully dynamic page—is the actual processing.

For a request processor, scalability is related to the number of requests that can be processed simultaneously. In a simple sense, scalability might be the ability to "survive" a certain number of hits at the same time, eventually delivering a proper response to each one, but we know from experience that this is not really the case. If a news site gets 10,000 simultaneous hits and responds to each of them within 3 seconds, we might say the site scales adequately, if not exceptionally. But if the same site gets 100,000 simultaneous hits, responding to each one within three minutes would not be acceptable.*

A better definition of scalability is a system's ability to grow in order to handle increased demand. Obviously, no single server can be expected to handle an infinite number of requests. In a scalable system, you have options when a single server has reached its maximum capacity. In general, you can:

- Buy a faster server
- Buy more servers

While it may seem obvious that a faster server can handle more requests, it is not always the case. Imagine a bank that stores its total assets in a single record, which must be updated every time money is deposited or withdrawn. If the record can only be updated by one request at a time, the maximum number of transactions will be limited by the time it takes to write this record. Increasing the speed of the server's CPUs might help a little, since the asset total could possibly be updated faster. But the overhead of allowing multiple CPUs to communicate—more processes contending for access to the single resource—could mean that adding CPUs actually decreases overall speed! A single point that limits the scalability of the entire application (like our total asset record) is known as a *bottleneck*.

Potential bottlenecks multiply when you start using multiple servers. Imagine a distributed version of our banking application, with a total asset counter on its own dedicated server. Each request processor will send a message to this server every time money is deposited or withdrawn. Clearly, scalability of this system is still limited by the time it takes to update the total asset counter. But the network between the

* For developers, acceptable scalability might be related to what's going on behind the scenes: a dynamic page won't be expected to return as quickly as a static page. Users don't share this nuanced view.

request processors and account server might also limit scalability, as well as the time it takes to translate data to and from a format that can be sent over the network.

In practice, scalability requires tradeoffs. One such tradeoff is between a few large systems and many small systems: many smaller systems are usually cheaper, but the communication can limit scalability. There is also a tradeoff between caching and memory use. Using a large cache may mean that more requests can be served from the cache, and the system therefore will be much faster. However, it also means the memory for the rest of the system is limited, so requests that are not served from the cache may be even slower. The art of building a scalable application is in eliminating unnecessary bottlenecks while balancing code complexity and resource allocation.

Content Caching

One of the best ways to increase scalability is through caching. Caching static pages is a well established practice on the web, and dedicated web caches are taking their place with routers and switches as standard equipment in the data center. A web cache maintains copies of the most frequently requested files, which allows it to serve those requests quickly, without involving the real web server. Most caches rightly ignore dynamically generated pages, as there's no way to efficiently determine whether any two requests should receive the same response (or, for that matter, whether the web server needs to process the request for other reasons).

To create a high performance J2EE environment, we need to bring the benefits of caching to the dynamic components of the presentation tier. In this section, we'll look at two approaches to this problem: caching the input used to generate the dynamic content, and caching the generated content itself.

Caching Content Components

The traditional communication model for web applications is *synchronous*. In other words, clients make requests for URLs and wait until they receive a web page in response. Given the nature of this exchange, it is easy to see why most application logic is implemented in a synchronous manner. As requests come in, the various parts of the response are calculated and correlated, and then the final response is generated. The problem with this approach is that it can be quite inefficient.

In Chapter 4 we discussed a method of adding remote news feeds to a page by building a simple RSS parsing mechanism. Our approach was synchronous: when a request came in for a page containing a news feed, a JSP custom tag read and parsed the remote data and then formatted it for display. If we scale this approach up to the point where we have multiple servers each talking to multiple feeds, we might end up with something like the situation shown in Figure 5-1.

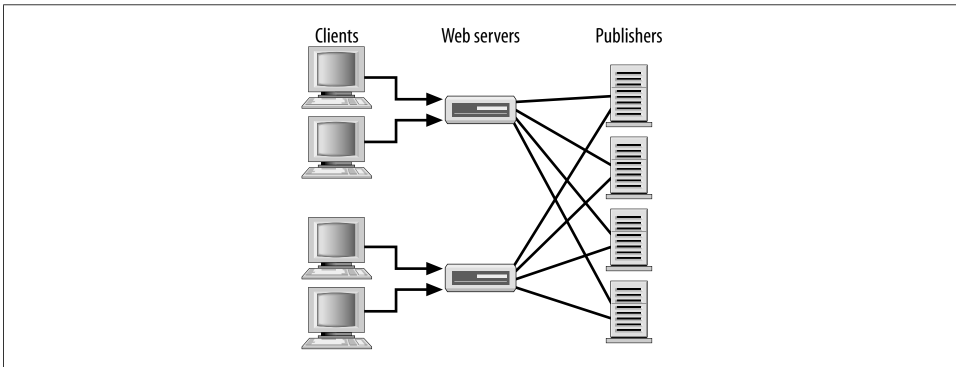


Figure 5-1. Reading multiple news feeds

This approach is inefficient for a number of reasons. Contacting every publisher on every request wastes bandwidth and saturates the publishers' servers. Requests would also be quite expensive in computer resources, since the feed needs to be parsed and translated into HTML on every request. Caching the data on each server should significantly increase the number of clients that can be supported.

While caching represents a significant win for scalability, it does not exploit the fact that news feeds only update intermittently. Having a cache requires a caching policy; for instance, the data could be updated on every tenth request, or every ten minutes. Keeping the data up-to-date still requires each web server to contact the publisher frequently, wasting bandwidth and potentially CPU time. It also means rereading and reparsing the data separately on each server. A better solution would be one where the data was only transmitted when it changed. Figure 5-2 shows the same system using a publish-subscribe model. A single machine, the application server, subscribes to a number of news feeds. When new data becomes available from a particular feed, the data is parsed and sent to all the individual web servers.* Because the data is only sent from the publisher to the subscriber as needed, we say this solution is *asynchronous*. Often, an asynchronous solution requires far less bandwidth than a synchronous one, since data is only sent to the many web servers as needed.

The Asynchronous Page Pattern

The benefits of asynchronous communication are not new. Messaging in particular has been a major component of enterprise backends for a long time. The Java Message APIs and the recent addition of message-driven JavaBeans have solidified the place of asynchronous communication in the Java enterprise architecture. While no standard is currently in place, asynchronous web services are starting to crop up. However, with the exception of a few "push-based" (another term for publish-subscribe) content

* We'll discuss publish-subscribe in other contexts and in more detail in Chapter 11.

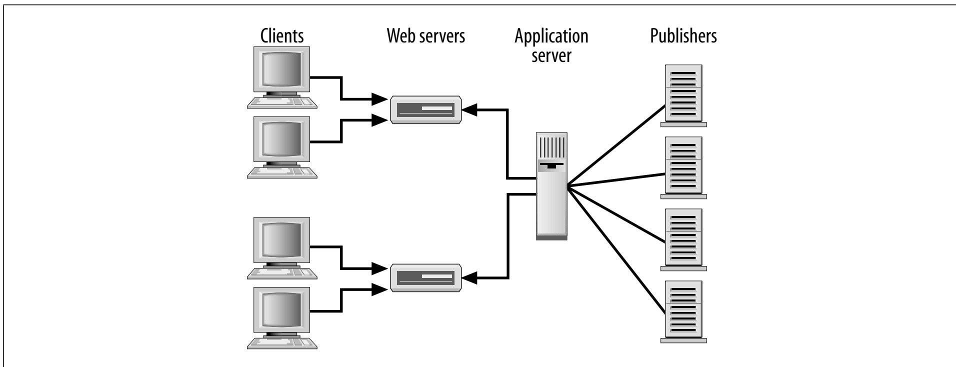


Figure 5-2. A publish-subscribe model

providers, asynchronous communication has never taken off at the client tier because the standard clients—web browsers—do not support publish-subscribe systems.

The lack of browser support for push-based systems does not mean asynchronous communication has no place in a web-based world. It can still be a powerful tool for improving scalability by reducing the work required to handle each transaction.

The *Asynchronous Page pattern* takes advantage of asynchronous retrieval of remote resources to improve scalability and performance. Rather than waiting for a request for a stock quote, for example, a server may accept all stock quotes as they are generated. When a request comes in for a particular quote, the server simply replies with the data it has already received.

Figure 5-3 shows the interactions in the Asynchronous Page pattern. In general, there is a single subscriber that listens to feeds from a number of publishers. As data is updated, the subscriber updates the models of the dependent web applications. When requests come in, the responses incorporate the latest values that have been published.

It's important to note that the interface between the publisher and the subscriber does not need to be push-based. Although it is ideal for the publisher to directly notify the subscriber when there are changes to the data, it is also reasonable to have the subscriber poll the publisher at regular intervals.

The amount of work involved in updating the model can also vary. In some cases, the raw data read from the publisher may be inserted directly into the model unmodified. The pattern is more beneficial, however, when the subscriber processes the data, reducing the work for all the various models. A common tactic is to remove the dynamic page altogether, replacing it with a static page that is simply rewritten each time data is published.

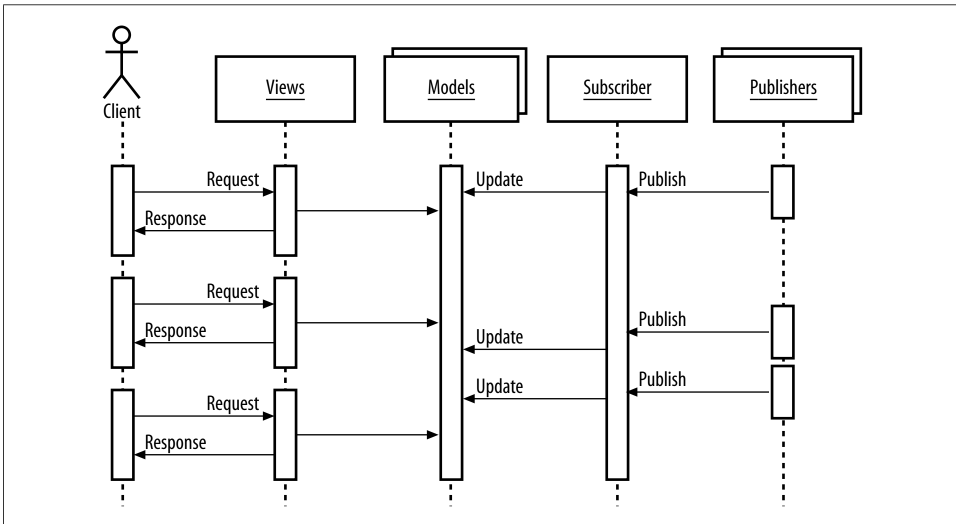


Figure 5-3. Asynchronous Page interactions

Implementing the Asynchronous Page pattern

Let's update our earlier RSS-parsing example to use the Asynchronous Page pattern. Remember, RSS is a standard for sharing news feeds between web sites. It's an XML-based format, and we want to present it as HTML.

Originally, we created a class and two custom tags to flexibly parse RSS. The `RSSInfo` class reads and parses the RSS from a given URL. Based on this class, we created two tags. The first, `RSSChannel`, takes a URL as an argument and reads the remote data. Within the `RSSChannel` tag, two scripting variables store the name of the channel and its link. The `RSSItems` tag may be nested in an `RSSChannel` tag. The `RSSItems` tag iterates through each item in the channel, storing its title and link in scripting variables.

Our problem is that the `RSSChannel` tag is actually reading the data from the remote source. It would be more efficient if the data were stored locally, and only updated as needed. Unfortunately, RSS does not provide a subscription mechanism, so we must poll the remote data. Rather than go through that process at each read, we want a dedicated mechanism to do the polling and update the local copy. The dedicated mechanism will allow us to read the data from a single, central source, and then distribute it to the actual request processors if it is changed.

In this example, we will add a single class, the `RSSSubscriber`. `RSSSubscriber` allows subscriptions to be added to different RSS URLs. Once a subscription is added, a separate thread polls the URL at a specified interval. If the URL is changed, the new

data is added to a local cache. All requests after the original are served from this cache. An implementation of the `RSSSubscriber` class is shown in Example 5-1.

Example 5-1. The RSSSubscriber class

```
import java.util.*;
import java.io.IOException;

public class RSSSubscriber extends Thread {
    private static final int UPDATE_FREQ = 30 * 1000;

    // internal representation of a subscription
    class RSSSubscription implements Comparable {
        private String url;
        private long nextUpdate;
        private long updateFreq;

        // sort based on next update time
        public int compareTo(Object obj) {
            RSSSubscription rObj = (RSSSubscription)obj;
            if (rObj.nextUpdate > this.nextUpdate) {
                return -1;
            } else if (rObj.nextUpdate < this.nextUpdate) {
                return 1;
            } else {
                // if update time is the same, sort on URL
                return url.compareToIgnoreCase(rObj.url);
            }
        }
    }

    // a set of subscriptions, sorted by next update time
    private SortedSet subscriptions;
    private Map cache;
    private boolean quit = false;

    // singelton subscriber
    private static RSSSubscriber subscriber;

    // get a reference to the singleton
    public static RSSSubscriber getInstance( ) {
        if (subscriber == null) {
              subscriber = new RSSSubscriber( );
              subscriber.start( );
        }
        return subscriber;
    }

    RSSSubscriber( ) {
        subscriptions = new TreeSet( );
        cache = Collections.synchronizedMap(new HashMap( ));
        setDaemon(true);
    }
```

Example 5-1. The RSSSubscriber class (continued)

```
    // get an RSSInfo object from cache, or create a new
    // subscription if it's not in the cache
    public RSSInfo getInfo(String url) throws Exception {
        if (cache.containsKey(url)) {
            return (RSSInfo)cache.get(url);
        }

        // add to cache
        RSSInfo rInfo = new RSSInfo( );
        rInfo.parse(url);
        cache.put(url, rInfo);

        // create new subscription
        RSSSubscription newSub = new RSSSubscription( );
        newSub.url = url;
        newSub.updateFreq = UPDATE_FREQ;
        putSubscription(newSub);

        return rInfo;
    }

    // add a subscription
    private synchronized void putSubscription(RSSSubscription subs)
    {
        subs.nextUpdate = System.currentTimeMillis( ) +
                          subs.updateFreq;
        subscriptions.add(subs);
        notify( );
    }

    // wait for next subscription that needs updating
    private synchronized RSSSubscription getSubscription( ) {
        while(!quit) {
            while(subscriptions.size( ) == 0) {
                try { wait( ); } catch(InterruptedException ie) {}
            }

            // get the first subscritpion in the queue
            RSSSubscription nextSub =
               (RSSSubscription)subscriptions.first( );

            // determine if it is time to update yet
            long curTime = System.currentTimeMillis( );
            if(curTime >= nextSub.nextUpdate) {
                subscriptions.remove(nextSub);
                return nextSub;
            }

            // sleep until the next update time
            // this will be interrupted if a subscription is added
            try {
                wait(nextSub.nextUpdate - curTime);
```

Example 5-1. The RSSSubscriber class (continued)

```
            } catch(InterruptedException ie) {}
        }
    }

    // update subscriptions as they become available
    public void run( ) {
        while(!quit) {
            RSSSubscription subs = getSubscription( );

            try {
                RSSInfo rInfo = new RSSInfo( );
                rInfo.parse(subs.url);
                cache.put(subs.url, rInfo);
            } catch(Exception ex) {
                ex.printStackTrace( );
            }

            putSubscription(subs);
        }
    }

    public synchronized void quit( ) {
        quit = true;
        notify( );
    }
}
```

Our new RSS subscription mechanism runs on a single host, but it is easy to see how it could be extended to multiple servers. In any case, it supports more simultaneous requests by having a dedicated thread for updating and parsing requests. Except for the initial request for a given subscription, no thread ever has to block on reading or parsing the remote data. It is effectively done in the background.

To use the `RSSSubscriber`, our custom tag simply calls `getRSSInfo( )` for the URL it is passed. `getRSSInfo( )` will read the data from the cache when possible and create a new subscription when it is not. Example 5-2 shows the custom tag class.

Example 5-2. The RSSChannelTag class

```
import javax.servlet.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class RSSChannelTag extends BodyTagSupport {
    private static final String NAME_ATTR = "channelName";
    private static final String LINK_ATTR = "channelLink";

    private String url;
    private RSSSubscriber rssSubs;

    public RSSChannelTag( ) {
```

Example 5-2. The RSSChannelTag class (continued)

```
        rssSubs = RSSSubscriber.getInstance( );
    }

    public void setURL(String url) {
        this.url = url;
    }

    // get the latest RSSInfo object from the subscriber
    // this method is called by the RSSItems tag as well
    protected RSSInfo getRSSInfo( ) {
        try {
            return rssSubs.getInfo(url);
        } catch(Exception ex) {
            ex.printStackTrace( );
        }

        return null;
    }

    // use the updated RSSInfo object
    public int doStartTag( ) throws JspException {
        try {
            RSSInfo rssInfo = getRSSInfo( );
            pageContext.setAttribute(NAME_ATTR,
                                     rssInfo.getChannelTitle( ));
            pageContext.setAttribute(LINK_ATTR,
                                     rssInfo.getChannelLink( ));
        } catch (Exception ex) {
            throw new JspException("Unable to parse " + url, ex);
        }
        return Tag.EVAL_BODY_INCLUDE;
    }
}
```

Although the RSS reading example is simple, it shows one of the many opportunities for asynchronous communication in a request-driven environment. The opportunity, of course, depends on how the data is accessed. Imagine accepting, parsing, and storing a quote for every stock on the NYSE, and only getting one or two requests for quotes before it is all updated again. The time and memory expended receiving these values asynchronously would be wasted. Of course, in some applications it's not acceptable to have any out-of-date data—like when you're transferring money at an ATM. In evaluating asynchronous methods, it is important to take into account the costs in terms of data staleness, network use, and memory use, as balanced against the benefits of speed and scalability.

Dynamic Content Caching

There is another class of dynamic data that is amenable to caching. Imagine an online car dealership where users go through a few pages selecting various options

and then view the resulting car's price. The price computation could be a lengthy process, possibly accessing an external system that's also used by dealers, or even one that keeps track of dealer stock.

Certain cars and options—the sport package, a sunroof—are far more common than others. Since the same set of options always yields the same price, it's inefficient to recalculate it every time. Even worse for the efficiency of the site is all the overhead of generating the page dynamically: querying prices in the database and assembling multiple views into a composite. We would like to cache the page with the price calculated and the HTML generated.

In a perfect world, the application itself would not have to worry about caching at all. HTTP 1.1 allows caching dynamic GET requests, as long as we set the correct HTTP header fields.* Once these fields are set, the client, an intermediate cache, or even the HTTP server can do the caching. In practice, however, we frequently have to do it ourselves.

The Caching Filter Pattern

The *Caching Filter pattern* uses a servlet filter to cache dynamically generated pages. We talked a fair bit about filters, including their capabilities and implementation, when we talked about the Decorator pattern in Chapter 3. The caching filter is a specific implementation of a decorator. When applied to the front controller, it caches fully generated dynamic pages. The classes in the Caching Filter pattern are shown in Figure 5-4.

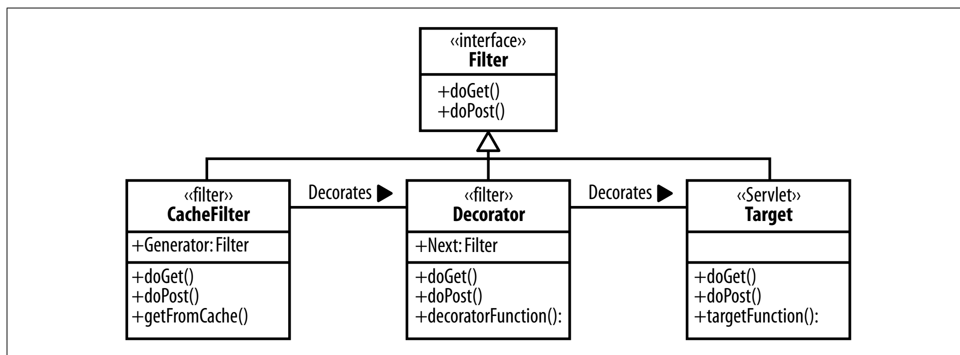


Figure 5-4. Classes in the Caching Filter pattern

The `CacheFilter` itself works just like any other filter: it presents the API of its one child. It also provides extra methods to read pages from the cache as well as to add the results of executing a particular request to the cache. When a request comes in,

* For more information on the variety of caching options available in HTTP 1.1, see section 13 of RFC 2616, available from: *http://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html#sec13*.

the cached page is simply returned if it exists. If the page is not cached, the rest of the chain must be executed and the result stored in the cache. The process of handling a request is shown in Figure 5-5.

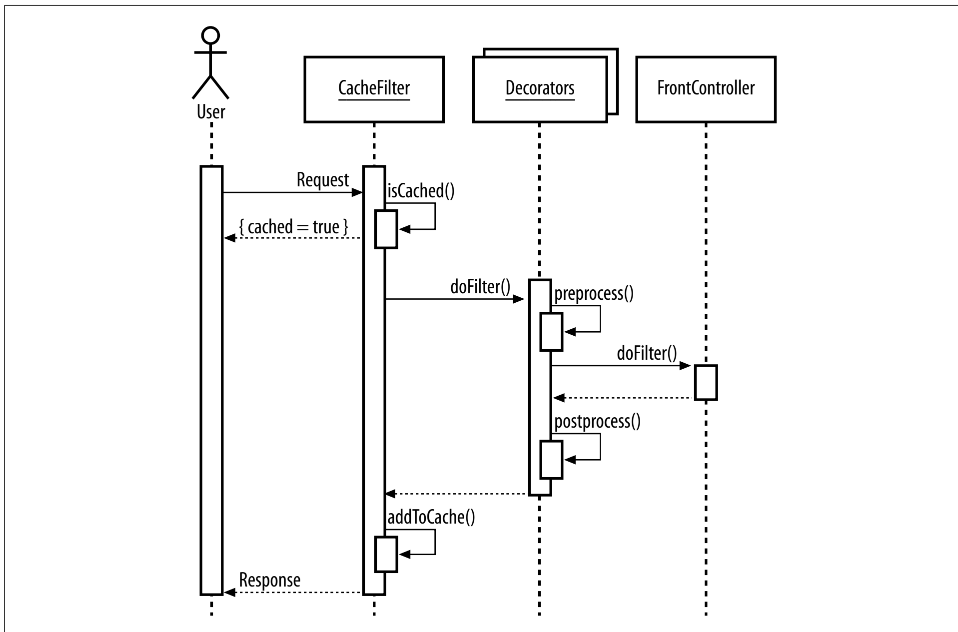
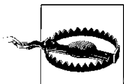


Figure 5-5. Interactions in the Caching Filter pattern

It is important to think about where caching filters are positioned in the filter chain. In principle, a caching filter can be added anywhere in the chain, caching the results from all the filters after it. There can even be multiple caches at different levels in the chain, perhaps caching part of the processing while still doing some of it dynamically.



One frequent mistake is putting caches before security measures. This can lead to unauthorized clients reading cached copies of the data. If a filter is being used for security, it should come before the cache in the chain.

Implementing a Caching Filter

In order to cache the data, we need to change how data is communicated to the server. In many cases, the client simply requests the next page without passing all relevant parameters. The controller uses a combination of the requested page and parameters stored in the session to generate the final output. The client's request might look like:

```
GET /pages/finalPrice.jsp HTTP/1.1
```

To which the server would add its stored entries about the selected options. Unfortunately, the `GET` request looks exactly the same, regardless of the options selected. The fact that the user wants electric blue paint and alloy wheels is not reflected in the request at all. To find a cached page, the session variables would have to be read, sacrificing some of the speed gained by caching.

Instead, the URL should contain all the options in the query string. By including the options, our request might look like:

```
GET /pages/finalPrice.jsp?paint=Electric+Blue&Wheels=Alloy
```

We can implement the most efficient caching when the query string fully specifies the page (see the sidebar "GET, POST, and Idempotence").

GET, POST, and Idempotence

The HTTP 1.1 specification declares that GET requests are *idempotent*, which is Latin for "the same thing can be done." An idempotent action can be repeated as many times as necessary without any ill effects. Our car manufacturer example is an idempotent activity: you can request a final price on the car as many times as you want, without affecting the price.

Actually purchasing the car, however, is anything but idempotent. That's why HTTP provides the POST method to send information to a server. POSTs are not idempotent, which is why sometimes hitting the reload button in a browser brings up a message saying that a form must be resubmitted if the reload is to occur. POST requests have the side benefit of hiding the submitted information from the user (the query parameters don't appear on the browser's location display) and from anyone perusing the server's log files. Credit card information, for example, should only be transferred via a POST, even if you're also using SSL.

Because of idempotence, when implementing caching for dynamic content, we only do it for content generated via GET requests, and even then it's up to the developer to determine when caching is a safe approach.

To implement a caching filter, we will use the servlet's filter API. As we did in Chapter 3, we will decorate the response object we pass down the filter chain with one that stores the results of processing the rest of the chain. This wrapper is implemented in the `CacheResponseWrapper` class, shown in Example 5-3.

Example 5-3. The CacheResponseWrapper

```
public class CacheResponseWrapper extends HttpServletResponseWrapper {
  // the replacement OutputStream
  private CacheOutputStream replaceStream;
  // the replacement writer
  private PrintWriter replaceWriter;
```

Example 5-3. The CacheResponseWrapper (continued)

```
// a simple implementation of ServletOutputStream
// that stores the data written to it
class CacheOutputStream extends ServletOutputStream {
  private ByteArrayOutputStream bos;

  CacheOutputStream( ) {
    bos = new ByteArrayOutputStream( );
  }

  public void write(int param) throws IOException {
    bos.write(param);
  }

  // read back the stored data
  protected byte[] getBytes( ) {
    return bos.toByteArray( );
  }
}

public CacheResponseWrapper(HttpServletResponse original) {
  super(original);
}

public ServletOutputStream getOutputStream( )
throws IOException {
  if (replaceStream != null)
    return replaceStream;

  // make sure we have only one OutputStream or Writer
  if (replaceWriter != null)
    throw new IOException("Writer already in use");

  replaceStream = new CacheOutputStream( );

  return replaceStream;
}

public PrintWriter getWriter( ) throws IOException {
  if (replaceWriter != null)
    return replaceWriter;

  // make sure we have only one OutputStream or Writer
  if (replaceStream != null)
    throw new IOException("OutputStream already in use");

  replaceWriter = new PrintWriter(
      new OutputStreamWriter(new CacheOutputStream( )));

  return replaceWriter;
}

// read back the stored data
```

Example 5-3. The CacheResponseWrapper (continued)

```
  protected byte[] getBytes( ) {
    if (replaceStream == null)
      return null;

    return replaceStream.getBytes( );
  }
}
```

By passing a CacheResponseWrapper to the next filter in the chain, we can store the output in a byte array, which can then be cached in memory or to disk.

The actual caching filter is fairly simple. When a request comes in, it determines if the page can be cached or not. If it can, the filter checks to see if it is in the cache and either returns the cached version or generates a new page and adds that to the cache. The filter code is shown in Example 5-4.

Example 5-4. The CacheFilter class

```
public class CacheFilter implements Filter {
  private FilterConfig filterConfig;

  // the data cache
  private HashMap cache;

  public void doFilter(ServletRequest request,
                       ServletResponse response,
                       FilterChain chain)
  throws IOException, ServletException
  {
    HttpServletRequest req = (HttpServletRequest)request;
    HttpServletResponse res = (HttpServletResponse)response;

    // the cache key is the URI + query string
    String key = req.getRequestURI() + "?" + req.getQueryString( );

    // only cache GET requests which contain cacheable data
    if (req.getMethod( ).equalsIgnoreCase("get") && isCacheable(key))
    {
      // try to retrieve the data from the cache
      byte[] data = (byte[]) cache.get(key);

      // on a cache miss, generate the result normally and
      // add it to the cache
      if (data == null) {
        CacheResponseWrapper crw = new CacheResponseWrapper(res);
        chain.doFilter(request, crw);
        data = crw.getBytes( );
        cache.put(key, data);
      }

      // if the data was found or added to the cache,
      // generate the result from the cached data
```

Example 5-4. The CacheFilter class (continued)

```
      if (data != null) {
        res.setContentType("text/html");
        res.setContentLength(data.length);

        try {
          OutputStream os = res.getOutputStream( );
          os.write(data);
          os.close( );
          return;
        } catch(Exception ex) {
          ...
        }
      }
    }

    // generate the data normally if it was not cacheable
    // or the cache failed for any reason
    chain.doFilter(request, response);
  }

  // determine if the data is cacheable
  private boolean isCacheable(String key) {
    ...
  }

  // initialize the cache
  public void init(FilterConfig filterConfig) {
    this.filterConfig = filterConfig;

    cache = new HashMap( );
  }
}
```

Notice that we haven't made the `cache` variable static. According to the filter specification, only one filter instance will be created for each `filter` element in the deployment descriptor. We can therefore keep separate caches in each filter, allowing multiple caches at different points in the filter chain, without worrying about spreading the same data across multiple cache objects.

Our simple filter avoids two of the difficult parts of caching. The first is determining if a page can be cached at all. In most environments, there will be a mix of cacheable and uncachable pages. In our car dealership example, the various configurations of cars may be cacheable, but a user's credit card information certainly isn't! A typical solution is to provide a mapping, like a servlet or filter mapping in an XML file, to determine which pages can be cached.

The second difficulty this filter avoids is cache coherency. Our model assumes the generated page will never change. If the prices for certain options change, users will still see the old, cached copy of the page. Many coherency strategies are possible, depending on the nature of the generated pages; at the minimum, pages should expire from the cache

after some predefined period. If they don't, the cache grows without bounds, a situation discussed in Chapter 12, when we cover the Leak Collection antipattern.

Resource Pool

A *resource pool* is a collection of precreated objects that can be loaned out to save the expense of creating them many times. Examples of resource pools are everywhere in J2EE. When a connection comes in, a thread is retrieved from a thread pool to handle the request. If the processing requires an EJB, one may be allocated from a pool of EJBs. And if the EJB requires access to a database, the connection will come from—surprise!—a connection pool.

Pools are so prevalent because they solve two problems simultaneously: they improve scalability by sharing the cost of instantiating complex resources over multiple instances, and they allow precise tuning of parallelism and memory use.

To illustrate, let's discuss the classic use case for pools: database connections. The process of connecting to a database, especially a remote one, can be complex and costly. While it may only require a single method call, instantiating a database connection may involve any or all of the following steps:

1. Instantiate the database connection object.
2. Pass in the database address and user credentials.
3. Create a network connection to the database.
4. Perform a complex handshake to determine supported options.
5. Send user credentials to database.
6. Verify the user's credentials.

After all of these steps, the connection is finally ready to use. It's not just expensive in terms of time, either. The connection object must store many of the options it is passed, so each connection requires a fair bit of memory, too. If connections are not shared, the connections and costs grow with the number of requests. As the costs add up, the need to create and maintain database connections limits the number of clients that an application can support.

Obviously, sharing connections is a must for scalability, but sharing has costs, too. At the extreme, you could develop an entire application with a single database connection shared between all the clients. While this effectively removes the cost of creating the database connection, it limits parallelism since access to the database must be synchronized in some way. Sharing database connections also prevents individual components of the application from engaging in transactions (see Chapter 10).

A better solution is to create a number of connections that are shared in a common "pool" between clients. When a client needs access to a database, the client takes a connection from the pool and uses it. When the client is done, it returns the connection so another client can use it. Because the connections are shared, the startup and

maintenance costs are amortized over many clients. The number of connections has an upward boundary, so the creation and maintenance costs don't spiral out of control.

Another major advantage of pools is that they create a single point for effective tuning. Putting more objects in the pool uses more memory and increases startup time, but usually means you can support more clients. Conversely, a smaller pool often improves scalability by preventing a single operation from hogging memory and CPU time. By changing the pool parameters at runtime, the memory and CPU usage can be tailored to each system the application runs on.

The Resource Pool Pattern

The *Resource Pool pattern* can be applied to many costly operations. Pools show the most benefits for objects like database connections and threads that have high startup costs. In these cases, the pattern amortizes the startup costs over multiple objects. But pools can also be adapted for operations—like parsing—that simply take a long time, allowing fine-grained control of memory and CPU usage. Figure 5-6 shows a generalization of a resource pool.

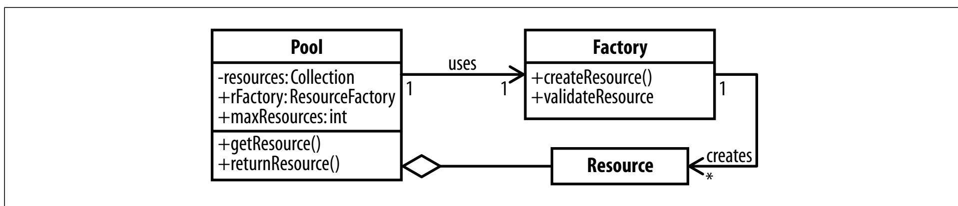


Figure 5-6. The Resource Pool pattern

The `Pool` object is responsible for creating, maintaining, and controlling access to the `Resource` objects. A client calls the `getResource( )` method to get an instance of the resource. When it is finished, it uses the `returnResource( )` method to add the resource back to the pool.

The pool uses a `Factory` object to create the actual resources. By using a factory, the same pool can work for many different kinds of objects. The factory's `createResource( )` method is used to generate new instances of a resource. Before a resource is reused, the pool calls the factory's `validateResource( )` method to reset the resource to its initial state. If, for example, the resource is a database connection that has been closed, the `validateResource( )` method can simply return `false` to have a new connection added to the pool instead. For further efficiency, the factory may even try to repair the returned object—say, by reopening the database connection. This is sometimes called a *recycler method*.

There are really no limits on the `Resource` class itself. The contents and use of a resource must be coordinated between the creator of the pool and the various clients. Usually, pools only store one type of object, but this is not required. Advanced

implementations of pools sometimes even allow a filter to be provided to the `getResource( )` method in order to specify desired criteria of the resource.

Implementing a Resource Pool

Within a servlet engine, a pool of threads handles requests. Each request is handled by a single thread from this pool, which accesses shared instances of servlets to generate the result. In most application servers, the number of threads in the thread pool is configurable at runtime. The number of threads is a critical variable for tuning the scalability of your web application: if the pool is too small, clients will be rejected or delayed; if the pool is too large, the server can't to keep up and the application runs slowly.

Just because the servlet threads are already pooled does not mean we are done with pools. The servlet thread pool represents the most coarse-grained pool possible. Using a single pool assumes that the same thing limits all requests: for example, the speed of XML parsing or connecting to the database. In reality, different requests are usually limited by different operations. Having a separate pool for XML parsers and database connections could allow the total number of threads to be increased, with the limits placed at parsing or connection time, depending on the type of request.

As the pattern description suggests, implementing a resource pool in Java is quite simple. We would like our resource pool to be generic, so that we can easily create a pool of any object by writing an appropriate factory. Obviously, our pool needs to be thread-safe, as we assume that multiple threads will access it simultaneously (any pool used in a servlet environment will face this situation). Example 5-5 shows a simple pool implementation.

Example 5-5. ResourcePool.java

```
import java.util.*;

public class ResourcePool {
    private ResourceFactory factory;
    private int maxObjects;
    private int curObjects;
    private boolean quit;

    // resources we have loaned out
    private Set outResources;

    // resources we have waiting
    private List inResources;

    public ResourcePool(ResourceFactory factory, int maxObjects) {
        this.factory = factory;
        this.maxObjects = maxObjects;

        curObjects = 0;

        outResources = new HashSet(maxObjects);
        inResources = new LinkedList( );
    }
```

Example 5-5. ResourcePool.java (continued)

```
    // retrieve a resource from the pool
    public synchronized Object getResource( ) throws Exception {
        while(!quit) {

            // first, try to find an existing resource
            if (!inResources.isEmpty( )) {
                Object o = inResources.remove(0);

                // if the resource is invalid, create a replacement
                if(!factory.validateResource(o))
                    o = factory.createResource( );

                outResources.add(o);
                return o;
            }

            // next, create a new resource if we haven't
            // reached the limit yet
            if(curObjects < maxObjects) {
                Object o = factory.createResource( );
                outResources.add(o);
                curObjects++;

                return o;
            }

            // if no resources are available, wait until one
            // is returned
            try { wait( ); } catch(Exception ex) {}
        }

        // pool is destroyed
        return null;
    }

    // return a resource to the pool
    public synchronized void returnResource(Object o) {

        // Something is wrong.  Just give up.
        if(!outResources.remove(o))
            throw new IllegalStateException("Returned item not in pool");

        inResources.add(o);
        notify( );
    }

    public synchronized void destroy( ) {
        quit = true;
        notifyAll( );
    }
}
```

Example 5-5 assumes the very simple factory interface we sketched earlier:

```
public interface ResourceFactory {
    public Object createResource( );
    public boolean validateResource(Object o);
}
```

To see resource pools in action, let's look at an operation that is used frequently but rarely pooled: XML parsing. Like database connections, XML parsers can be expensive to create and maintain. By using a pool of parsers, we not only share the cost of creating them, we can control how many threads are performing expensive XML parsing operations at any given time. To create a pool of parsers, all we have to build is the `XMLParserFactory` shown in Example 5-6.

Example 5-6. XMLParserFactory

```
import javax.xml.parsers.*;

public class XMLParserFactory implements ResourceFactory {
    DocumentBuilderFactory dbf;

    public XMLParserFactory( ) {
        dbf = DocumentBuilderFactory.newInstance( );
    }

    // create a new DocumentBuilder to add to the pool
    public Object createResource( ) {
        try {
            return dbf.newDocumentBuilder( );
        } catch (ParserConfigurationException pce) {
            ...
            return null;
        }
    }

    // check that a returned DocumentBuilder is valid
    // and reset parameters to defaults
    public boolean validateResource(Object o) {
        if (!(o instanceof DocumentBuilder)) {
            return false;
        }

        DocumentBuilder db = (DocumentBuilder) o;
        db.setEntityResolver(null);
        db.setErrorHandler(null);

        return true;
    }
}
```

To use our pooled XML parsing mechanism, a simple client might look like:

```
public class XMLClient implements Runnable {
    private ResourcePool pool;
```

```
    public XMLClient(int poolsize) {
        pool = new ResourcePool(new XMLParserFactory( ), poolsize);
        ...
        // start threads, etc.
        Thread t = new Thread(this);
        t.start( );
        ...
        // wait for threads
        t.join( );
        // cleanup
        pool.destroy( );
    }

    public void run( ) {
        try {
           // get parser from pool
           DocumentBuilder db = (DocumentBuilder)pool.getResource( );
        } catch(Exception ex) {
           return;
        }

        try {
           ...
           // do parsing
           ...
        } catch(Exception ex) {
           ...
        } finally {
           // make sure to always return resources we checkout
           pool.returnResource(db);
        }
    }
}
```

Resource pools look good on paper, but do they actually help for XML parsing? And if we are going to use them, how do we choose the correct size? Let's take a minute to look at the real-world use and performance of pools.

An important step in using a pool is sizing it properly. For our test systems, we used a two-CPU server and a six-CPU server, both with plenty of memory; we expected to be able to handle a fair number of threads. Using a sample program similar to the one outlined above, we looked at how long it took to parse a 2,000-line XML file with various combinations of number of threads and pool size. Table 5-1 shows the optimal pool size for each thread count on each server. It's not surprising that for a CPU-limited task like XML parsing, the optimal pool size is generally pretty close to the number of CPUs in the system. For an I/O-limited task, like reading data from the network, we would expect very different results.

Table 5-1. Optimal pool sizes for various numbers of threads

Thread count	Optimal pool size 2 CPUs	Optimal pool size 6 CPUs
1	1	1
2	2	2
4	4	4
8	2	8
16	2	15
32	2	6
64	2	6
128	2	6
256	2	6
512	2	4
1024	2	6

Now that we have optimal pool sizes worked out, we can see the scalability improvements. We tried two variations of the sample program, one with the optimal pool size, and one with no pool, and compared the time required per thread. Figure 5-7 shows the results of our experiment.

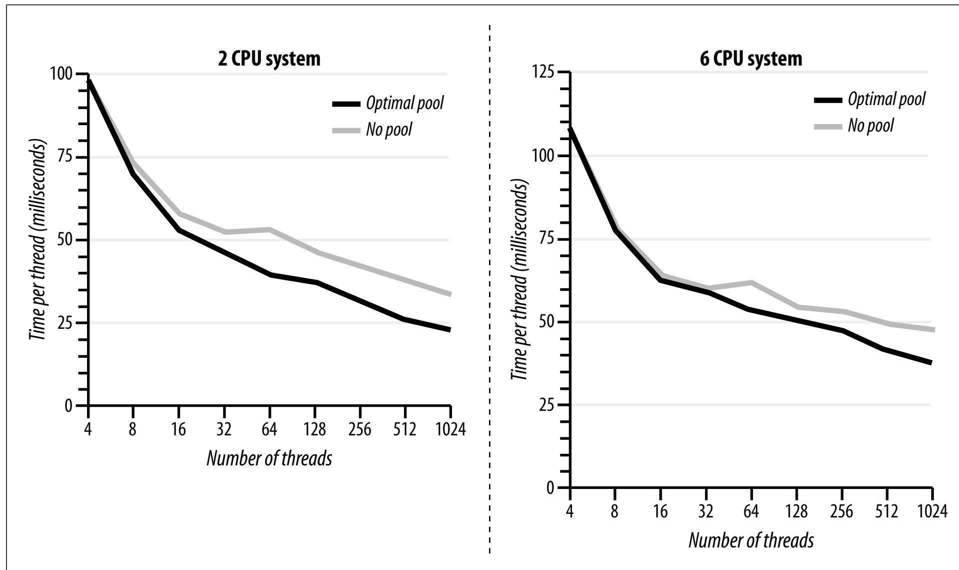


Figure 5-7. XML parsing speed with and without a pool

The pool gives a significant advantage, especially when more than 32 threads are active. These results fit into our theory that pools increase scalability for CPU-intensive tasks by limiting the overhead of switching between too many tasks at

once. It should not be surprising that in addition to the speed gains we saw with pools, there was also much less variation between different trials with the same number of threads when pools were in use.

In this chapter, we looked at three patterns that increase the scalability of the presentation tier. The Asynchronous Page pattern shows how to cache data when it is read from external sources. The Caching Filter pattern describes how to cache entire pages as they are generated. The Resource Pool pattern creates a pool of expensive objects that can be loaned out. All these patterns also allow the developer to tune the application, balancing memory and CPU use for the entire system.