Pro Web 2.0 Application Development with GWT

Jeff Dwyer

Apress®

Pro Web 2.0 Application Development with GWT

ISBN-13 (pbk): 978-1-59059-985-3

ISBN-10 (pbk): 1-59059-985-3

ISBN-13 (electronic): 978-1-4302-0638-5

ISBN-10 (electronic): 1-4302-0638-1

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editors: Steve Anglin, Ben Renow-Clarke
Technical Reviewer: Massimo Nardone
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Kylie Johnston
Copy Editor: Heather Lang
Associate Production Director: Kari Brooks-Copony
Production Editor: Liz Berry
Compositor: Dina Quan
Proofreader: Linda Marousek
Indexer: Carol Burbo
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `http://www.apress.com/info/bulksales`.

The source code for this book is available to readers at `http://www.apress.com`.

Contents at a Glance

About the Author

JEFF DWYER is a developer and entrepreneur who is the founder of ToCollege.net and MyHippocampus.com. His background is in medical software, where he has published research on aneurysm stress and endovascular repair and has patented techniques in anatomical visualization. He currently works at PatientsLikeMe. As a developer, Jeff likes nothing better than to leverage high-quality, open source code so he can focus on getting results. He believes that GWT has fundamentally altered the feasibility of large Web 2.0 applications. Jeff is a graduate of Dartmouth College and lives in Vermont, where he enjoys skiing, rowing, and interminably long winters.

About the Technical Reviewer

MASSIMO NARDONE was born under Mount Vesuvius. He holds a master of science degree in computing science from the University of Salerno, Italy. He currently works as an IT security and infrastructure architect, a security consultant, and the Finnish Invention Development Team Leader (FIDTL) for IBM Finland. His responsibilities include IT infrastructure, security auditing and assessment, PKI/WPKI, secure tunneling, LDAP security, and SmartCard security.

With more than 13 years of work experience in the mobile, security, and WWW technology areas for both national and international projects, he has worked as a project manager, software engineer, research engineer, chief security architect, and software specialist. He also worked as a visiting lecturer and supervisor for exercises at the Networking Laboratory of the Helsinki University of Technology (TKK) for the Security of Communication Protocols course.

Massimo is very familiar with security communication protocols testing tools and methodologies and has been developing Internet and mobile applications involving different technologies using many programming languages. He also works as a security application auditing expert to check on new application vulnerabilities utilizing security standards like ISO 17799 and ISO 27001 (formerly BS7799:2).

He has researched, designed, and implemented security methodologies for different areas like Standard BS7799, PKI and WPKI, Security Java (JAAS, JSSE, JCE, etc.), BEA Web Logic Security, J2EE Security, LDAP Security, SSO, Apache Security, MS SQL Server Security, XML Security, and SmartCard Security.

Massimo has served as a technical reviewer for many different IT book publishers in areas like security, WWW technology, and databases.

He currently holds four international patents (PKI, SIP, SAML, and proxy areas).

Acknowledgments

I'd like to thank everyone at Apress who made this book possible. Kylie Johnston, you deserve a medal for keeping this book on track despite my best efforts to the contrary. Heather Lang, your grammatical psychiatry was much appreciated. I have a long way to go before attributive nouns and I see eye to eye, but I think we made big strides. Thanks also to Steve Anglin, Ben Renow-Clarke, Stephanie Parker, and Tina Nielsen. I've enjoyed working with you all. Finally, thanks to the technical reviewer, Massimo Nardone, known only as IBM_USER throughout the project. Though your identity was a mystery, your feedback was first rate.

Thanks to Sumit Chandel and Pamela Fox for your Herculean efforts to ensure that I was not sued as a result of this book. I shall recommend that you receive the Order of the Red Tape Cutters. Thanks to everyone at the Google Web Toolkit conference. From GWT team members to presenters to attendees, you all helped hone the message that I've tried to get across in this book.

To all of the various nontechnical people I know, merci beaucoup—and thank you for feigning interest! Thanks to my dear mother as well, who has had to put up with more of this than anyone should really have to. And to every one of the Hamlet Project members, truly, you keep me sane. If there are more than three people, they might just think it's a revolution. J3PO, this is just the start.

Finally, a large *mwah* to Brenda: I'm very much enjoying our recipe for success.

Introduction

When I quit my day job in the summer of 2006 to bring an idea of mine to life as an Internet startup, I was faced with a huge number of questions and not a lot of clear answers. The excitement of starting a new project was soon tempered by the difficulty of choosing among the dizzying array of possible technical platforms and solutions. While the shelves were full of books focusing on each of the components of a web application, what I really wanted was a look at how all the pieces fit together.

Too often, I found books that seemed like annotated versions of Javadocs, with helpful explanatory notes that `setEnabled(boolean enabled)` would set the enabled flag. At the other end of the spectrum were abstract architectural tracts that might have made for good reading but certainly didn't give me a head start on the architecture. I wanted to see how these technologies worked when used for nontrivial applications; I wanted to see ugly problems and their solutions—I wanted a look at the source code of a modern web application.

For those of us who learned HTML and JavaScript by using the View Source option, there's nothing like getting your hands on working source code. So I had an idea: what if I created a company, a real, functional company, with a modern web site and then gave you the keys to the castle to go poke around? That's the goal of this book, to give you the tour of a cool, state of the art Web 2.0 application.

The main content of this book focuses on developing one big site with the Google Web Toolkit (GWT). In the summer of 2006, GWT 1.0.21 had been freshly released, and I had the good fortune to give it a test drive. It turned out to be a fortuitous choice. Working in an unfunded start-up shines a powerful spotlight on the relationship between writing code and getting results. The only thing worse than struggling for hours to get your site to work in different browsers is not getting paid for those hours! Programming with GWT meant I didn't need to get a master's degree in browser compatibilities and that I could refactor my code quickly and easily as my ideas changed. Most importantly, programming in GWT was fun.

There's a lot of material in this book that isn't purely related to GWT, which, in my mind, is a good thing. I could've cropped out some of this information to make things more focused, but it seems to me that this interplay between things is precisely what takes the most time to figure out on your own, and that discussing this interplay is the area in which this book can be most valuable. Walking through online tutorials is usually enough to get your head around one technology, but getting all the elements to play nicely together is where the rubber hits the road.

Who This Book Is For

This book is for people who are looking to get beyond small proof-of-concept sample applications. It's for people who want to see what the guts of a full-fledged GWT application should look like. Perhaps you've already been convinced that GWT is a technology worthy of your attention. You've certainly read some simple tutorials and comparisons of frameworks, and you're familiar with the pros and cons. Before you commit, however, you want to get your hands dirty and find some definitive answers to some of the cons that you've heard about, such as questions about how GWT works with Hibernate and whether using GWT will prevent Google from indexing your site. You want to get beyond sending a simple object to the server over RPC to issues of caching, asynchronous command patterns, security, and the best practices of a GWT-enabled site.

If you have a background in Swing or SWT GUI toolkits, you're going to love GWT. If you've developed a Swing client application with remote services, you'll feel right at home programming with GWT, even if you've never programmed a line of JavaScript. You'll need to get used to some aspects of the web paradigm and study up on CSS, but the development environment we'll set up in this book will give you a clear, industry-standard way to deploy your applications, and you'll be able to thank your lucky stars that you got to skip the days of debugging Internet Explorer 6 scripts. If you're coming from JavaScript and text editors, be prepared for a lot of Java coming your way in this book, no bones about it. We're going to look at GWT itself, but we're also going to spend a lot of time integrating it into a powerful Java server-side application.

I'll assume that you're comfortable using online documentation and forums when you're stuck, and that you feel that an API's rightful place is in a developer's IDE or online Javadoc and not on dead trees. This book leaves configuration-specific details to these sources, so I can focus on answering the bigger architecture questions. I'll give you tips along the way, though, and in the end, you'll be able to see the proof in our pudding, since you'll have the source code to a fully functional web site. This approach recognizes that the modern developer's work flow is constantly Web enabled. Answers to specific questions are far more easily found on Google or in forum posts than in book indexes. What I hope to gain by adopting this approach is to convey the larger issues surrounding a GWT application. By chopping down a couple of the trees, I'm hoping to more clearly define the forest.

How This Book Is Structured

This book has two parts. The first part is a short, sweet introduction to the Web 2.0 landscape and where GWT fits into it. The main focus of the book is Part 2, which explores a full-fledged application called ToCollege.net. This application specializes in helping students who are applying to colleges, helping them get on top of their application process and letting them compare the rankings that they give to each school. This book also

sports an appendix that provides you with everything you need to get the source code for the site and instructions for getting it all running on your machine.

- Part 1: What Can GWT Do for You?
  - Chapter 1, "Why GWT?" lays out the GWT value proposition. We'll look at the problems that GWT was created to help solve.
  - Chapter 2, "Getting Started," is where we'll write our first GWT code together. We'll mimic a sample application from a book on pure JavaScript, and we'll show how a GWT edition of this simple application improves on the original.
- Part 2: ToCollege.net
  - Chapter 3, "Designing ToCollege.net," will provide a broad overview of the ToCollege.net application. We'll discuss the functionality that we want to deliver with this application, and we'll go over the domain design that we'll use throughout the rest of the book.
  - Chapter 4, "GWT and Spring MVC," dives right into the action and shows how to integrate GWT with one of the most popular web frameworks. We'll go over the reasons for integrating with a framework and why we chose Spring MVC. After that, we'll dive into the details of how to connect these two technologies.
  - Chapter 5, "Securing Our Site," will show you how to apply the robust industry standard Acegi Security for Spring package to our Spring MVC web site. We'll even look beyond basic form-based authentication at how to accept OpenID logins.
  - Chapter 6, "Saving Our Work," will focus on getting Hibernate set up on the server side and will talk about the GWT-specific issues that crop up when trying to use GWT-RPC together with Hibernate. We'll also develop the ToCollege.net Command pattern, which is going to be the fundamental architectural feature of ToCollege.net.
  - Chapter 7, "ToCollege.net's GWT GUI," brings our focus back to the GWT client side, where we'll explore how to write responsive, interactive GUIs. We'll also look at the amazing `ImageBundle` class, which will allow us to drastically minimize the number of `HTTPRequest` requests that our page will need to load.
  - Chapter 8, "Google Maps," will show us how to integrate the maps into our project using the GWT Google APIs project. We'll also cover geocoding, and by the end of this chapter, we'll have maps that show the location of all of the schools in ToCollege.net.

- Chapter 9, "Suggest Boxes and Full Text Search," will start out by showing you how to create text boxes that suggest entries to the user. We'll end up deciding that to get the proper results, we'll need real full text search capability, so we'll go ahead and set up the Compass search engine on the server.

- Chapter 10, "Forums," will cover a GWT-based forum system that will let our users communicate with each other. It will also give us an opportunity to discuss the JavaScript Native Interface (JSNI).

- Chapter 11, "Security and Authorization," is a critical chapter for anyone concerned about writing a secure web site. We'll cover the security considerations that GWT users need to be aware of and the ToCollege.net response to XSS and XSRF attacks.

- Chapter 12, "Search Engine Optimization," will show you how ToCollege.net solves one of the stickiest issues with rich AJAX web sites. Because search engine spiders don't execute JavaScript, it's far too easy to write GWT sites that are entirely opaque to search. Not showing up in searches isn't an option for ToCollege.net. I'll take you through the code that let's Google index and crawl the site.

- Chapter 13, "Google Gears," shows how to integrate Google Gears into ToCollege.net. With Google Gears support, we'll end up being able to leverage the full power of a SQL database right from our GWT application in order to create easy-to-use request caching.

- Appendix: This book's appendix, "Building ToCollege.net," will go over everything you need to get the full ToCollege.net project code running on your development machine. It will cover Maven, Eclipse, and MySQL setup.

Downloading the Code

The source code for everything in this book is hosted on the ToCollege.net project on Google Code at `http://code.google.com/p/tocollege-net/`. This book's appendix gives you all the details you need to download this code and set up your development environment so that you can run your own copy of ToCollege.net. The source code for this book is also available to readers at `http://www.apress.com` in the Downloads section of this book's home page. Please feel free to visit the Apress web site and download all the code there. You can also check for errata and find related titles from Apress.

Contacting the Author

For questions or issues about this book, check out the project home page at `http://code.google.com/p/tocollege-net/`. This page has a wiki, an issue tracker, and a link to the Google Group for this project.

CHAPTER 12

Search Engine Optimization Making AJAX Searchable

ToCollege.net does not exist in a bubble. It needs users. In fact, it needs lots of them. A fair piece of the value that we'd like to provide our users comes from the value provided in our school-specific forums. This is a great way for users to get the scuttlebutt on which schools are really good places to be and which are just an extraordinarily expensive way to spend your formative years drinking light beer. Of course, if $160,000 sounds like a decent price for free beer, the forums will let you know who's got the best. In short, the forums are a repository for the real scoop on colleges. The only problem? Right now, they're empty.

How do we fill them up? Well, the first step in doing that is to make them as readily available as we can. Supporting OpenID is a great first step toward that, because it reduces the barrier to entry for our users. Anyone with a Yahoo, AOL, or other OpenID-compatible account can immediately log in and leave a message without going through any signup shenanigans. With that solved, the problem becomes one of getting our users to ToCollege.net in the first place. How do we get users to our site? I've got just three words for you: search, search, search.

In this chapter, we're going to look at how to make our GWT site search engine friendly. To do this, we're going to reimagine the way that we get data to our GWT widgets, moving from RPC to a bootstrapping method where we'll use GWT serialization to serialize our objects right into the HTML host page.

How Search Works

The Internet is driven by search results. Search engines are everyone's start pages and the way we all look for information. So how does it work? Getting from your server to your users' search results happens in three steps: crawling, indexing, and serving. Let's look at these steps now, so you can see what we'll need to optimize for.

Crawling

First of all, Google (we'll use Google as our example search engine in this chapter, but most search engines will follow a process similar to this) needs to know that your page exists. Google achieves this by following links from one page to another. Google will find our site eventually once we can get someone to link from an existing site to our web page, but we can speed this process along by submitting our site directly to Google by going to `http://www.google.com/addurl/`. Once our main page is added, Google will start sending a robot (named the Googlebot) to our site to perform indexing. The Googlebot will read in our start page and follow all of the links on the page. Each of those links will then be indexed and crawled as well. The important thing for us to think about is that every page we want to be searchable *must be linked* through a chain of links back to the main page. If we have a page that's only accessible via some other method (such as a search or a JavaScript event) it will never be found, because the Googlebot won't use our search button or our JavaScript code.

Indexing

For each page that the Googlebot crawls, the HTML on the page will be fed into the Google indexer. The indexer reads through the page and picks out the keywords that represent this page. So what does that mean? Well, it means that the text on the page is king. Images will be ignored unless they have alternate text specified. JavaScript will typically be skipped over. To give you an idea of what this looks like, in this chapter, we'll look at some tricks for seeing what our pages look like to the Google indexer, so we can make sure we get it the right information.

Serving

The last step happens when a user goes to the search bar and looks for keywords. Google has indexed a ridiculous number of pages, and it needs to figure out whether to send back your results, or your competitors, or Wikipedia's, or any other page on the Internet. How Google chooses is based on your site's page rank (PageRank), which is a rough guide to how popular your web site is based on how many other sites link to your site and a number of other factors. This is a bit of a chicken and egg problem, but a lot of Google's success is due to their ability to properly assign PageRank. In short, if we've got good content and it's crawlable and indexable, we should be able to show up well in search results.

Search Engine Optimization (SEO)

Now that you've seen the three steps that make up the search process, what can we do to optimize our results? Well, we can do a little search engine optimization; that's what.

You should be aware that SEO has a bit of a split personality. On one side, SEO is one of the best things you can to do to be a good Internet citizen. See the compelling argument at `http://www.alistapart.com/articles/accessibilityseo/`. It points out that the only thing you should really need to do to show up well in Google search results is to design your site to be exceptionally well formed with regard to the guidelines for accessibility: the argument is that if your site can display its content well for screen readers and other alternative forms of browser, it will, by definition, be much easier for the Googlebot to index. Besides being a nice idea, this also happens to be true.

On the other side, SEO is a bit of a catchall for all sorts of dirty tricks that you can use to try to boost your search results. Searching for SEO will find any number of dubious providers that promise to boost your `PageRank`. Frankly, even if we weren't the paragons of virtue that we are, these schemes seem like more trouble than they're worth. All the search engines have lots of smart people focused on preventing this sort of abuse, and quite frankly, I have a feeling that they're winning. Getting our page blacklisted for bad practices is definitely not on our to-do list, so we'll focus on doing things the right way here.

In fact, the main takeaway of this is that bad SEO practices are so prevalent that we need to be careful not to get caught in the net. Not only do we have to be honest about the way we serve up our pages, we're going to want to appear squeaky clean. There are some techniques that might work that we'll avoid simply because we can't afford to look like one of the bad guys.

■ **Note** Google's advice on this subject is pretty well documented. You can read their webmaster guidelines here: `http://www.google.com/support/webmasters/bin/answer.py?hl=en&answer=35769`. You owe it to yourself and your PageRank to read over this document carefully and to try and do as much as you can to conform to these guidelines. We'll be doing everything we can to conform to them here. Also, you owe it to yourself to sign up for the Google webmaster tools (available here: `http://www.google.com/webmasters/start/`), where you'll be able to get a better idea of what Google sees when it indexes your site. The web crawl diagnostics it provides are worth the trip by themselves, as they're a great way to discover any 404 errors your page might have.

With the description of SEO out of the way, let's look at the specific problems of AJAX-heavy sites such as ours.

AJAX and Searching

You've got the idea of how search works and seen that making our site accessible is a good solution to our problem. So what's the problem? Well, the problem is that this book is about AJAX web sites, and JavaScript web sites are inherently pretty inaccessible. JavaScript is just bad news when it comes to searches.

Let's look at crawling. Say we design our page in GWT. We have a simple start page with a couple links to pages that are specific to each school. We might have code that looks something like the following:

```
HorizontalPanel hp = new HorizontalPanel();
hp.add(new Hyperlink("Dartmouth College", "dartmouth_college");
hp.add(new ExternalLink("About Us", "http://tocollege.net/site/about.html");
RootPanel.get("slot-1").add(hp);
```

When we compile our GWT code, we'll have a host page that looks something like the following:

```
<div id="slot1"></div>
<script language='javascript'
   src='com.apress.progwt.SampleApp.nocache.js'></script>
```

Not so good. Yes, somewhere inside the JavaScript that will be loaded, the text "Dartmouth College" will appear, but search engines aren't going to make it that far. All they'll see is `<div id="slot1"></div>`, and they'll think to themselves, "This is a mighty boring page." So as far as crawling goes, this page is a total bust. We lose our link to the About Us page, so Google will never find that page. What's worse is that this page is an indexing disaster as well. We don't even get search index credit for having mentioned Dartmouth. The only way we'd show up in a search result might be if somebody happened to search for "slot1", and that's not really what we're shooting for.

GWT and Searching

Now, is this search opacity just a GWT problem? Not really. This is really a fundamental problem with rich content. Sites that depend on JavaScript, Flash, Silverlight, and RealAudio are all going to suffer in much the same way (or even worse). Even a site with animated GIFs is going to have to find a way around the inherent difficulty in making rich content searchable.

That said, GWT can definitely exacerbate this problem. As I said in the first chapter, GWT was developed so that developers could scale their AJAX applications. With GWT, big applications are possible. With simpler AJAX frameworks, it's a good idea to compartmentalize functionality in a different order to save sanity, and this can result in web pages that have a bit of an HTML/JavaScript mix and are thus at least partially indexable. With GWT, however, we have such a powerful tool for rich web application design that it's tempting to put a large application in just a single "slot1" `<div>` like we did previously. While we can all hope that Google has some magic trick up its sleeve that will let the Googlebot index GWT automatically, there's no hint that this is on the radar. With that in mind, we're going to need to come up with something else.

In this chapter, we'll take a look at how we can mitigate these issues with GWT development while still retaining the ability to write mind-blowingly dynamic applications.

Optimizing ToCollege.net for Indexing

Our problem comes in two parts: the ability to crawl and the ability to index. Let's consider how the Googlebot indexes the naïve implementation of our site first.

Let's take a look at what we've got so far with our ToCollege.net application. Figure 12-1 shows the page at `http://tocollege.net/site/college/Dartmouth_College.html`.

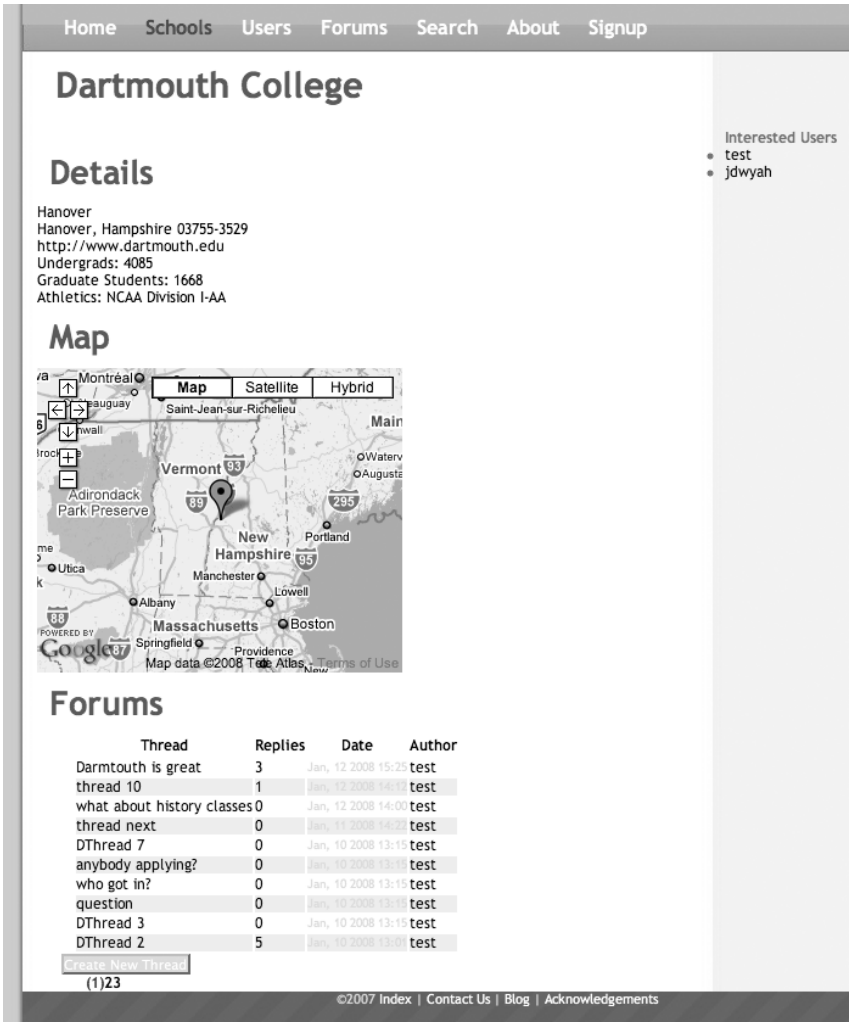

Figure 12-1. *Dartmouth College's page with its school-specific forum*

It's a decent looking page. We have a nice URL, so that's a good start. This is no small feat, and we can chalk it up to our nice Spring MVC `CollegeController` that we're able to have this nice REST-style URL. On top of that, we have some useful information about the school itself. Last and most importantly, we have the really good stuff—the school-specific forums. This is the real content that our site contains, and it's precisely what we want Google to index. So what does the Googlebot see when it comes to visit?

Be the Bot: Visualizing Your Site Like a Bot Does

We have our college page in Figure 12-1, but it's tough to imagine exactly what the Googlebot is going to see. If this were *Starship Troopers*, Neil Patrick Harris might say, "To beat the Googlebot, we must understand the Googlebot."

How do we do that? Let's step way back to the dawn of the Internet and fire up a text-only browser!

Using Lynx

Lynx is a text-only web browser. You can find it at `http://lynx.isc.org/`, but you'll need to compile it from source, which is a bit of a pain. For Windows, I found a precompiled Win32 version at `http://www.vordweb.co.uk/standards/download_lynx.htm`. Once you download and unzip this file, you should be able to run the `lynx.exe` file in the output directory.

■ **Note** If you don't want to use Lynx, you can also use the very handy browser simulators at `http://www.seo-browser.com/` and `http://www.webconfs.com/search-engine-spider-simulator.php`. These will allow you to simply type in a URL and will present you with a text-only display of the page. This tool will only work on pages that are online already, however, so it's nice to have Lynx installed for development work.

So, what does our ToCollege.net page look like in text-only mode? Let's see. Once we fire up Lynx, we can just type **g** to go to a specific page. If we type **http://localhost:8080/site/college/Dartmouth_College**, we'll be able to see our college page. Use the up and down arrows to move between links and the Enter key to follow a link. Type **q** to quit Lynx. Enough directions, let's take a look at our page in Figure 12-2.

Yikes! I see why they invented graphical web browsers; that's not too attractive. The good news is that our page seems to have at least a portion of the information we need. The browser outputs the college details using the FreeMarker template, so they can be indexed. But our forums are a disaster. Right where we wanted our treasure trove of searchable data all we get is "Loading." That's not good.

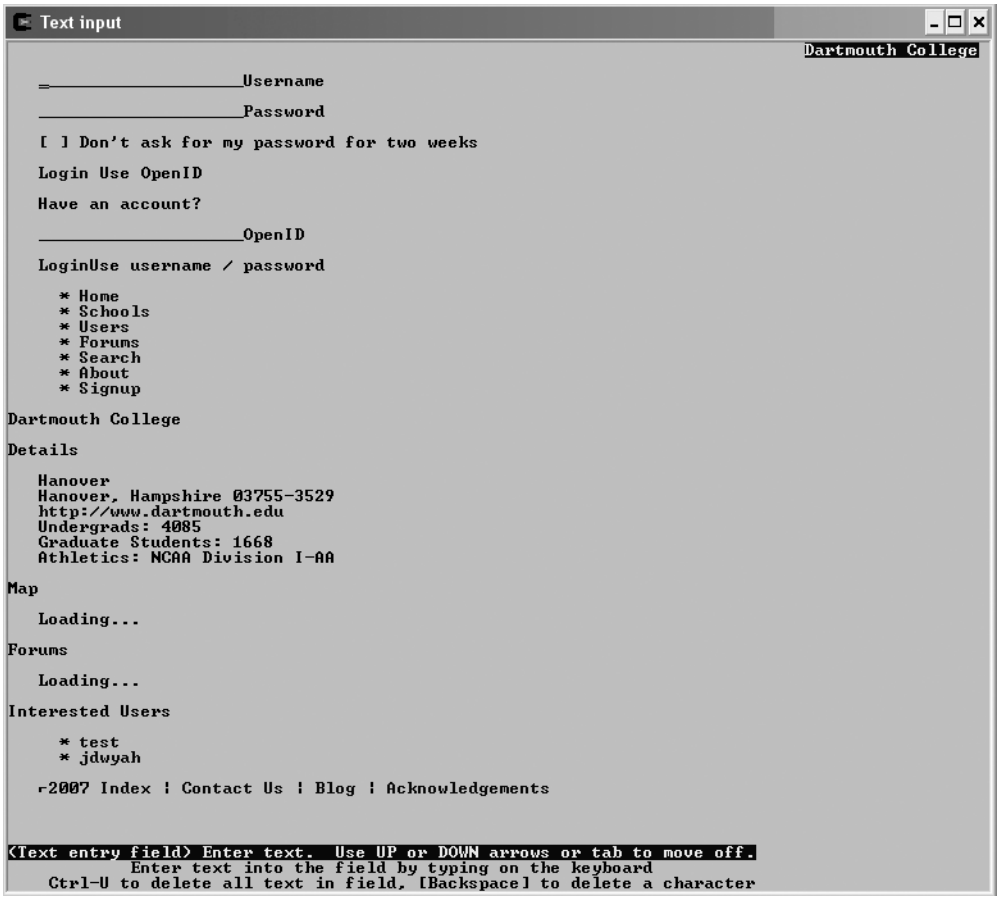

Figure 12-2. *The Dartmouth College page in the eyes of the Googlebot. Where are our forums?!*

Of course, this output is just what we would have expected. Our forums are pure GWT, and we loaded them all into a single `<div>`. When Lynx got to the page, all it saw was the "Loading" message, and because it didn't read and execute the JavaScript, it never saw any of the changes that happen to the DOM in a regular browser. So, that leaves us at a bit of an impasse. We're happy with the utility of our forums, but they don't fulfill a primary need of ours—they aren't searchable. Let's look at our options:

- Create a parallel site
- Change content based on user agent
- Display the content in hidden `<div>` or `<noscript>` elements

Option 1: Secondary Site

The most brute-force solution to this problem is to simply throw our hands in the air and write a secondary site. We could go through and write a bare bones version of our forums that would just display pure HTML. While this isn't a terrible solution, it's certainly not ideal. We'll need have two separate (but similar) URL schemes, as well as some way for real users that go to the secondary site to get back to the rich primary site, and we'll need links from the primary site to the secondary site so that the bots can crawl it. In short, it's a bit of a mess, but it will definitely work. Let's see if we can do better.

Option 2: User Agent

One way of looking at this problem is as a user agent problem. The user agent is the HTTP header that specifies what type of browser is visiting our web site. When the Googlebot comes calling, it would be pretty easy for us to parse the user agent out of the `HTTP-Request` and serve up a different, more bot-friendly page. Of course, we'd have to include the other bots as well, but this wouldn't be too hard. If we then write the secondary site from the previous option, we'd be able to direct the bots to the secondary site, while keeping regular browsers on the GWT-enhanced page. That way, everybody gets what they need: our user gets rich functionality, and the bot gets clean text to read and index, and our SEO problem is solved.

So what's not to like? First of all, just like the first solution, this will require us to write and maintain a secondary site. Even if that were easy, there's another problem. This practice of sending the Googlebot to a different resource than normal users is called shadowing, and it makes the SEO enforcers at Google very wary. This is precisely the sort of thing that an evil web site might try to do to appear in search results for "George Washington" (by giving the Googlebot Wikipedia entries) while actually displaying gambling or penny stock scams to regular users. Trying to figure out how much of this user agent detection is valid is a bit tricky, and there are differing opinions as to what Google thinks on this issue. We're going to take the easy way out and just avoid this practice all together.

Option 3: Hidden and <noscript> Elements

The fundamental problem with our current design is that information simply isn't available at the time the page is rendered, and this is something we can certainly work around. All we need to do to get the information here is add it to the model in our controller. That's easily done, and we can format it in a simple SEO-compatible way without too much trouble, but how do we display it in a way that lets the bot, but not our users, see it?

Well, one solution that is regularly suggested is to put it inside a `<div>` and set the style to `display: none;`. Because most bots don't read CSS, this will probably work. It's not ideal, however, because these styling elements were meant for other uses. Again, we run

the risk of the Google indexer thinking that we're up to no good by trying to hide different content in our document.

A better solution is to put this output in a `<noscript>` tag. This is more representative of what we're actually trying to do, and is thus a bit more "honest." Bad web site owners abuse the `<noscript>` tag in much the same way as they do comments and other meta fields, so there's a possibility that we'll still be flagged as an evil-doer, but at least we're doing what we're supposed to and should have no problem defending ourselves if the SEO enforcers look closer.

So, this is a decent solution and it's what ToCollege.net uses. Every time we put a GWT widget on our page that should be indexable, we'll put up a `<noscript>` tag and cough up the contents of the page there. Of course, this puts a burden on us, since we're going to have to be able to find a way to represent the model again, but this was pretty much unavoidable.

There is one other unfortunate repercussion of this solution. Now that we're displaying the forum information on the page, the asynchronous `load()` call that the forums made on startup starts to be kind of redundant. For one thing, we'll end up doubling the number of requests to the database, since we're now requesting the forums once for the FreeMarker page and once when GWT loads. Let's see if we can do a little better.

Implementing Bootstrapping

The reason we're making two requests is a form of an impedance mismatch. We have the forum model all set on the server, and we can output the HTML using the FreeMarker template, but our GWT forum needs this data over GWT-RPC. Or does it?

While it's certainly the traditional thing to do to populate our GWT widgets with asynchronous calls over RPC, it's by no way the only way. Indeed, we've already found an alternative way to pass data to GWT in previous chapters. One example of this was when we passed longitude and latitude coordinates in JavaScript dictionaries to our `CollegeMapApp`. But how could we pass our whole forum model, which is essentially a `PostsList` object (a `List<ForumPost>` wrapper)? One solution would be to work out a JSON representation. This is certainly doable, but is it really our only solution? We already have an automatic serialization scheme set up for GWT-RPC. Can we just use that? Let's begin with reimagining the perfect GWT widget FreeMarker macro. This is our interface between the server and GWT, so it's a good place to start.

Updated Freemarker <@gwt.widget>

Since our FreeMarker template is the place where we're asking this magic to happen, wouldn't it be nice if we could simply write something like Listing 12-1 into our college template? Then, it could output both a `<noscript>` element and some GWT widget inclusion code that would pass the forums over as a serialized block of information.

Listing 12-1. *src/main/webapp/WEB-INF/freemarker/college.ftl*

```
<@common.box "boxStyle", "forums", "Forums">
    <@gwt.widget widgetName="Forum" bootstrap=model.postList />
</common.box>
```

We've been using the `extraParams` parameter of the `widget` method, but now, we're going to hypothesize a new bootstrap parameter. This sure seems easy enough to do. Listing 12-2 shows what the `widget` method will need to do to accomplish our twin goals of serialization and `<noscript>` display.

Listing 12-2. *src/main/webapp/WEB-INF/freemarker/commonGWT.ftl*

```
<#assign widgetID = 0>
<#macro widget widgetName, extraParams={}, bootstrap="">
    <#assign widgetID = widgetID + 1>
    <#if widgetID == 1>
        <script language="JavaScript">
            var Vars = {}
        </script>
    </#if>
    <script language="JavaScript">
            Vars['widgetCount'] = "${widgetID}"
            Vars['widget_${widgetID}'] = "${widgetName}"
            <#list extraParams?keys as key>
                Vars['${key}_${widgetID}'] = "${extraParams[key]}"
            </#list>
            <#if bootstrap?has_content>
                <#--Replace \ with \\ and " with \"-->
                Vars['serialized_${widgetID}'] =
     "${bootstrap.serialized?default("")?replace("\\","\\\\")?replace("\"","\\\"")}"
            </#if>
    </script>
    <#if bootstrap?has_content>
        <noscript>
        ${bootstrap.noscript}
        </noscript>
    </#if>
        <div id="gwt-slot-${widgetID}"></div>
        <div id="gwt-loading-${widgetID}" class="loading"><p>Loading...</p></div>
        <div id="preload"></div>
</#macro>
```

As you can see, this FreeMarker template expects the `bootstrap` object to have two specific properties: `serialized` and `noscript`. The `noscript` variable should have some HTML content appropriate for the search engines, and the `serialized` variable should be the GWT-RPC serialization of our `PostsList` object. That's pretty much it. We'll pass the `serialized` value over using our standard JavaScript dictionary technique. Then, we just cough up whatever the object has in its `noscript` field.

Let's make these two requirements official in Listing 12-3 by declaring what methods this object needs to have in an abstract parent object for all classes that we'd like to be able to serve as host page bootstrapping objects.

Listing 12-3. *src/main/java/com/apress/progwt/client/domain/dto/GWTBootstrapDTO.java*

```
public abstract class GWTBootstrapDTO {
    private transient GWTSerializer serializer;
    public GWTBootstrapDTO() {
    }
    public GWTBootstrapDTO(GWTSerializer serializer) {
        this.serializer = serializer;
    }
    public GWTSerializer getSerializer() {
        return serializer;
    }
    public abstract String getNoscript();
    public abstract String getSerialized() throws InfrastructureException;
}
```

This class is going to be the parent of everything that we want to be able to serialize. Listing 12-4 shows an explicit implementation of this class for the forums. We'll call it `ForumBootstrap`. Let's take a look at it.

Listing 12-4. *src/main/java/com/apress/progwt/client/domain/dto/ForumBootstrap.java*

```
package com.apress.progwt.client.domain.dto;
public class ForumBootstrap extends GWTBootstrapDTO implements
        Serializable {
    private ForumTopic forumTopic;
    private PostsList postsList;
    public ForumBootstrap() {} //default ctor for serialization compatibility
    public ForumBootstrap(GWTSerializer serializer, PostsList postsList,
            ForumTopic forumTopic) {
        super(serializer);
        this.forumTopic = forumTopic;
```

```
        this.postsList = postsList;
    }
    @Override
    public String getNoscript() {
        StringBuffer sb = new StringBuffer();
        for (ForumPost fp : postsList.getPosts()) {
            fp.appendNoscript(sb);
        }
        return sb.toString();
    }
    @Override
    public String getSerialized() throws InfrastructureException {
        return getSerializer()
                .serializeObject(this, ForumBootstrap.class);
    }
    //getters setters omitted
}
```

OK, this object serves our two purposes pretty explicitly. It carries the `forumTopic` and `postsList` fields so that the serializer will have things to serialize. The two methods it implements each serve to output these fields in the appropriate format. The `getNoscript()` method will be called by FreeMarker within the `<noscript>` tags, and the output from the call to `getSerialized()` will go into a JavaScript dictionary for passing to the client.

Now, we just need to see how to create this object. The only tricky bit will be figuring out how to get a `GWTSerializer`. Before this, we never had to worry about serializing things explicitly, since serialization happened automatically within the internals of the RPC methods. Now, we need to find a way to access that functionality by itself.

Reusing Serialization

Up until now, our GWT serialization has been occurring in our `GWTSpringControllerReplacement` class. This is the class that gets all our RPC requests forwarded to it from the dispatcher servlet. We'll keep that functionality, but it's not too much work to refactor this class to allow for explicit serialization as well. We'll mark this functionality with a new `GWTSerializer` interface and inject it into `CollegeController`. That way, our `CollegeController` will be able to create instances of the `ForumBootstrap` class that we defined in Listing 12-4. Listing 12-5 shows our changes to this class.

Listing 12-5. *src/main/java/com/apress/progwt/server/gwt/GWTSpringControllerReplacement.java*

```
package com.apress.progwt.server.gwt;
public class GWTSpringControllerReplacement extends RemoteServiceServlet
        implements ServletContextAware, Controller, RemoteService,
        GWTSerializer {
    private ServerSerializationStreamWriter1529 getWriter() {
        //creates serialization writer using 'serializeEverything' Policy
    }
    public String serializeObject(Object object, Class<?> clazz)
            throws InfrastructureException {
        ServerSerializationStreamWriter1529 serializer = getWriter();
        try {
            serializer.serializeValue(object, clazz);
        } catch (SerializationException e) {
            throw new InfrastructureException(e);
        }
        String bufferStr = "//OK" + serializer.toString();
        return bufferStr;
    }
}
```

That's all we need to do. You can see that we call `getWriter()` to get the serialization writer. This returns a writer that will have our `HibernateFilter` already attached to it, so we'll be sure to get the same serialization that we get for the rest of our domain. After that, we simply serialize and add the "//OK" string, which GWT-RPC adds to proclaim that this is not an error. Now, we should be able to get a serialized `String` representation of any objects that GWT can serialize. Let's see how we modified the `CollegeController` to take advantage of this and supply the model for our FreeMarker page.

Bootstrapping Controller

All we need to do here is use our injected serializer to create the `ForumBootstrap` object and add this into the model under the name "forumBootstrap", as shown in Listing 12-6.

Listing 12-6. *src/main/java/com/apress/progwt/server/web/controllers/CollegeController.java*

```
public class CollegeController extends BasicController {
    private GWTSerializer serializer; // injected
    protected ModelAndView handleRequestInternal(HttpServletRequest req,
            HttpServletResponse arg1) throws Exception {
```

```
        //skip school fetching
        PostsList forumPosts = schoolService.getForum(school, 0, 10);
        ForumBootstrap forumBootstrap = new ForumBootstrap(serializer,
                forumPosts, school);
        model.put("forumBootstrap", forumBootstrap);

        model.put("school", school);
        model.put("interestedIn", schoolService
                .getUsersInterestedIn(school));

        ModelAndView mav = getMav();
        mav.addAllObjects(model);
        return mav;
    }
```

You can see that we've added a call to `schoolService.getForum()`. This is going to be the replacement for the call that used to happen on the GWT side. This call, however, won't go to the GWT service; it will just go to the regular service layer. With this data in hand, we'll create our bootstrap object and add it to the model.

That's it for the server. The last thing I should mention is what we're injecting for the `GWTSerializer`. Remember that the `GWTSpringControllerReplacement` class is an abstract class, so we're not injecting that. What are the instances? Why our `GWTSchoolServiceImpl` class, of course. That's the class that implements `GWTSerializer` and will have the serialization functionality.

GWT Startup with Bootstrapping

On the client side, let's see what our host page looks like now that we've included the serialized output of our `ForumBootstrap` object and the `noscript` tags (see Listing 12-7).

Listing 12-7. *Sample HTML Output for college.html*

```
<script language="JavaScript">
            Vars['widgetCount']= "1"
            Vars['widget_1'] = "Forum"
            Vars['serialized_1'] = "//OK[100,0,0,24,63,6…
</script>
    <noscript>
Title: Darmtouth is great<br>
Post: I really like it!<br>
Topic: School:486:name:Dartmouth College<br>
Author: |1:test|<p>
```

```
Title: thread 10<br>
//etc
    </noscript>
```

The major change is the new `serialized_1` entry in our JavaScript dictionary (which is a much longer string than is shown here). This is what we're going to turn back into an object when we use the GWT client deserializer. Let's see how that happens.

Recall that all our GWT applications extend `GWTApp`, which gives them all access to this common functionality. Listing 12-8 shows what additions we can make to `GWTApp` to allow its children apps to deserialize the `serialized_1` entry.

Listing 12-8. *src/main/java/com/apress/progwt/client/GWTApp.java*

```
protected Object getBootstrapped() {
    return getBootstrapped("serialized");
}
private Object getBootstrapped(String name) {
    String serialized = getParam(name);
    if (serialized == null) {
        Log.warn("No param " + name);
        return null;
    }

    try {
        ClientSerializationStreamReader c = getBootstrapService()
                .createStreamReader(serialized);
        Object o = c.readObject();
        return o;
    } catch (SerializationException e) {
        Log.error("Bootstrap " + name + " Problem ", e);
        return null;
    }
}
private RemoteServiceProxy getBootstrapService() {
    return (RemoteServiceProxy) getSchoolService();
}
```

In Listing 12-8, you can see that we start off by using the `getParam()` method that we'd previously developed to get the `serialization_1` entry out of the JavaScript dictionary. Once we've got this string, how do we serialize it? All we need to do is get a `ClientSerializationStreamReader` for the serialized string and call `readObject()` on it.

Of course, the trick is understanding how to create a `ClientSerializationStreamReader` in the first place.

Luckily, we can get one of these readers from any `RemoteServiceProxy`. This is the abstract class that is extended when we create our RPC service classes with `GWT.create()`. For that reason, we can just cast our `GWTSchoolServiceAsync` into a `RemoteServiceProxy`, and we'll be just about set. We've got all the right connections here to set up our serialization.

There's one thing we're missing, however, and it's a bit subtle. The call to `GWT.create()` is not just a simple constructor. What happens underneath the covers is really a bit complicated and goes to the heart of how GWT generators work. For our purposes, it's enough to understand that the `GWTSchoolServiceAsync` class returned from this method will have special deserializer methods created for it when it's compiled to JavaScript. It will actually have a deserializer method created for every type of class that it's going to expect to deserialize. This optimization saves us from having to create deserializers for classes that will never be serialized. The way the GWT compiler figures out what deserializers need to be created is by looking at the interface that this class extends, in this case `GWTSchoolService`. Unfortunately, this means that there won't be a deserializer created for `ForumBootstrap`, since no methods in that interface return a `ForumBootstrap`. Luckily, all we need to do is add a dummy method like the following one to that interface (and the asynchronous interface and the implementation):

```
public interface GWTSchoolService extends RemoteService {
    ForumBootstrap forumBootstrapDummy();
}
```

This method won't be used normally, but it will cause the GWT compiler to create a deserializer for `GWTSchoolServiceAsync`; that means we'll be able to avoid a nasty runtime deserializaiton error that we'd get without this. Bottom line: if you're going to serialize things explicitly, make sure that the RPC service serializes them as well.

With that last little bit out of the way, let's look at how the `ForumApp` class uses the bootstrapped variable to avoid the asynchronous load (see Listing 12-9).

Listing 12-9. *src/main/java/com/apress/progwt/client/forum/ForumApp.java*

```
public ForumApp(int pageID) {
        super(pageID);
        initServices();
        String uniqueForumID = getParam("uniqueForumID");
        ForumBootstrap bootstrapped = (ForumBootstrap) getBootstrapped();
        //skipped GUI creation
        if (bootstrapped != null) {
            Log.info("Running off Bootstrap");
            load(0, bootstrapped.getPostsList(), false, bootstrapped
```

```
                .getForumTopic(), FORUM_POST_MAX);
            History.newItem(bootstrapped.getForumTopic()
                .getUniqueForumID());

        } else if (initToken.length() > 0) {
        //skipped
        }
        History.addHistoryListener(this);
    }
```

Not bad. We do need to cast the result of the `getBootstrapped()` call, but after that, we've got a regular old domain object, transferred from the server to the client using GWT serialization but going through JavaScript instead of over RPC.

The Proof in the Pudding

So did it work? Did we achieve our goal of better search engine visibility? What does this all look like? Let's take a look at the new college page in Lynx in Figure 12-3 and see what Lynx does with the elements that we outputted to the `<noscript>` element.

Perfect! We've found a way to output HTML so that text readers and bots can easily understand what information is contained within the page. What's better is that we've really minimized the impact that this has on our site design. Finally, we've also developed a technique for passing complicated objects to GWT directly within the page. Applying this technique to other areas of the site will speed up perceived GWT load time in many places, because our widgets will be able to avoid making an asynchronous request as they load.

With this approach to enhancing the way ToCollege.net is indexed, we should feel confident that the pages that are crawled can be well indexed, which, of course, relies on the assumption that our pages can be crawled. Let's take a look at that next.

Figure 12-3. *Our bootstrapped page with content in <noscript> tags*

Optimizing the Crawling of ToCollege.net

The key to a bot's ability to crawl our site is having good old-fashioned links on our pages that allow the crawler to branch out from the index page to all the other pages on the site. Our goal with our SEO initiative is to allow our school forums to be indexed by search

engines. We succeeded in this with the pages themselves, but now it's time to make sure that the Googlebot can get to those pages.

To make sure that we have links to all pages, we're going to use something like the secondary site approach that I described previously. The approach still has a small disadvantage in that we're displaying content that we don't really expect our user's to use, but those effects are pretty minimal in this context. Why don't we expect users to use this? Well, they could search alphabetically but, in general, we expect that they'll just use the site capabilities. There's really no harm in having this functionality, however, and the benefit to search crawlers is huge. Figure 12-4 shows what this page will look like when we're finished with it.

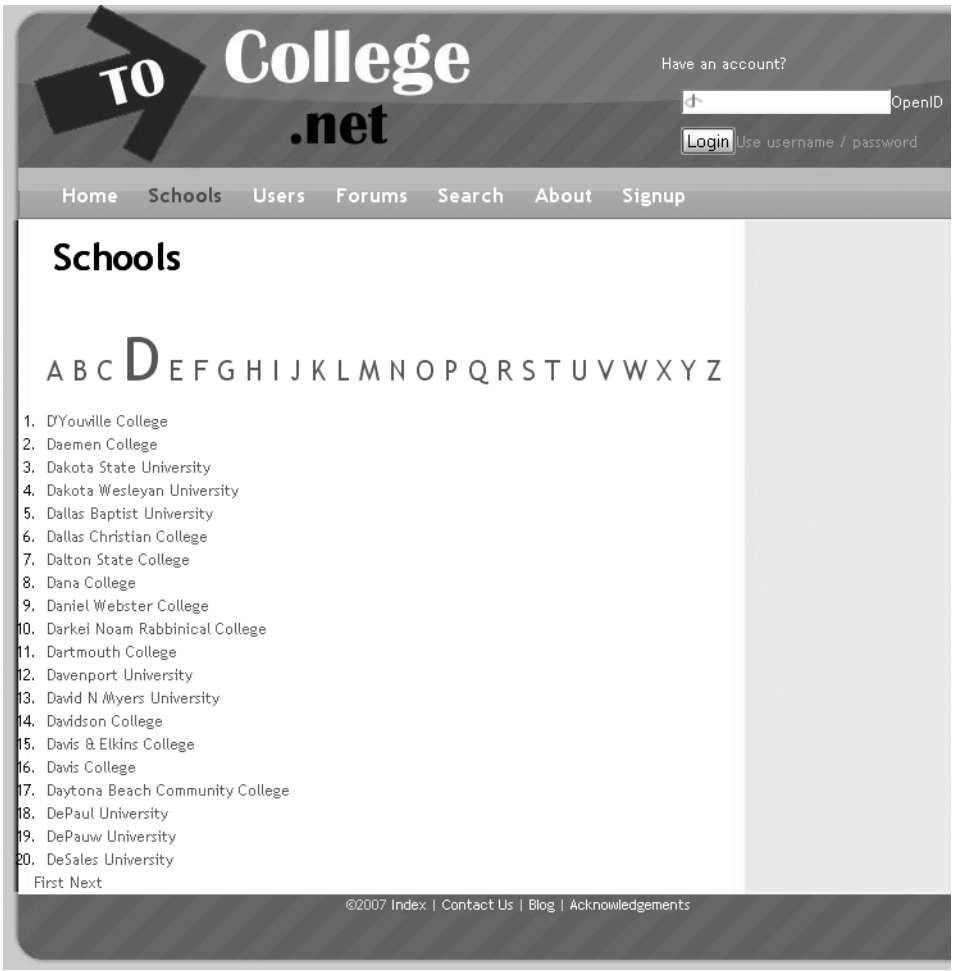

Figure 12-4. *An alphabetical school index designed to ensure that all schools will be found*

As I said, this is a really simple index of all the schools that our site tracks. Each school will be a link to the school-specific page (and the school-specific forum gold that lies within!). Listing 12-10 shows the addition to our `SimpleAnnotatedController` that we'll need to make to enable this page.

Listing 12-10. *src/main/java/com/apress/progwt/server/web/controllers/SimpleAnnotatedController.java*

```
@Controller
public class SimpleAnnotatedController {
@RequestMapping("/schools.html")
    public ModelMap schoolsHandler(HttpServletRequest req,
            @RequestParam(value = "startLetter", required = false)
            String startLetter,
            @RequestParam(value = "start", required = false)
            Integer start) {
        ModelMap rtn = ControllerUtil.getModelMap(req, userService);
        if (start == null) {
            start = 0;
        }
        List<School> schools = null;
        if (startLetter == null) {
            startLetter = "";
            schools = schoolService.getTopSchools(start, 20);
        } else {
            schools = schoolService.getSchoolsStarting(startLetter,
                    start, 20);
        }
        rtn.addAttribute("start", start);
        rtn.addAttribute("startLetter", startLetter);
        rtn.addAttribute("schools", schools);
        return rtn;
    }
```

Excellent, Spring MVC annotations come through strongly for us here again, allowing us to easily set up a controller with no extra XML configuration. You can see that we also have two request parameters for this controller. These will be pulled off the URL bar and will enable us to page results and sort the schools alphabetically. We'll add the request parameters back into the model along with the list of schools that we retrieve from the database. Listing 12-11 shows the template that renders this output.

Listing 12-11. *src/main/webapp/WEB-INF/freemarker/schools.html*

```
<html>
//std header
<body id="schools">
<h1>Schools</h1>
      <div id="main">
        <#assign letters = ["A","B","C","D","E","F","G","H","I","J","K","L","M","N",
"O","P","Q","R","S","T","U","V","W","X","Y","Z"]>
        <ul class="letterSelector">
         <#list letters as letter>
                <li <#if letter=startLetter>class="selected"</#if>>
                <a href="<@spring.url "/site/schools.html?startLetter=${letter}"/>">
                        ${letter}</a></li>
        </#list>
        </ul>
        <p>
        <ol start=${start + 1}>
         <#list schools as school>
            <li><@common.schoolLink school/></li>
         </#list>
        </ol>
        <a href="<@spring.url
          "/site/schools.html?startLetter=${startLetter}&start=0"/>">First</a>
        <#if start gt 20>
             <a href="<@spring.url
             "/site/schools.html?startLetter=${startLetter}&start=${start - 20}"/>">
             Prev</a>
        </#if>
             <a href="<@spring.url
             "/site/schools.html?startLetter=${startLetter}&start=${start + 20}"/>">
             Next</a>
    </div>
</body>
</html>
```

That's it. First, we style our selected `<li>` element differently to highlight the selected letter. Next, we loop over the letters and create links to the various school pages for each letter in the alphabet. We then add some simple navigation controls using the request parameter and some simple math operations to decide whether the previous and next buttons are necessary.

With this page in place, we've essentially created a dynamic index of all of the college pages on our site. Now, any search engine spider that comes to the front page will be able to easily hop to the school-specific pages using standard HTML links. We're free to use JavaScript links on the rest of the site without affecting the bots' ability to crawl.

Robots.txt

The last search-related thing that we're going to cover in this chapter is the `Robots.txt` file for ToCollege.net. We won't explore this file in detail, but it's helpful to have a quick description of the file and its purpose. For more details, check out `http://www.robotstxt.org/` and `http://en.wikipedia.org/wiki/Robots.txt`. Basically, this file is something that crawler robots should read before crawling your site, and its main purpose is to prevent robots from crawling and indexing certain directories on your site.

■ **Caution** Using `Robots.txt` is *not* a way to prevent access to your site. There's absolutely nothing stopping a bad robot (or any other user) from ignoring this file. Authorization should be achieved through other means, such as those discussed in Chapter 5.

Why would you want to limit robot access? Well, some canonical examples are to disallow the indexing of publicly available `/cgi-bin` or `/images` directories. In our case, the only files we'd like to keep out of the search engine are some of our search pages. Why do that? Well, the problem with search pages is that the results are inherently transient. Keeping crawlers out of them will help us avoid the all too common occurrence of clicking a Google link that lands us on a page that no longer contains what we were interested in. Other examples of this might be What's Popular or What's New pages. It's much better to have these crawled and indexed with their final resting URLs.

■ **Note** If you allow users to post links on your site, you'll also want to look into the `nofollow` attribute for links, which prevents certain types of search engine spam. See `http://en.wikipedia.org/wiki/Nofollow` for an overview.

Listing 12-12 shows the ToCollege.net `Robots.txt` file.

Listing 12-12. *src/main/webapp/robots.txt*

```
User-agent: *
Disallow: /site/search
Crawl-delay: 5         # non-standard
Request-rate: 1/5      # extended standard
Visit-time: 0100-0645  # extended standard
```

Again, this is pretty simple. You can see that we've disallowed search results from being indexed. We've also used a couple of extensions to the `Robots.txt` format in an attempt to get a little bit more control over when the robots come to visit. See `http://en.wikipedia.org/wiki/Robots.txt#Extended_Standard` and `http://en.wikipedia.org/wiki/Robots.txt#Nonstandard_extensions` for details on these extensions. They're not supported by all robots, but including them can't hurt.

The Google web master tools that I mentioned before are a great way to make sure that you've got this file configured properly. This tool will let you know if it's found the `Robots.txt` file (it must be in the root directory), and you'll be able to type in URLs to see if they'll be blocked.

Summary

Well, that's all the time we have for ToCollege.net search optimization. In this chapter, we looked at crawling and indexing, the two elements of searching that we have control over. We looked at the dangers of AJAX applications and rich media applications in general with regards to SEO. We found a great way to make our GWT application more search-engine friendly by cleanly inserting data into a `<noscript>` field on our page. In so doing, we also developed a fantastic new approach to getting data into our GWT application without using RPC. Finally, we looked at increasing the ability of a Googlebot to crawl our site by adding an alphabetical school listing.

The next steps for us as webmasters would be to register for the Google webmaster tools mentioned in this chapter and to continue to browse our site with Lynx to make sure that we are including alternate text for all our images and generally being as accessible as possible. After that, all we need to do is spread the word and get people to link to our site so that our PageRank goes right through the roof!

OK, it's time to turn our focus back to the user experience. In the next chapter, we're going to take a look at what Google Gears can do for our site. We'll take a look at how we might create an offline mode for ToCollege.net and how to get amazing browser-side caching support with a Google Gears database.