Extracted from:

Rails, Angular, Postgres, and Bootstrap

Powerful, Effective, and Efficient Full-Stack Web Development

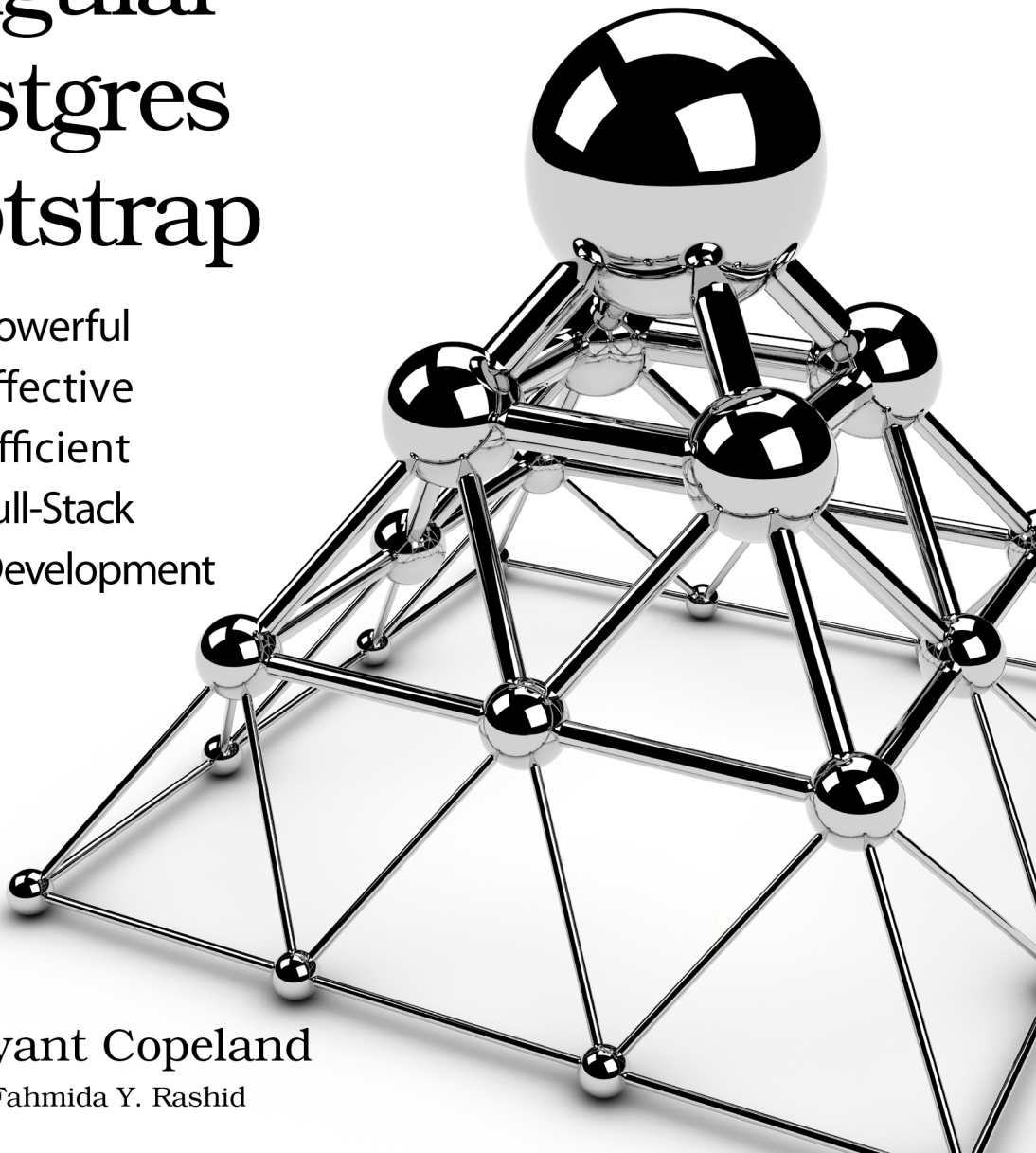
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Rails Angular Postgres Bootstrap

Powerful
Effective
Efficient
Full-Stack
Web Development



David Bryant Copeland
edited by Fahmida Y. Rashid

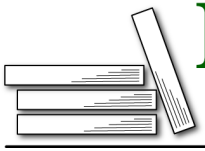
Rails, Angular, Postgres, and Bootstrap

Powerful, Effective, and Efficient Full-Stack Web Development

David Bryant Copeland

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *https://pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-126-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—July 29, 2015

Paginating the Results using Bootstrap's Components

Adding pagination can be done in just two steps: adjusting the query to find the right "page", and adding pagination controls to the view. There are several RubyGems out there that can help us, but it's actually not that much code to just do it ourselves. Since we'll be porting our view over to Angular in the next chapter, there's little benefit to integrating a gem at this point.

We'll take it one step at a time. First, we'll adjust the controller to handle pagination.

Handling Pagination in the Controller

For simplicity, we'll hard-code the size of a page to 10 results, and look for a new parameter, :page that indicates which page the user wants, with a default of 0.

search/pagination/shine/app/controllers/customers_controller.rb

```
class CustomersController < ApplicationController
  PAGE_SIZE = 10

  def index
    @page = (params[:page] || 0).to_i

    # ...

  end
end
```

Next, we'll use both PAGE_SIZE and @page to construct parameters to ActiveRecord's offset and limit methods. Since our results are sorted, we can rely on these two methods to allow us to reliably page through the results without the order changing between pages.

search/pagination/shine/app/controllers/customers_controller.rb

```
@customers = Customer.where(
    customer_search_term.where_clause,
    customer_search_term.where_args).
  order(customer_search_term.order).
➤   offset(PAGE_SIZE * @page).limit(PAGE_SIZE)
```

That's all there is to our controller. Now, we'll adjust the view to allow paging.

Adding Pagination Controls to the View

To keep things simple, we'll go with a previous/next style of pagination. This means we'll need two links on the page, which we can create by adding or

subtracting 1 to @page and passing that to the Rails-provided customers_path helper.

To style the links, Bootstrap provides a component we can use, called a *pager*. Let's set it up in a partial, which we'll then use to place the pager before *and* after the results list (this allows the user to always have the pager handy). We've highlighted the markup and classes Bootstrap requires to style the pager. Pay special attention to disabled, which will give our Previous button a disabled look if we're on the first page.

search/pagination/shine/app/views/customers/_pager.html.erb

```
<nav>
  <ul class="pager">
    <li class="previous <%= page == 0 ? 'disabled' : '' %>">
      <%= link_to "&larr; Previous".html_safe,
                  customers_path(keywords: keywords, page: page - 1) %>
    </li>
    <li class="next">
      <%= link_to "Next &rarr;".html_safe,
                  customers_path(keywords: keywords, page: page + 1) %>
    </li>
  </ul>
</nav>
```

Now, we'll include the partial in app/views/customers/index.html.erb.

search/pagination/shine/app/views/customers/index.html.erb

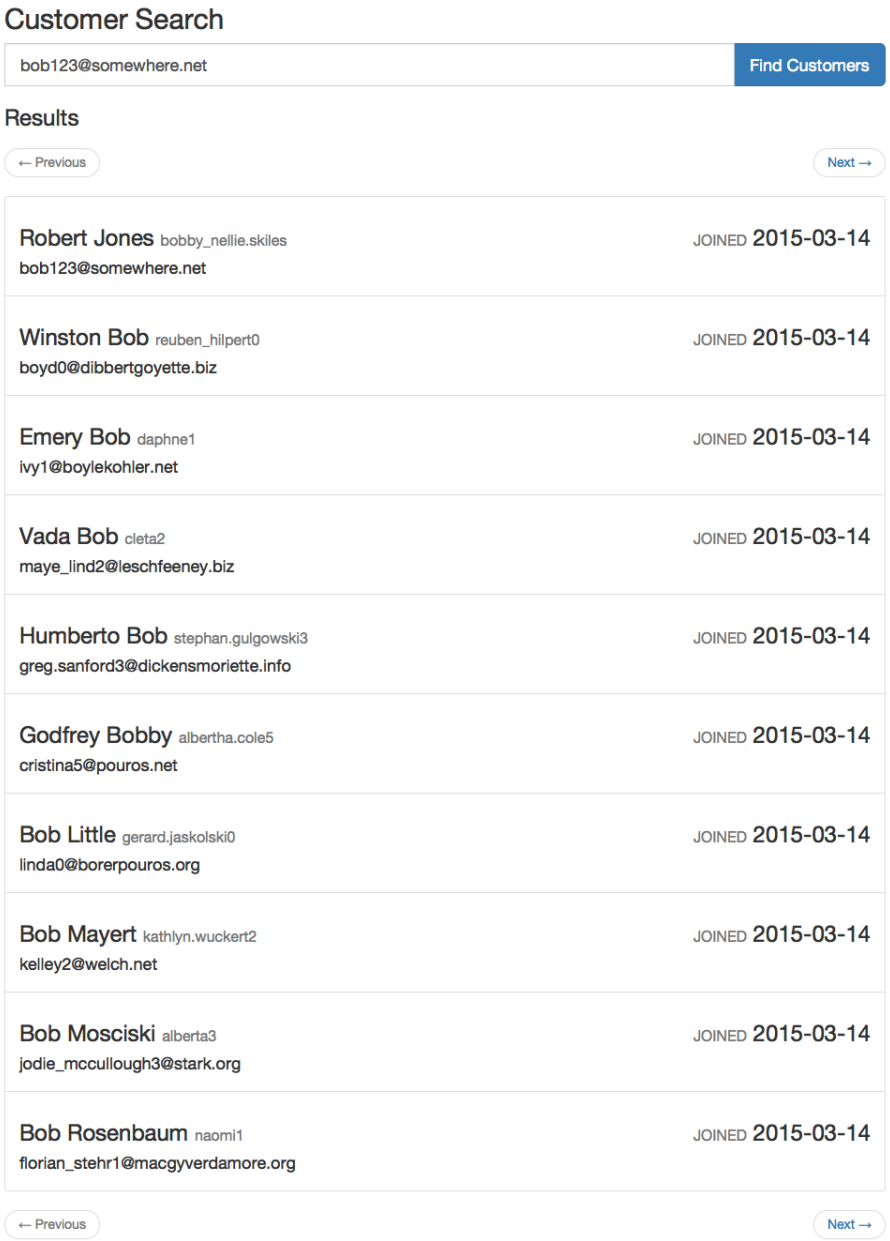
```
<section class="search-results">
  <header>
    <h1 class="h3">Results</h1>
  </header>
  <%= render partial: "pager",
             locals: { keywords: @keywords, page: @page } %>

  <ol class="list-group">

    <!-- ... -->

  </ol>
  <%= render partial: "pager",
             locals: { keywords: @keywords, page: @page } %>
</section>
```

If we start our server and search, we'll now only see 10 results, and our pager controllers work to allow us to step through them.



Because of our indexes, Postgres' powerful implementation of `order by`, and Bootstrap's pre-made components, we were able to add performant pagination in just a few lines of code.