WHAM! Packaging

Original Google doc by John Skiles Skinner

Packaging our custom library catalog autocomplete feature

John Skiles Skinner

Updated 29 June 2020

Introduction

This document investigates how we could distribute a production build of a catalog autocomplete feature based upon a prototype autocomplete feature that Tim Worrall built in a fork of Cornell's library catalog app.

Distribution via Blacklight

I was previously under the misapprehension that Cornell's library catalog uses a customized version of Blacklight. For anyone who made the same mistake as me: the Blacklight gem Cornell uses is the standard one used by other institutions. This should be distinguished from Cornell's catalog app, which employs the Blacklight gem, and which is a custom Cornell product.

Because Blacklight is used by a community, adding our feature to Blacklight presents a way to distribute our code. Further, Blacklight already contains an autocomplete feature; it seems possible that we could package our autocomplete as an improvement on this existing feature, reusing parts of it.

Because Cornell's catalog app does not use the autocomplete feature, the investigation in this document uses the Blacklight demo app, a generic Rails app that has the autocomplete feature turned on. I pointed the demo app to a local copy of the Blacklight gem.

Turning autocomplete on

These two configuration settings appear in the Ruby code of the demo app or any catalog app that uses the Blacklight autocomplete feature:

in catalog_controller.rb

config.autocomplete_enabled = true

config.autocomplete_path = 'suggest'

The autocomplete_enabled variable turns the autocomplete feature on. Its value is read both by 1.) Blacklight itself, and 2.) an app using the feature, such as the blacklight demo app (but not by Cornell's Blacklight app, which does not yet use autocomplete).

The autocomplete_path variable specifies 1.) the name of a Solr request handler that Blacklight will ask for suggestions, and 2.) the name of a particular key that Blacklight expects in JSON returned by Solr.

How autocomplete works

When autocomplete is switched on in an app that uses Blacklight, the Blacklight gem provides an API at the path appname/catalog/suggest. Blacklight also provides JavaScript that connects to this API, requests and receives suggestions from it, and makes the suggestions available to the user.

In particular, this line of Ruby in Blacklight selects parts of the JSON response that are presumed to hold suggestions:

(response.try(:[], suggest_path).try(:[], suggester_name).try(:[], request_params[:q]).try(:[], 'suggestions') || []).uniq

This navigation of JSON is configurable to some extent, and to some extent contains baked-in assumptions. The key designated suggester_name is configured as the variable autocomplete_suggester in the configuration.rb file in Blacklight. The suggest_path key is fixed not within Blacklight, but in the catalog app; its value is the autocomplete_path setting previously mentioned — meaning that it shares a name with the Solr request handler. Finally, the su ggestions key is hard-coded, and not configurable.

A further level of selection on the suggestion data is done in this JavaScript file that makes suggestions available to the catalog user. The JS also contains a hard-coded JSON key name assumption.

JSON data that does not conform to the assumptions contained in the above code will not be parsed correctly by Blacklight's autocomplete feature. This means that Blacklight will only pull suggestions from Solr if Solr returns documents according to these expectations.

Interactions with Solr

Blacklight's autocomplete feature pipes user-entered strings to a Solr index that is presumed to exist and to be able to provide suggestions in a certain format. It then pipes this suggestion data back to the user interface.

As far as I can tell, Blacklight apps are built to only hook to one Solr index and collection. The URL of this Solr is specified in the .env file in the SOLR_URL variable. Autocomplete suggestions are assumed to come from the same Solr index and collection as all other Solr catalog functions.

The autocomplete feature is built to hook to the Solr suggester component. The path of the Solr request handler that the feature connects to is configurable, but it appears to be common that the path used is /suggest because this is the default name of the suggester component request handler endpoint.

This presents two problems for the Solr service that Huda and Tim have been working on:

1. The service is separate from the main Solr library collection. Blacklight is not built to point at a second Solr collection and index name.

2. The service uses Solr's /select request handler, rather than /suggest, and accordingly returns a different JSON structure with key names that don't match Blacklight's assumptions.

Directions in addressing this problem

The mismatch in expectations between Blacklight's autocomplete code and our Solr service's return values can be addressed by one or more of:

1. Modifying Blacklight to have different expectations, perhaps by creating a gem that institutions can elect to apply to their Blacklight app, or by forking Blacklight, or
2. Modifying our use of Solr to conform to Blacklight's expectations, possibly by switching to the use of the Solr /suggest endpoint, or
3. Abandoning the idea of distributing the autocomplete feature in connection with the existing autocomplete feature in Blacklight, perhaps instead keeping our code located in the catalog app rather than in Blacklight

Because I have been assigned to investigate the packaging part of this feature, and I haven't been working enough on Solr to know what's possible, possibility #2 is of particular curiosity to me. I wonder if the work that Huda and Tim have done can be changed to conform to the expectations of the /suggest request handler? And if they can be presented to Blacklight as a part of the Solr server that the rest of the catalog uses? If so, we may be able to avoid modifying Blacklight altogether.

The North Carolina solution

Four institutions in North Carolina — Duke U, UNC, NC State, and NCCU — collaborated to build a Rails engine, called Argon, that includes an autocomplete solution for their library catalogs. Argon uses a few tricks to modify Blacklight's behavior without modifying Blacklight's code.

Like Blacklight itself, Argon is a Ruby gem that is meant to be included in a library catalog application's Gemfile. Argon defines modules, classes, methods, and JavaScript files with the same names as corresponding entities in Blacklight. These definitions are intended to override parts of Blacklight. Argon omits a method called isolate_namespace normally used by Rails engines to prevent just the kind of overriding that Argon relies upon to modify Blacklight. Administrators must list Argon *after* Blacklight in the catalog app's Gemfile so that order of operations favors Argon's definitions over Blacklight's.

In particular, Argon overrides the Ruby code that initializes and parses Solr responses, and the code that assembles query parameters and pathnames for Solr queries. Argon also replaces the JavaScript that calls the API and displays data to users. Argon uses /suggest request handlers, just as Blacklight autocomplete does by default, except that it uses four of them rather than one: three specialised handlers for subject, author, and title, plus a general one.

It seems that overriding techniques used by Argon could be extended to cause Blacklight to accommodate a /select request handler, if that's what our team finalizes upon. Argon's approach to modifying Blacklight's behavior seems to me a promising path forward in making our own autocomplete feature work with Blacklight.

End product

This project ran short of time. Rather than truly override Blacklight's existing functionality with an improvement, I packaged my team's proof of concept into an engine. This engine, which we named *Nectarguide*, is meant to be used with Blacklight's existing autocomplete / autosuggest featured switched off.

Code repository for Nectarguide

A video demo of Nectarguide in action

A screenshot of Nectarguide providing search suggestions: