DIY API

A Sierra-Based, Homegrown Catalog API
Hi there! My name is Jason Thomale, and I'm the Resource Discovery Systems Librarian
at the University of North Texas. And, over the course of the next hour, I'm going to introduce you
to a project I began about two and a half years ago
DIY Catalog API to build a local API for our Sierra catalog data.
Right at the outset, I wanted to offer a calming word of reassurance about the technical nature of this topic. I'm part librarian and part software developer, and I also support our ILS. So I have one foot in the world of
developers and the other in the world
of the librarians and library staff who have to use the software. One of my goals is to present this in a way that will be informative and comprehensible for both groups, so please don't sweat it.
if you aren't technically inclined. Also, I'm going to start a little basic and then get into more depth as I go, so if you're a developer type, please don't get up and leave when I start explaining what APIs are and what REST is here in a minute. It will get more interesting for you shortly, hopefully. Or,
at the very least, you can just be amused by my attempts to explain things and then tell me afterward how wrong I am.
So here's the agenda.
First I'll describe a little about what the project is,
Agenda

What it is
Why I built it

and then I'll talk about why I built it.
I'll talk a bit about how I built it and how it works.
Agenda

What it is
Why I built it
How it works
The apps

And we'll look at a few of the applications we've built with it so far.
My Goal

Ultimately, my goal is
My Goal

Useful for you?

to help you see if this might be something useful for you to deploy at your own institution.
On GitHub

http://github.com/unt-libraries/catalog-api

The code is up on GitHub, and it's open source. So you can download it, use it, and contribute back, if you want. And, even if what I've built isn't useful to you, maybe this will trigger some ideas for things that you could do on your own.
So first up, I want to talk about what this is and what it's all about.
In a nutshell, the overarching goal is to provide
a REST API layer for Sierra that is under local control and therefore completely customizable. Okay? Now, let's talk about what this actually means--I mean, "REST" and "API" are both pretty high on the buzzword spectrum. Let's start with the noun in this phrase,
"API." API stands for
Application Programming Interface. In computer terms,
an interface is a
boundary between two different components of a computer system that they
System Component A

System Component B

share and use to
exchange information. Interfaces are extremely prevalent in computer system design. You have everything from
interfaces between different hardware components that let them communicate with each other,
to interfaces that allow hardware components to communicate with software,
Device Driver

Operating System

to interfaces in software
Application A

Operating System

at
pretty much
Application Module A

Application Module B

every level
that allow different programs to communicate with one another. So, generally speaking, an interface between applications or between application components is called
an API. Let's compare that with a type of interface that we're all more familiar with,
user interfaces. UIs are interfaces that allow people to communicate and interact with systems. And I like to draw an analogy
between UIs and APIs, because there are certain qualities that we more readily associate with UIs that I think help explain APIs better, especially to non-programmers. So,
a user interface is
Designed to get input from humans,
Designed to get input from humans, give feedback to humans, and to give feedback to humans,
Designed to get input from humans, give feedback to humans, and help humans complete particular tasks.

and it's designed to help them use the system to complete particular tasks. Likewise,
an API defines
API

Designed to get input from programs,

how input from other programs is taken
API

Designed to get input from programs, give feedback to programs, and how feedback or output is delivered,
API

Designed to get input from programs, give feedback to programs, and help programs complete particular tasks with the idea that it helps another program use the system to complete certain tasks.
So whereas a UI will use
Standard interaction design conventions,
UI

Standard interaction design conventions, visual and

visual and
Standard interaction design conventions, visual and auditory cues

auditory cues to help make it useful for people,
an API will use
API

Standard architectures,
API

Standard architectures, standard data formats,
API

Standard architectures, standard data formats, standard protocols

and protocols to help make it useful both for other programs AND for the people writing those programs.
For example, here’s the UI in Sierra that lets someone perform a title search and see what bib records in the catalog match their query. It’s an amalgam of common UI design conventions, and so it’s not difficult for us, as human beings, to look at it and understand how it works and what it does.
Here's a similar search done via our catalog API, which I'm sure isn't very comprehensible to the average person but uses standard machine-actionable protocols like HTTP combined with other standards
like URLs and JSON, which makes an interface that is easy to write programs that "understand" how to use it.

So the thing I want you to get from this long rambly explanation is
Interface Design

that interface design
isn't just for user interfaces. Just like good UIs, good APIs follow certain conventions and are designed to be used in particular ways and to be useful for particular purposes. Often it seems that system vendors
It Has APIs

use APIs as a buzzword and a selling point--but it's important to understand that
“It Has APIs”
Good for you!

the real value of an API
“It Has APIs”

Good for you!
What do they let me do?

lies in the specifics of what that API is designed to help you or your programmers do and how well that actually matches up with your goals and needs. And the answer to that all-important question,
what can I do with your APIs, depends on at least a couple of factors. One of these of course is what type of API it is.
So, our API is a REST API, just like III's Sierra API.
REST stands for
REpresentational State Transfer. The term
REpresentational State Transfer;
Coined by Roy T. Fielding;

was coined in a dissertation written by Roy T. Fielding in 2000, used to describe
REpresentational State Transfer; Coined by Roy T. Fielding; Describes the architectural style of the Web

the architectural style of the Web. It's
the set of architectural constraints that makes the Web the hugely distributed yet inherently simple, scalable, performant, and reliable system that we've all come to use daily. Because we're familiar with the Web, REST should feel pretty familiar.
REST Constraints

Client/server

It uses a client server/architecture. Our Web browser (client) interacts with the Web by connecting to Web servers.
It's stateless. Each request your browser makes to a Web server is completely independent from previous requests. The server doesn't store session state over multiple requests; your browser is responsible for that and stores things like your history and cookies.
Responses from a Web server can be cached to improve scalability and performance.
Intermediary servers can exist between the client and the server providing the content, in order to help with things like security and scalability, and the client shouldn't have to know whether or not it's connecting to the end server.
And, last, a REST system provides a uniform interface.
And this is what's most at play with a REST API. On the Web, this interface looks something like this:
you have resources that are identified
Uniform Interface

Resources identified by URIs at URLs

by URIs located at URLs, and sending a request for a resource at a particular URL will return
Uniform Interface

Resources identified by URIs at URLs that give a resource representation (HTML, XML, JSON)

a representation of that resource, which your browser then renders for you.
The resource representation itself contains data that defines what further actions you can take from that resource, like links to other resources or controls for forms that let you manipulate that resource. Now--
in the years since Dr. Fielding's dissertation, the REST style has been widely adopted (and some might say coopted)
“REST” API

by Web services that provide APIs for working with non-HTML data over the Web. And since REST is a style and not a practical implementation, it has been bastardized to some extent, which means that
“REST” API

Practices vary

certain REST practices will vary from API to API. But, generally speaking, REST APIs will tend to piggyback on the Web itself, using the existing infrastructure of the Web to
expose resources
at particular URIs,
where different resources represent different entities within a particular system or domain. And these resources will be
interlinked with one another based on their relationships, depending on the domain.
A client application can then interact with a resource
by sending HTTP requests
that identify a resource by URI, and what the API returns
is a representation of that resource that carries
the current state of the resource
and all possible links from that resource to other resources within that system.
This architecture allows you to build very lightweight apps that interact with your underlying system (including the ability to change the state of data within that system) just by sending HTTP requests, receiving data, and interacting with resources in a consistent standardized way,
just like the Web. Right? In theory, anyway. So, as you can see, saying that
Customizable REST API

Something is a REST API does tell you a bit about how the API probably works and how it's meant to be used, but it
What Can I Do With Your APIs?

still doesn't fully answer this question. There's one other important factor, which depends on
the underlying system *behind* the API--the thing that the API is providing an interface for. What is that designed to do?
And that's where the customizable part of this comes in.
The Sierra API built by III is an interface for
Sierra. As we know, Sierra is an Integrated Library System. And, as such, it has its own particular functions.
Like creating and managing records for items, bibs, authorities, patrons, etc.
Exporting data in batch to other systems.
Managing acquisitions.
Checking items in and out.
Managing holds. And so on.
And the Sierra API, even as it's still being developed, seems very much geared toward facilitating those types functions. Which of course absolutely makes perfect sense--III shouldn't focus building their API to do things that Sierra itself isn't designed to do. But, in libraries, we often think of our
catalog kind of as this entity that's separate from our ILS, right? Even though it's generally contained within the ILS. And so there are things maybe we want to be able to do with our **catalog** data that Sierra doesn't really support—things that aren't and likely won't be facilitated by the Sierra API because they don't really pertain to the Sierra system.
And so that's a big impetus behind this project. It isn't an API for Sierra. It lets you extract data from Sierra, transform it any way you like, and then store that in one or more Solr instances that are completely independent from Sierra. And those Solr instances are what actually feed the API, so when you request API resources, you're querying Solr, not Sierra. (And this provides a number of advantages--and at least one disadvantage--that we'll talk about in a little bit.)
So--while, out of the box, it does include a basic (partial) API implementation, which contains several resources we've created so far to support applications we've built--beyond that basic implementation,
it provides a toolkit that allows you to turn any of the data exposed via the Sierra database views (and even data from other sources) into your own REST resources.
And this leads me now into "why I built it," a question I want to delve into in a lot more detail. One of my fears all along has been that I'm
just reinventing the wheel. I mean,
there's the Sierra API--and, though I've already talked a little bit about why you might want your own local API, I realize that still may not be terribly clear.
And there’s also the SQL access to the Sierra database. These two things are supposed to allow customers to build their own applications on top of Sierra, which I know many of you are already doing.
So--how is this project not just reinventing the wheel? I want to answer this question
Story time

with a story—the story of how the project came to be. It starts with
Inventory

an inventory project, as many good stories do. Well--really, "how to do inventory" is kind of a mundane topic, but it does come up on III listservs frequently enough. (And I also know there's another DIY API presentation at this year's conference that involves doing inventory…) So, historically we've done inventory in kind of an
Historically ad hoc by department ...

ad hoc fashion. Our library is fairly distributed, with several specialized departments (like a music library, media library, science library, etc.) who have all been responsible for managing what's in their stacks. As you can imagine, our central Collection Management Division has not been very pleased with the fact that this is so ad hoc, and they have been growing increasingly displeased over recent years as our collections have been shifting around more often.
There must be a better way, right? Well, as I'm sure you know, the
Historically ad hoc by department ... 
There must be a better way!

III's Inventory Product?

typical method for doing inventory in Millennium or Sierra involves using III’s inventory product—which we don't have. As I understand it, this product lets you scan barcodes and import that list straight into a review file, at which point you can then update fields like "inventory date" using global update. Pretty easy, right? Well, for us,
Inventory

Historically ad hoc by department ... There must be a better way!

III's Inventory Product?

this didn't seem like a great option. That's because the items on our shelves
look like this. If there were a barcode on the spine, then scanning would make a lot of sense. But the barcode isn't on the spine,
it's inside the front cover. So in order to scan an item, you have to pull it off the shelf, open it up, scan it, close it, and put it back on the shelf. For a few items, that's not a big deal--but if you have to scan every single item? Well, let's assume
Scanning

scanning each item takes
Scanning

At 5 seconds per item,

around 5 seconds per item. At that rate,
Scanning

At 5 seconds per item,
One FTE:

one FTE could do around
Scanning

At 5 seconds per item,
One FTE:
720 per hour

720 per hour, or
Scanning

At 5 seconds per item,
One FTE:
28,800 per week

28,800 per week, or
Scanning

At 5 seconds per item,
One FTE:
1.4 million per year ...

around 1.4 million in a year. And, in our case,
Scanning

At 5 seconds per item, One FTE: 1.4 million per year ... for 2.7 million items

we have around 2.7 million physical volumes in our collection. That's a significant time and manpower investment. Plus, it doesn't take into account other drawbacks of this method like the fact that it requires you to
Scanning

Handling each item,

handle each item,
Handling each item, including hard-to-handle items,

including items that are hard to reach or otherwise difficult to handle,
Handling each item, including hard-to-handle items, increases risk of damage increasing the risk of dropping or damaging items. So, during the summer of 2013, the head of Collection Management
came to me with this issue. At the time, we had recently migrated from Millennium to Sierra, and she knew that Sierra came with the promise of comparatively open data,
Can Sierra Help?

so she wanted to know if we could come up with a good inventory method that would work well for us using whatever new capabilities Sierra had. Well, after discussing it,
we came up with an idea for a virtual shelflist application.
Essentially, it would be something you could load onto an iPad or other tablet (or a laptop) and take with you into the stacks. Just like a printed shelflist, you could use it to compare what Sierra said we had with what's actually there just by looking at the call numbers on the items' spines, and then
you could mark whether an item was on the shelf or not right there in the application.
You'd never actually need to remove an item from the shelf
unless the call number wasn’t on the spine or you needed to reshelve an item. In theory it seemed like a workable idea—we could just pull the data straight from the Sierra database using SQL, and maybe even keep the shelflist
## Shelflist, location w4m

<table>
<thead>
<tr>
<th>Call Number</th>
<th>Vol</th>
<th>Copy</th>
<th>Status</th>
<th>On Shelf?</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1.A13 H3</td>
<td>1</td>
<td></td>
<td>AVAILABLE</td>
<td></td>
</tr>
<tr>
<td>M1.A5122 1990</td>
<td>1</td>
<td></td>
<td>LIB USE ONLY</td>
<td>Yes</td>
</tr>
<tr>
<td>M1.A55 1999</td>
<td>1</td>
<td></td>
<td>LIB USE ONLY</td>
<td>Yes</td>
</tr>
<tr>
<td>M1.A6365</td>
<td>v. 1</td>
<td>1</td>
<td>AVAILABLE</td>
<td>No</td>
</tr>
</tbody>
</table>

up-to-date as things changed in Sierra.
<table>
<thead>
<tr>
<th>Call Number</th>
<th>Vol</th>
<th>Copy</th>
<th>Status</th>
<th>On Shelf?</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1.A13 H3</td>
<td>1</td>
<td>1</td>
<td>AVAILABLE</td>
<td></td>
</tr>
<tr>
<td>M1.A5122 1990</td>
<td>1</td>
<td>1</td>
<td>LIB USE ONLY</td>
<td>Yes</td>
</tr>
<tr>
<td>M1.A55 1999</td>
<td>1</td>
<td>1</td>
<td>LIB USE ONLY</td>
<td>Yes</td>
</tr>
<tr>
<td>M1.A6365</td>
<td>v. 1</td>
<td>1</td>
<td>MISSING</td>
<td>No</td>
</tr>
<tr>
<td>Call Number</td>
<td>Vol</td>
<td>Copy</td>
<td>Status</td>
<td>On Shelf?</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----</td>
<td>------</td>
<td>----------------</td>
<td>-----------</td>
</tr>
<tr>
<td>M1.A13 H3</td>
<td>1</td>
<td>1</td>
<td>AVAILABLE</td>
<td>Yes</td>
</tr>
<tr>
<td>M1.A5122 1990</td>
<td>1</td>
<td>1</td>
<td>LIB USE ONLY</td>
<td>Yes</td>
</tr>
<tr>
<td>M1.A55 1999</td>
<td>1</td>
<td>1</td>
<td>LIB USE ONLY</td>
<td>Yes</td>
</tr>
<tr>
<td>M1.A6365</td>
<td>v. 1</td>
<td>1</td>
<td>MISSING</td>
<td>No</td>
</tr>
</tbody>
</table>
So I built a working prototype in a few weeks.
The frontend was a JavaScript app, and
I had a backend component written in PHP that handled the communication
with the Sierra DB—just a super simple Web API that responded to Ajax requests sent by the frontend.
In the JS app, you’d enter
JavaScript

PHP

Sierra

w4m

a location code
and a call number range.
The app sent that to the server as an AJAX request.
On the backend, it would translate those parameters to
an SQL query,
send that to Sierra, and
pull back the call number, volume, copy, barcode, item status, and some other info for each item matching the query.
After retrieving results, it would
Normalize call numbers
and sort the list using those normalized call numbers. Then it
would build a JSON file that contained the entire shelflist and store that on disk on the server. The final step was
to return the first page of results
to the JavaScript client, which would format the results and display them to the user.
So--then, for the duration of that session, all further
requests involving that shelflist
would simply access that
JavaScript

PHP

Sierra

JSON file
instead of accessing the Sierra DB.
And when your session was done,
the JSON file would be deleted.
The prototype was a proof-of-concept showing that the idea could work,
Prototype

Serious drawbacks

but it did have a number of serious drawbacks.
First,

Drawback 1
the database query used to build the shelflist
could take a very long time to run.
Drawback 1

Slow shelflist query

For small shelflists it would complete in a minute or two, and for larger ones it could take 4 or 5 minutes. For very large shelflists, sometimes the query would time out. So if I actually put this into use, not only would people need to be aware that their shelflist could take several minutes to build, but they'd also need to know not to create a shelflist that was too large or it might not work at all. Not a very user-friendly approach.
Second,
shelflists were stored in JSON files that were tied to sessions and could essentially only be accessed by the browser that created them for the duration of that session.
So this particular prototype wouldn’t work very well in a multi-user environment or one where you’d need your "on shelf" or "not on shelf" information to have any sort of persistence. To get it to work at all in a real-world context, I’d need to add functionality for user management, permissions, and saving/editing/sharing shelflists. Which would be doable, but was a lot to add when I wasn't even really sure that the overall approach I was taking was best for this idea.
Drawback 3

Third,
there wasn’t really a good mechanism to update shelflists automatically with changes made in Sierra. The most straightforward way would require re-running the super long original query to the database and then updating the JSON file with changes—which would be simple enough to program, but then
Drawback 3

Syncing an existing JSON shelflist with Sierra

Dealing with the user interaction aspect gets tricky. Do you make people trigger an update and then make them wait? Or do you just do it automatically in the background every so often? Regularly having asynchronous requests open for minutes at a time seemed like a pretty bad idea, and the whole thing would fall apart anyway once you had five or more people using it at once, as they’d hit up against the concurrent connection limits in Sierra. It just really quickly turned into a big problematic mess when you started thinking about how it would actually work in a real-world setting.
So when I took a step back and analyzed things,
The Problem

I realized that the main problem here was
The Problem

Data access limitations → Application design

that limitations with the data and the limitations in the methods for accessing the data were dictating the application design. Right? I mean, the drawbacks I just listed all have to do with
the fact that I was querying Sierra to get the entire shelflist all at once and then
saving the whole shelflist to a file on the server. And the limitations that led to this particular architecture directly involve how I could access the underlying data.
The main limitation was that I couldn't access
appropriately normalized call numbers.
Sure, the Sierra database does make a normalized call number field available for us to use,
Normalized Call Numbers

`item_record_property`.

`call_number_norm` is useless for:

but that field doesn't work for us in certain important cases.
Normalized Call Numbers

`item_record_property.call_number_norm` is useless for:

Sudoc numbers,

It doesn't contain Sudoc numbers, which many of our Government Documents use for shelving—or, if it does (because we repeat the Sudoc number in a call number field), it doesn't normalize them as Sudoc numbers.
Normalized Call Numbers

*item_record_property.*
*call_number_norm*

is useless for:
Sudoc numbers,
Some local call numbers,

It doesn't deal well with certain quirks in some of our local call numbers.
And it doesn't contain a normalized version of the volume field, which has its own unique issues and has to be normalized independently.

Because of these limitations, I had to implement my own call number normalization. And because I used my normalized call numbers to sort the list in shelf order, I couldn't query the database just for a single page of results at a time— I had to pull back the entire shelflist and then normalize all the call numbers so that I had a complete set of normalized call numbers to sort on. In addition to that, my database query had to deal with the fact that it was querying non-normalized data, which led to a pretty monstrous query that I'm frankly a little surprised worked at all.

The other limitation that this approach also sort of worked around
was a general speed and responsiveness limitation. A Web app will feel sluggish if server responses take longer than a second to return, and the Sierra database just isn't reliably quick enough to make that happen. So by front-loading the database interaction and then just using the JSON file for normal end-user interaction, it made the Web app function more smoothly than it would have if I had been able to take an approach where I hit the database for each page of data.
Ultimately I didn’t want to spend extra time and effort continuing to try to work around these (and other) limitations, and so this version of our inventory app never made it out of the prototype stage. But, at the very least, the prototype clarified the underlying issue for me, which was
The Problem

Sierra Database is designed for Sierra!

that the Sierra database was designed and optimized for Sierra, not for my application. And design decisions that III's software engineers made about the database—how data is represented, how it's normalized or not normalized, and what indexes are built—these decisions all have real effects on how I can access the data and what my application can realistically do.
The Problem

Sierra API is designed around Sierra!

And there's a very similar situation with the Sierra API. It hadn't been released yet when I was working on this first prototype of the inventory app, but even once it was released I saw that there was no functionality there that would have helped for this use case--this just wasn't something it was designed to do. I mean, it all really comes back to the question of
What Can I Do With Your APIs?

what you can do with a given API. APIs didn't exist that did exactly what I needed. So maybe another option
was simply to invent my own. I had seen some messages on the Sierra listserv indicating that a few people were extracting portions of the Sierra database out to a separate PostgreSQL instance because of issues similar to the ones I'd run into, and that got me thinking along those lines. I started to consider what it would take to, number 1,
Invent a New API

Extract data from Sierra,

extract all of the needed data from Sierra and store it in some storage mechanism that I could control; number 2,
Invent a New API

Extract data from Sierra, sync data from Sierra,

ensure that I could keep the secondary database in sync with changes from Sierra; and, number 3,
Invent a New API

Extract data from Sierra, sync data from Sierra, and provide a usable interface.

provide some kind of usable interface on top of that data.
The more I thought about it, the more I thought having an infrastructure for creating and managing extraction processes as well as what data is exposed via the API could let us build out an ecosystem of unique projects in a controlled and consistent way. I also thought this kind of project could be useful to other III customers, potentially allowing others to build out their own API resources without having to invent their own infrastructure for that.
Now let's talk about how the catalog API project works.
Here's a big ugly diagram that shows all of the various pieces, how they fit together, and how data flows throughout the system. The project is mostly Python, and in this diagram the blue boxes are all Python components. The yellow pieces are non-Python processes or applications, and the red cylinders of course indicate data storage. The black arrows on the right show the flow of record data, and the green arrows on the left show the flow of logs, messages, and application data.
The Catalog API is a Python Django project.
There's an Export component, which contains code defining all of the various jobs that extract data from Sierra as well as how those export jobs are managed. To export data from Sierra, export jobs use
models, which are objects that model the Sierra database tables. An export job will use
models to retrieve data from the Sierra database.
Once an exporter gets data from Sierra via the model, it can essentially do whatever you program it to do with that data. For instance, we have exporters defined for Bib records.
that will use pymarc to translate bib record data to MARC records
and then call SolrMarc to index the data in Solr.
Other exporters will use Haystack or pysolr to index non-MARC data in Solr directly,
and there are a few exporters that have to store certain application data in Redis, because it's more efficient to retrieve it from Redis than it is from Solr.
When an export job runs,
a record of that export job gets stored in the Django database, and detailed log files get stored on the server. This lets you see what export jobs have run, whether they ran successfully, and what errors or warnings happened during export.
A task queue and scheduler called Celery keeps export jobs running smoothly. The Exporter exposes a Web interface that lets you manually trigger an export job or schedule an export job via Celery's scheduler.
Finally, the API component is where resources for the REST API are defined along with things like where the data for those resources comes from and how those resources can be queried. When you hit the API,
it uses Haystack or pysolr to query Solr and return results,
and for resources that need to pull information from Redis, that's retrieved as well. Then it serializes the relevant data in the right data format (in our case, JSON) and returns a response to the client.
Now let's take a closer look at how some of these components work, starting with Django. If you're not familiar with it,
Django is
Django

Web application framework:

a framework for building server-side, database-driven Web applications. It's basically a set of
Django

Web application framework: a set of Python modules;

libraries or modules that provide most of the functionality that you need to help you build a dynamic Web application so you don't have to invent boilerplate stuff yourself.
Django

Web application framework: a set of Python modules; fully featured

It is very fully featured and has built-in components for handling things like authentication, forms, unit testing, caching, and on and on. Because it's just a set of Python modules,
Django

Web application framework: a set of Python modules; fully featured and extensible

it's also very extensible. Django add-ons are just Python packages and can be installed and managed as such. Most Django sites, or
Django

Projects are composed of pluggable “apps,”

projects, are composed of pluggable apps,
Django

Projects are composed of pluggable “apps,” each of which serves a particular purpose within the larger project and is more or less self-contained.
So in our project,
these three components are separate Django apps. The
Base app
Base App

mainly contains
the models for the Sierra database. All 300-some-odd database views that III gives us access to are implemented in the Base app, even though right now we only actually make use of a handful of those.
What the models do is they give us a nice object-oriented interface for our Sierra data. So, essentially, rather than setting up a database connection, sending SQL queries, and then looping through results, we set up model objects that represent database tables and then send commands to those objects to create and filter sets of data. Django takes care of all the lower-level details for us, like connecting to the database, sending SQL queries, and caching results.
Models

Simpler Sierra data access,

This lets us access Sierra data in a way that's a bit more intuitive--
Models

Simpler Sierra data access, but you still need to know the database structure.

BUT, you do still need to know the underlying database structure, because the models directly reflect that structure.
Sierra Database

Quirks

And there are a few quirks with how the models are set up to work with the Sierra database.
Sierra Database

Some variation from institution to institution?

First of all, there’s probably going to be some degree of variation from institution to institution in whether certain tables and fields are used, depending on your setup and what III products you have installed on your system. Obviously, the models in this project were developed for our Sierra instance, and so you may run across some things you'll need to tweak for your situation. Second,
Ill customers actually only have access to database views, not the actual tables used by Sierra, and I've run across a few instances where data integrity isn't strictly enforced—like there are instances where you'd think there should be a relationship between tables, but there isn't a proper primary-foreign key relationship, which can cause problems if you try to force a relationship in the models.

In order to help with these two issues, I've written
extensive comments in the base app models.py code detailing oddities, and I've also written
Model Tests

base/tests.py

some unit tests on models that you can run that will check the models against your Sierra database tables. (There’s some more info about that in the README on Github.)
Let's talk about the the Export app. As you can see from the diagram,
It's pretty central to the overall system. The Export app
is where we have our exporter classes. And these are what define the jobs that we use to export different types of data out of Sierra.
Exporter Classes

Most of the exporter classes we have defined
Exporter Classes

Take some set of records,
Exporter Classes
Take some set of records, filter it,

filter that set based on a few specific pre-defined filters and criteria you define when you run the export,
Exporter Classes

Take some set of records, filter it, and output to Solr

and output the filtered record set out to Solr. For example, some of our exporter classes include ones that export
Exporter Classes

BibsToSolr

bib records,
Exporter Classes

BibsToSolr
ItemsToSolr

item records,
Exporter Classes

BibsToSolr
ItemsToSolr
LocationsToSolr

locations,
Exporter Classes

BibsToSolr
ItemsToSolr
LocationsToSolr
ItypesToSolr

item types,
Exporter Classes

BibsToSolr  
ItemsToSolr  
LocationsToSolr  
ItypesToSolr  
ItemStatusesToSolr

and item statuses to Solr. We also have some compound exporter classes,
Exporter Classes

BibsAndAttachedToSolr

like BibsAndAttachedToSolr,
Exporter Classes

BibsAndAttachedToSolr
ItemsBibsToSolr

and ItemsBibsToSolr. When you run these, they'll run multiple export processes at one time over a single record set. For example, if you run BibsAndAttachedToSolr, it will run all of the selected bib records through BibsToSolr. Then it will run all of the items attached to those bib records through ItemsToSolr. Then it will run any holdings records attached to the bib records through a HoldingsUpdate process. This can be useful when, for example, you want to have a Solr document that has data from multiple types of Sierra records, and you want to keep it in sync when any of those records changes.
Running Exports

Also included in the Export app are tools for helping run export jobs as regulated background processes. As you can imagine,
Running Exports

Large record loads:

you might want to run an export job that exports a huge number of records--like all the bib records in your database, for example. And we can't simply query the database and deal with all of those records at one time because it would return way too much data to handle at once. So what we do is we pass the export job off to a separate process that lets us
Running Exports

Large record loads: break into smaller chunks

break our filtered record set into chunks of set sizes, like 500 or 1000 records, and then run each chunk through the export job, as appropriate. In addition, doing this lets us
Running Exports

Large record loads: break into smaller chunks and run in parallel

process those chunks in parallel; so instead of one gigantic database query, we run several smaller queries and allow a certain number to run simultaneously, based on how many concurrent connections we're allowed to have to Sierra. This greatly speeds up our export processes. The thing that lets us work this magic is
Celery, which is an asynchronous, distributed task queue. Basically, you run one or more Celery workers as background processes. When you want to run a particular exporter, you pass the name of that exporter class and the filter options you want to use to a function that breaks the export job into tasks and passes them off to the Celery workers. Celery then churns through the tasks in the background until they're done--whether that takes seconds, hours, or days.
Scheduled Tasks

Celery also provides the ability to schedule your tasks. This is tremendously useful in a production environment,
Scheduled Tasks

Keep things in sync with Sierra

where we want to have certain exports run periodically to ensure that our API data is kept in sync with Sierra.
The last thing that the Export app does is it hooks into the built-in Django admin Web interface, which lets you manage export jobs from a Web browser.
Here's what that looks like. From here, looking at the top portion, you can see that we have links that let you manage certain metadata associated with exports. But the links at the bottom, where it says Manage Export Jobs, are what I really want to show you. From here, if we click Trigger New Export, at the bottom,
we can manually trigger a new export job to run, using this form to select what job we want to run and what filter parameters to use. If we click Go, it runs the export process. Whenever a new export process runs, a new "export" record in the Django database is created. This stores
the current status of the job, the export type that was used, the export filter that was used along with any filter parameters, the user that ran the export, the time that it was started, and the number of errors and warnings that were generated during the export. (And there are logs we can check on the server that will give us a detailed list of errors and stack traces as well as an exact count of how many records were processed, if we want that much detail.)
Then, back on the export admin screen, if we select View in-progress or completed exports,
we get a list of all the export jobs that have run, and we can filter by various facets, like by status. This is useful for tracking down problems and troubleshooting issues where, for example, one or more records weren't properly synced.
And, last, if we click View Scheduled Exports,
we go to a screen that lets us view and edit all of the exports we've scheduled. Since this screenshot was taken on my dev server, there's only one test task there. But, if we click on that,
then we see how that's set up. We specify the Celery task that we want triggered along with the arguments that are sent to that task, and we specify how often it runs. So here, we've scheduled a task that will run a BibsToSolr job using a filter that grabs all records that have been updated since the last export, and it will run that every 2 minutes. You have the choice between setting up specific time intervals, like this, or using a crontab-like schedule that lets you run a task at specific times on specific days.
So using the Export app and having various export jobs run at certain times,
we have data flowing from Sierra into Solr on a regular, ongoing basis. Before I talk about how the API app works, I want to take a really quick detour just for a minute and talk about
Solr. If you aren't familiar with it, Solr
Solr
Open source search index

is an open source search index, written in Java. It's been around for a while--since 2004, and it's very widely used in the library world.
For this project, we're using it kind of as a NoSQL database,
Solr

Not really NoSQL

although--again, Solr is a search index, not a NoSQL database.
On a painfully broad and oversimplified level, this is how Solr works.
Along with some other configuration, you provide a schema, which is an XML file that defines what fields you want in your index, what data types those fields use, and how you want to parse those fields for indexing.
Then you provide a set of "documents"--like XML documents or JSON files--that contain data in fields you've specified in your schema. Solr builds an index of the provided documents based on your schema.
Then you can query Solr and retrieve search results in JSON or XML via Web applications that you built on top of it.
For the catalog API, we have exporters defined that use a few different methods to generate the documents that go into Solr.
Catalog API

Haystack,

Haystack,
pysolr,
and SolrMarc are all different tools we use for generating those documents from our Sierra data and getting them indexed.
And whenever we want to change what's indexed, like if we want to add new fields or new API resources, we just need to
update the Schema file,
update code in the Catalog API as needed to export the new data and implement the fields or resources in the API,
and then reindex the affected documents.
And you might be wondering why we chose Solr over other options, like ElasticSearch. This could trigger a whole separate tangent, but—number 1, Solr matched our needs, and number 2, we have a lot of in-house support for it, since UNT's digital library is built on Solr. It was kind of a natural choice. And, in retrospect—it works well for this. It's easy to work with, powerful, and performs very well.
Finally, let's talk about the API app. As you'd expect, this is what actually implements the REST API.
It uses a Django add-on module called Django REST Framework, which provides a lot of helpful functionality and a nice simple framework for building out the API. And I think how this works is actually best demonstrated by looking at our API and talking through its features. First I want to describe the
API

Hypermedia

hypermedia model that we're using.
At the root of the API is a catalogAPI resource. The main purpose of that resource is to help clients discover what resources are part of the API. So it links out
to all of the different "list" or collection resources that are available. Each of these collection resources provides a paginated list for a particular group of resources, like, for example,
the "bibs" collection, which links to every bib contained in that collection. At the collection level,
you can use query parameters to filter and search the list as well as page through it.
Each individual bib then contains links to resources that are related to it, like a marc resource that contains MARC-in-JSON for that bib record,
and whatever items are associated with it.
Besides the root resource, we currently have 11 different resources defined, and all of them
but these two follow this basic pattern.
So let's look at what this looks like in a Web browser. If we visit the API root,
here’s what we see. The first thing you might notice is that what it's giving me isn't actually JSON--it's a handy HTML representation of the resource at that URL, with the JSON data displayed in a nice readable format.
So, right off the bat, that's one very valuable thing that we get with Django REST Framework--with absolutely no extra work on our part, visiting our REST Framework resources in a Web browser gives us that browseable HTML representation.
But of course you get plain JSON if that's what you request from a Web application that's using the API.
So another thing you can see here is that the content type we're serving is
HAL+JSON.
HAL stands for Hypertext Application Language, and it's a content format designed to provide linkable resources within a REST API in perhaps the simplest way possible. And that's mainly what we're using HAL to do--
to provide links between resources in a straightforward and standardized way. This _links object here is a HAL construct, and that gives us our links to the various collection resources that hang off the API root resource.
So if we follow the link to the "bibs" resource,
here's what we get. This is a paginated list of every bib available in the API. And just about every list or collection resource follows this pattern. We get a
count of all of the resources in that list—a little over 2.9 million,
and the starting and ending row for the current page.
We have links here that represent the current page, the next page, and—if we weren’t already on the first page, the previous page.
And then the data for all 20 bib resources appears on the page embedded here within this _embedded object.
If we scroll down, then we can see more of the data for the first bib resource.
Each bib has its own links you can follow. One points to itself (which gives you the URI just to get that one record), and others point to the resources I mentioned earlier--marc and items.
And then you have the other data fields associated with that bib resource, as we've defined them.
Collection resources let you search them, and I'm actually kind of proud of the querying capabilities that I've built into our API, because this is one of the things that sets this apart from the Sierra API.
First of all, the majority of the fields on any given resource can be used for querying. So you can actually search and filter on record content and not just a small set of fields. And because it's using Solr on the backend, which was built for search--searches are nice and fast.
Second, you can query using several different operators.
Query Operators

exact

Exact match,
**Query Operators**

- **exact**
- **startswith**

String starts with some other string.
Query Operators

effective
startswith
endswith

string endswith some other string,
Query Operators

exact
startswith
endswith
contains

string contains some other string,
Query Operators

exact
startswith
endswith
contains
gt/gte

greater than and greater than or equal to,
Query Operators

exact
startswith
endswith
contains
gt/gte
lt/lte

less than and less than or equal to,
Query Operators

- exact
- startswith
- endswith
- contains
- gt/gte
- lt/lte

range--which can apply to strings, numbers, or dates--
Query Operators

exact
startswith
endswith
contains
gt/gte
lt/lte
range
in

in, which lets you match based on a list of possible values,
<table>
<thead>
<tr>
<th>Query Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>exact</td>
</tr>
<tr>
<td>startswith</td>
</tr>
<tr>
<td>endswith</td>
</tr>
<tr>
<td>contains</td>
</tr>
<tr>
<td>gt/gte</td>
</tr>
<tr>
<td>lt/lte</td>
</tr>
<tr>
<td>range</td>
</tr>
<tr>
<td>in</td>
</tr>
<tr>
<td>isnull</td>
</tr>
<tr>
<td>isnull,</td>
</tr>
</tbody>
</table>
### Query Operators

<table>
<thead>
<tr>
<th>exact</th>
<th>range</th>
</tr>
</thead>
<tbody>
<tr>
<td>startswith</td>
<td>in</td>
</tr>
<tr>
<td>endswith</td>
<td>isnull</td>
</tr>
<tr>
<td>contains</td>
<td>keywords</td>
</tr>
<tr>
<td>gt/gte</td>
<td></td>
</tr>
<tr>
<td>lt/lte</td>
<td></td>
</tr>
</tbody>
</table>

Keywords, which lets you perform a boolean keyword search on certain fields,
<table>
<thead>
<tr>
<th>Query Operators</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>exact</td>
<td>range</td>
</tr>
<tr>
<td>startswith</td>
<td>in</td>
</tr>
<tr>
<td>endswith</td>
<td>isnull</td>
</tr>
<tr>
<td>contains</td>
<td>keywords</td>
</tr>
<tr>
<td>gt/gte</td>
<td>matches</td>
</tr>
<tr>
<td>lt/lte</td>
<td></td>
</tr>
</tbody>
</table>

and matches, which lets you match a string on a regular expression. (And is still a little experimental.)
Query Operators

- exact
- startswith
- endswith
- contains
- gt/-gte
- lt/-lte
- range
- in
- isnull
- keywords
- matches

You can also use a hyphen to indicate "not," so we have doesn't exactly match, doesn't start with, doesn't end with, etc.
Now let's look at a few examples of what queries look like.
This queries on main call number starts with MT. You can see here that you just include the field you want to query, with your operator in square brackets, and then you query value.
If we run this query, we filter that list of 2.9 million bib records down to just 13,874, and
if we scroll to some of the results, you can see that the filtered records do match.
We can add more criteria to further filter, if we want. In this case, we're adding a suppressed=true filter. Since there's no operator specified, it defaults to "exact."
We can also use an order_by parameter, which sorts the list of resources.
Here's an example that shows a keyword search--
looking for hamlet or shakespeare
in the fullTitle field,
bibs/
?fullTitle[keywords]=hamlet%20OR%20shakespeare &fullSubject[keywords]=english

and english
bibs/
?fullTitle[keywords]=hamlet%20OR%20shakespeare
&fullSubject[keywords]=english

in the fullSubject field.
Here's another example, this time showing that we can also query

marc/?050a[startswith]=MT
marc/?050a[startswith]=MT

the marc resource by MARC field and/or subfield. This looks for a MARC 050 subfield 'a' starting with MT.
marc/?O50[matches]=^..\$aMT

And here,
marc/?050[matches]=^..\$aMT

it's querying on the full 050 field, matching using a regular expression that will find
mArc/?050[matches]=^..\$aMT

a subfield "a" starting with MT
marc/?O50[matches]=^..\$aMT

at the beginning of the field, after the two indicators, which can be any value.
Up to this point, I've shown you how to work with the API in a read-only capacity. Currently the API doesn't let you write data back to Sierra, because the records we have modeled can't be written back to Sierra.
But there are instances where it would be helpful to be able to write data to the API, even if that doesn't go back into Sierra. In fact, for our production implementation of our inventory app, we do have one resource--shelflistitem--that we can write to. So
HTTP PUT and PATCH methods are currently supported in our implementation for that one resource. If we had whole resources that could be written and deleted via the API, then we'd support POST and DELETE as well, but we haven't had the need to do that yet.
Because most of the API is read-only, authentication is not needed to access the majority of the API. However,
to issue PUT and PATCH requests to shelflistitems, you do have to authenticate using a simple custom authentication scheme that we've implemented. There is also an
apiusers resource that requires authentication to view.
That concludes our detailed look at how the system works. And now that you've seen how the various components fit together, I wanted to point out that one of the nice things about the way this is built is that using the API doesn't hit the Sierra database at all. This completely mitigates the issues on the client side that we saw when we were trying to access live data in the Sierra database. We don't have to worry about data being formatted or stored in ways we can't use. We don't have to worry about clients running up against the concurrent connection limit. And we don't have to worry about database queries slowing down API response times. Those issues are managed at the Export level, where data gets normalized and transformed as needed and speed and concurrency can be tightly regulated. The one downside is that our API isn't serving live Sierra data. So, depending on how often we run our export jobs, there will be a lag between the time that data in Sierra is changed and the next time data is synced.
And now it's time to put my money where my mouth is and show you some things we've built with this so far. I considered doing a live demo here but decided that screenshots would be a little safer and would help me stick to my allotted time better. Note that the screenshots I've included are zoomed in so you can see what's going on, but that means they look a little different than what you'd normally see.
I want to show you three end-user facing apps, and they are:
Three End-User Facing Apps

LibGuides widget-builder

first, a LibGuides widget-builder that allows subject librarians to embed lists of bib records from the Catalog API in their subject pages,
Second, our "bento box" discovery interface that we use as the primary discovery interface on our library website,
Three End-User Facing Apps

LibGuides widget-builder
“Bento Box” discovery
Inventory app

and third, the production version of our inventory app.
Let's start with the LibGuides widget-builder. A few months ago subject librarians asked me if there were a way to embed a feed of new catalog items in their LibGuides subject pages, which is something that comes up from time to time. There **is** an RSS feed builder product from III that would do it, but we don't have that product. So this seemed like a perfect application for the catalog API.
The way it works is pretty simple. I wrote a JavaScript component that I embedded into a private LibGuide. Librarians can visit this guide to construct their custom feed.
Using this form,
they can set how many results display,
and they can query on any of the fields that are searchable via the API on the bib resource. So, for example, Library of Congress call numbers
starts with
M, should give us a list of music items.
You can add as many fields as you want. Since querying multiple fields will AND them together, this narrows your search—in this case to things with a material type of Music Scores.
Then you can choose how to sort your list. Since this is meant to be a feed for new items, choices here are limited. If you sort descending by recordNumber, then you'll get the newest bib records added to the catalog; if you sort descending by timestamp, then you'll get the most recently updated bib records.
At any point, you can click the Update Preview button. This sends an AJAX request containing your query to the catalog API and gets results back as JSON. Two things on the page will update:
the code pane, and (if you scroll down) the preview pane. The code pane contains the code that you need to embed in your LibGuide to get your feed to display, and the preview pane contains a preview of what those results will look like. By default we've got the full title, which is linked to the record in the online catalog, the material type in brackets, the statement of responsibility (if present), and the imprint--but this display can be modified, which we'll see in a second.

At this point, if you want, you can go back up and edit your query and then update it again, and the code and preview panes will update accordingly each time you click the Update Preview button. Once you're satisfied, just
click the Select Code button, and copy and paste that
into a widget on your own LibGuide. If you save and refresh,
you should see the same results that you saw in the preview in your guide.
One thing to note is that the embed code contains the underscore-style template that's used for the results display. So if you wanted, you could easily
edit that template for your guide to change what fields get displayed, where, and how. Data from the full bib resource is contained in the "data" variable, so you can access and display any of those fields that you want. In this case, I've just added the bib record number.
to the top of each result. You could also add custom CSS to style things to your liking, as well.
So for the LibGuides widget builder, here’s a diagram summarizing what’s going on. You just have JavaScript embedded in LibGuides pages that sends AJAX queries to the catalog API, pulls back JSON data, and updates the display.
Now let's look at the bento box app. What this is is a discovery interface that I built that basically does a federated search of several different discovery systems and pulls results into one interface, subdivided by the result type.
The bento box app serves as the default search if you're searching from the library homepage. When you search,
you get results that look like this. You have results for Articles that get pulled in from our Summon instance and results for Books & More that come from our catalog.
You get a result for the most applicable subject librarian based on a search of LibGuide and staff directory content. You also get results for journals, databases, subject guides,
digital collections (from our digital library), reference materials from Summon,
and results from the library website. And if you click any of these results, it takes you to that result in whatever system it lives in.
The bento box app makes use of several different APIs in order to pull in all of these different results: the Catalog API, the Summon API, our Google Custom Search API for the website, our Digital Library API, and a couple of additional Solr indexes that we built for librarians and libguides data. What's going on is, we have the bento box frontend implemented as a JavaScript app, and that communicates with a relatively simple backend API that is responsible for searching each target and providing consistently-formatted results to the front-end. So, when you trigger a search, it fires off.
several AJAX requests at once--one for each different set of results. Each request hits a different API resource: one for articles, one for books-and-more, one for journals, one for databases, and so on. The backend then contains code to translate the request on each resource into the appropriate query for the API that serves that content.
It sends requests out to each API as needed, and then
gets results back. Each handler for each resource formats the results into a JSON object.
that's sent back to the frontend. And each separate request that's made is handled independently, so results come back and are updated in the frontend as they're received and returned from the backend. So, for instance, if Summon is down or responding slowly to requests, it's not going to make the rest of the results load slowly.
The catalog API is what serves results for the journals and databases boxes. I have special search filters in the catalog API I built specifically for the bento box that send searches to Solr with detailed weighting for relevance ranking. This lets us provide better results for journals and databases than you typically see, for databases, especially. We index all of the journal titles associated with a database in Solr on that eresource document, so you get a relevant database hit when there's a relevant journal title associated with it.
Results for Books & More, on the other hand, do not actually come from the Catalog API. Rather, they're scraped from our online catalog. This is because we want the results that display in the bento box to match
the results in the catalog, and we have no desire to try to reproduce III's relevance ranking in the Catalog API, which I doubt is even possible.
That brings us to the final thing I want to show you, which is our finished inventory app. Everything I talked about earlier regarding doing inventory still applies; in fact, many of the API resources that I've built, I've built to support a working inventory app that doesn't suffer from the various drawbacks I discussed.
Again, the idea is that you load this up on an iPad or other tablet and take to the stacks with you. It's just a Web App, so it works on a variety of devices. If we're not logged in, we can browse shelflists in read-only mode but can't actually record information, so I've already logged in. (Since we're zoomed in here, the log-in and log-out is hidden under the hamburger menu.)
The first thing we'd want to do is use the "Find Item" button to try to locate whatever item is in front of us on the shelf that we're trying to inventory.
We can try to enter the location code, if we know it,
and we get a handy autocomplete. If we don't know the location,
that's fine.
We can search for an item by barcode or by several different types of call numbers. Call Number (any) will search all call number types.
So let's say that we want to inventory the books on this shelf. The first one at the very edge on the left is PR2807 .A2 E4 1985.
So, we type the call number here. We get an autocomplete that matches our search and updates as we type, based on the options we've already selected--location and search type.
If we search on a partial call number, then it will try to take us to the first item that matches. But if we don't specify a location, then there's a chance that it might find matches on multiple shelflists. If it does, then we get
a screen showing the first match at each location with a match. From here, hopefully we can narrow it down. When we do, we touch the appropriate Go button to take us to that shelflist.
So here's our shelflist. We get taken straight to the item that we were searching for, which is the one that's first in this view. At the top of the screen we can see where we are in the overall shelflist for this location. In this case, we're in the shelflist for location w3, starting with row 231,024 out of 326,648.

So now we just go down the list and check the call numbers here against the ones on the books. If a book is on the shelf,
we touch the green thumbs up button, and that marks it as on the shelf. If it isn't on the shelf,
the red thumbs down marks it as not on the shelf. If you need to change something's status, you can just
<table>
<thead>
<tr>
<th>Call Number</th>
<th>CN Type</th>
<th>Vol</th>
<th>Copy</th>
<th>Barcode</th>
<th>Item Status</th>
<th>On Shelf?</th>
<th>Flags &amp; Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR2807.A2 E4 1985</td>
<td>lc</td>
<td>1</td>
<td></td>
<td>1001298248</td>
<td>AVAILABLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PR2807.A2 H28 1999</td>
<td>lc</td>
<td>1</td>
<td></td>
<td>1001551744</td>
<td>AVAILABLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PR2807.A2 H5 1987</td>
<td>lc</td>
<td>1</td>
<td></td>
<td>1000492424</td>
<td>AVAILABLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PR2807.A2 J4 1982</td>
<td>lc</td>
<td>1</td>
<td></td>
<td>1000212011</td>
<td>AVAILABLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PR2807.A2 J6 1987</td>
<td>lc</td>
<td>1</td>
<td></td>
<td>1000245507</td>
<td>AVAILABLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PR2807.A2 K45 1987</td>
<td>lc</td>
<td>1</td>
<td></td>
<td>1002495317</td>
<td>AVAILABLE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

touch it again to clear it.
As we're going, we might run across an item where it isn't entirely clear that the item matches the one on the shelf. If we touch the big "plus" button over on the left side of the screen, then
that row opens up and gives us some more information. If we're still not sure, we can view the record in the online catalog, which will open up on a new tab. We close this again by clicking the "minus" button.
As we're doing this, it isn't always simply a matter of checking to see whether something is or isn't on the shelf. There are a number of issues we might run across where we'd want to add a note or a flag to an item so we (or someone else) can come back to it later. So if we touch the pencil button on the right beside a row,
we get a window for editing flags and notes. There are a number of things going on here.
First, let's talk about notes. There are system notes and user-entered notes. I want to point out that
a system note gets added here each time you change the on-shelf status or change a flag. Normally this gives you the username for who set the status or flag, but I've cropped usernames out of all of my screenshots.
If you want, you can enter your own note down here, where it says enter new note. Again, normally the username for the logged-in user would appear in the box that's blank.
When we enter the note, the timestamp on the left updates to reflect the current time, and when we click save changes, it saves whatever changes we've made here. We can add additional notes if we want to, and we can click the red X button to delete a note, if we have permissions. Note that I'm logged in with an administrative user that has permission to delete system notes--regular users do not, and so they would not see the red button beside system notes.
The other thing you can do on this screen is set or clear flags. I've defined flags in two categories--ones indicating problems, and ones indicating various workflow statuses. For this item, I've said in my note that the call number needs to be fixed in Sierra, so I might
flag it as having a data problem and I might also pull the item for review.
I need to save changes again. And we also see here that system notes have been added showing that I set those two flags. Since notes sort by date and time, notes you've added that relate to flags you set will group together.
Now if I close that window, we see that the icons on the right for the row I was editing have updated. The blue push-pin icon indicates that the item has user-entered notes. The red barcode icon indicates I've flagged it as having a data problem, and the blue icon on the right indicates I've flagged it as pulled for review. The icons let you see at a glance what information has been recorded about an item without having to click into the flags and notes screen.

So, once we're done with an inventory session, we might want to go back later and try to fix issues that we ran across. In that case, we could, for instance, access this shelflist on a desktop computer, and then use
filters to find things on the shelflist we might want to fix.
We can filter by several fields--inventory status, item status, suppression, flags, notes, and call number.
For instance, I might want to filter by items that are marked as not on shelf
and have an AVAILABLE (and not checked out) status in Sierra.
Applying the filter limits the rows to the things that match.
We can also filter by flags,
and we can filter by notes. We can find items that have any kind of user-entered note, and we can also search by text anywhere in the note, including usernames.
The last thing I want to mention is that there's also an Export button. This will export the shelflist to a CSV file. If we have active filters set, then the exported list will be filtered accordingly.
The inventory app works just like the LibGuides widget builder. The front-end app is written entirely in JavaScript, and it communicates with the Catalog API via AJAX requests. Saving information in the inventory app saves that information back to Solr immediately, so—if you have two people looking at the same items on the same shelflist, then things that one person updates will be visible to the other person next time their screen updates. Unlike in the original inventory app prototype, shelflists are both universally shared and completely persistent.
Thank you!

That's the end of my presentation! Thank you for listening.
Let me reiterate that the code for the catalog API is up on GitHub, and please feel free to try to get it up and running in your own environment.
Image Credits

Slide 1: “20120112: Clock” by Shaun RootRueger, licensed under CC BY-NC-SA 2.0

Slide 2-5: “hand modeling” by Nate Steiner, licensed under CC 0

Slide 6: “Calm” by Gregory Rinaldi, licensed under CC BY-NC 2.0

Slide 7-8, 10: “Software developer beard” by quimby, licensed under CC BY-NC-SA 2.0
Image Credits

Slide 8: “Be Kind to Your Librarian--It's National Library Week--Daily Image 2011--April 13” by rochelle hartman, licensed under CC BY 2.0

Slide 9: “blunt instruments” by Crispin Semmens, licensed under CC BY-SA 2.0

Slide 20: “Walnut” by Harry, licensed under CC BY-NC 2.0
Image Credits

Slide 111: “Bad Ragaz - Reinvented the wheel” by Kecko, licensed under CC BY 2.0

Slide 115: “Open Book” by Dave Dugdale, licensed under CC BY-SA 2.0

Slide 180: “Hmm” by Erich Ferdinand, licensed under CC BY 2.0