Newt, assemble web applications with Pandoc, Postgres and PostgREST

R. S. Doiel, rsdoiel@caltech.edu

Caltech Library, Digital Library Development

July 14, 2023

Today, LAMP and its legacy

Four example systems found in Caltech Library

- EPrints
- Archivesspace
- Islandora
- Invenio RDM

Required Knowledge

App	Languages	Supporting services
ArchivesSpace	Java, Ruby, SQL	MySQL, Solr, Apache or NginX, Solr
EPrints	Perl, SQL, XML, EPrint templates	MySQL, Apache2 (tight integration), Sphynx
Invenio RDM	Python,SQL JavaScript/React NodeJS/NPM	Postgres, Redis, Elasticsearch, Docker, Invenio Framework, Python packaging system
Islandora	PHP/SQL	MySQL, Fedora, Apache 2

These are all really complicated pieces of software.

The problem

Each listed application is built on a stack. The stacks are complex. Because of the complexity it's hard to sustain them. Some we've outsourced to SAAS providers (e.g. ArchivesSpace). Some we treat as a back boxes (e.g. EPrints). It's just not fun supporting applications at this level of complexity. It takes too much time and energy. It detracts from delivering useful things to our Library, Archives and Caltech Community.

Why are these things so complex?

WARNING: gross generatilizations ahead

- We want more from our application so more code gets written, complexity acrues over time
- We must build systems to scale!
  - a not so subtile influence on developer “best practices” from Silicon Valley

Let's talk about scale

- Best practice often translates to building for scale, specificly scaling up
- Scaling up => programmable infrastructure, the siren song of Google, AWS and Azure
  - Scaling big is hard
  - Scaling big makes things really complex
  - Do we ever really need to build at Google/Amazon/Azure scale?

The alternative

- Scaling down $<=$ pack only what you need
- Simplify!

Scaling down

- Limit the moving parts
- Limit the cognitive shifts
- Minimize the toolbox while maximizing how you use it
- Write less code! But remain readable!

How minimal can we go?

- Off the self microservices
- Build with SQL and Pandoc

Can you make web applications using only SQL and Pandoc?

Just about. Here's the off the shelf microservices I am experimenting with

- Postgres
- PostgREST
- Pandoc
- Newt

A clear division of labor

Simplify through a clear division of labor.

- Postgres + PostgREST => JSON DATA API
- Pandoc => Powerful template engine
- Newt => data router

How would this work in practice?

1. Model our data using SQL (Postgres)
2. Define our JSON API usng SQL (Postgres+PostgREST)
3. Transform our structured data using Pandoc
4. Use Newt to orchestrate

A game of telephone web browser => Newt => PostgREST => Pandoc => Newt => web browser

Required Toolbox

- Text editor
- Spreadsheet (optional)
- Web browser
- Pandoc
- Postgres + PostgREST
- Newt

Server side knowledge requirements

- SQL
- Pandoc templates
- CSV file describing data flowing through the microservices

Web browser knowledge requirements

- HTML
- CSS (optional)
- JavaScript (optional)

What does this enable?

We can create interactive applications with some SQL, Pandoc templates and a little routing.

Why SQL?

SQL is really good at describing structured data. It also is good at expressing querys. With a little deeper knowledge of SQL you can also define data views, functions and your own procedures. With Postgres + PostgREST these provide everything you need in a JSON data API short of file upload. SQL does allot of lifting with a little code and usually remains readable.

Minimize the source Luke!

You don't need to design classes in your favorite programming languages and Schema in SQL. You don't need to learn an ORM. You don't duplicate the code in the data base, again in the middleware and inceasingly often in the browser. Data models are defined in one place, Postgres. PostgREST takes care of turning them into a JSON data API. Data transformation is hanlde by Pandoc. A program really good at translating things. Newt provides just enough orchestration based on defining some routes in a CSV file.

Fewer cognative shifts

- SQL (Postgres dialect)
- JSON
- Pandoc templates
- Describing data flow in a CSV file using a simple notation

data flow
: web browser => Newt => PostgREST => Pandoc => Newt => web browser

helpful to know

- HTML 5 related W3C technologies
  - HTML 5 markup
  - CSS
  - JavaScript
- Understand how HTTP works, including HTTP methods and Headers
- How to handle static file assets, e.g. image and video files

But is this really simpler?

What was needed for each version of birds?

Let's take a look at three versions of a bird sighting website.

- birds 1, a static site implementation
- birds 2, a dynamic site implementation, content viewing requires browser JavaScript
- birds 3, a dynamic site implemention, does not require browser JavaScript

Different birds

birds 1 static site (read only)

- Built with Pandoc from Markdown and CSV file
- Adds bird sightings via updating a CSV file and rebuilding site with Pandoc

Different birds

birds 2, dynamic site (read/write)

- Built with SQL using Postgres + PostgREST
- Add birds you using a web form
- Requires the web browser to assemble pages via API calls
- JavaScript becomes complex between fetching data and inserting it into the page
- Doesn't work in text only web browsers like Lynx

Different birds

birds 3, dynamic site (read/write)

- Build from SQL (Postgres + PostgREST) and Pandoc
- Add birds you using a web form
- Rendered on server and no longer reqires JavaScript
- Works even in text web browers like Lynx

Different birds

Pros and cons

version	site type	pros/cons
birds 1	static	easy to conceptualize / read only
birds 2	dynamic	read write site / requires browser JavaScript, JavaScript is complex
birds 3	dynamic	read write site, easy to conceptualize / requires SQL and knowledge of Pandoc

Birds 3 => Postgres+PostgREST, Pandoc and Newt

The complicated activities are handled by the off the self microservices. The remaining complexity is limited to SQL to model data and our Pandoc templates.

- Avoids browser side page assembly
- Leverages our Pandoc knowledge
- Data is modeled using SQL

Newt manages data flow

- request => data API => Pandoc => response
- Newt's routes can be managed in spreadsheet!

Developer workflow

1. Model data in Postgres
2. Create/edit Pandoc templates
3. Create/edit routes CSV file in a spreadsheet
4. (Re)start Newt and PostgREST to (re)load models and routes

Repeat as needed

Minimizing newness

- If you've attended a data science workshop you are likely know enough SQL
- If you've built a static website with Pandoc you know how Pandoc works
- I think there is community that knows some SQL, CSV files and knows Pandoc

=> Is this useful for Libraries, Archives and Museums?

Evaluating Postgres+PostgREST, Pandoc and Newt

- Weeknesses
  - Newt is limited (doesn't support file uploads)
  - PostgREST is new to me (and probably others)
  - SQL and HTML have a learning curve
- Strengths
  - SQL is proven and likely to be around a very long time
  - HTML is proven and likely to be around a very long time
  - Postgres and Pandoc are very mature software
  - PostgREST and Newt are useful today and worth exploring

Next steps for Newt?

- Newt is an experiment, I am building some staff applications Summer/Fall 2023
- Solr/Opensearch should work with Pandoc and Newt, seems promising

somday, maybe

- It'd be nice if Newt could send file uploads to a service like S3 (via Minio?)
- It might be nice if Newt could function as a static file server

Thank you!

- Presentation https://caltechlibrary.github.io/newt/presentation/
- Project: https://github.com/caltechlibrary/newt
- Email: rsdoiel@caltech.edu