
ODTUG
Kscope16
CHICAGO, ILLINOIS · JUNE 26-30
PLEASE FILL OUT YOUR EVALUATIONS

Stateful to Stateless

Modernizing with JSON and PL/SQL

???

Did you just use “modern” and “PL/SQL” in the same sentence?

Yes

And we'll talk about how old, backwards and uncool it is to use a big ORM or embed SQL in your application's middle tier.

Speaker: Bill Coulam

Contacts

- Website: dbsherpa.com (blog) and dbartisans.com
- Twitter: @billcoulam
- Email: bill.coulam@dbsherpa.com
- LinkedIn: billcoulam

Experience and Focus

- C, C++, Java, JavaScript and **PL/SQL since 1995**
- **Oracle data/database design and tuning since 1997**
- **Passionate about best programming/design practices**
- Andersen Consulting - San Francisco, Denver, Herndon VA
- New Global Telecom - Denver, CO
- The Structure Group - Houston, TX
- Church of Jesus Christ of Latter Day Saints - SLC, UT
- Speaker at RMOUG, IOUG, ODTUG and UTOUG since 2001
- 2015 Oracle Developer of the Year nominee

Learning Goals

Through a case study and lessons learned, we will cover:

1. ORM features and frights. When an ORM is replaced or removed, what critical functionality must be replaced.
2. Why SQL embedded in the middle tier application server is generally a bad idea and how to champion its relocation to the database where it belongs.
3. How to politically and technically leverage existing PL/SQL and Oracle assets to save the company time, pain and money, and provide a simple, robust RESTful data services layer as a bonus.

Learning Goals

- Despite the title, we will NOT cover JSON or Oracle's new JSON support. JSON was a key part of our project, but I won't be talking about how to generate it or parse it or make Angular use it.

The Tour

- **Background**
- Old System
- The Challenge
- Solution
  - Angular Angles
  - ~~ORMmmmm~~ Lightweight Services
  - Data Done Delightfully
- Lessons Learned

Case Study Background

Internal web application used to review, prepare and assign senior women and senior couples to missions and service assignments across the world.

Old system was created in a hurry in 2007 with little thought for the future, performance, reliability, etc.

Application had rough edges and slow performance. HQ users learned to live with it, but not love it.

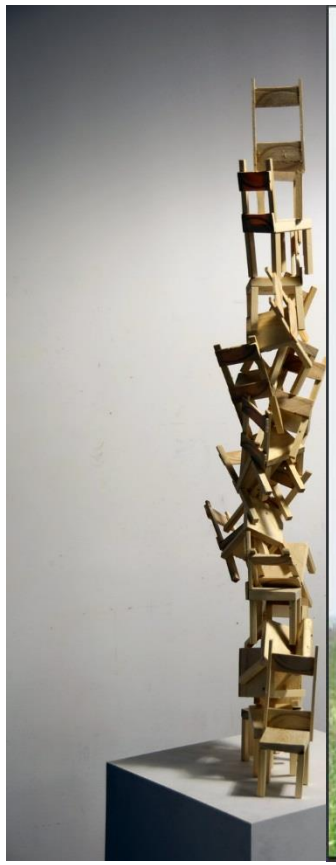
Some requirements were missed. HQ created its own complex spreadsheet to fill the gap, and much of the assignment process was dependent on one guy and that spreadsheet.

Case Study Background

- To incorporate “slots” from the spreadsheet, redesign the whole process, streamline, and add a few features, we were given a project to rework most pages of the application.
- But as we started into it…
  - UI developer found that enhancing in its current state took far longer than it would to rewrite from scratch.
  - Just the first couple pages took a week or so apiece.
  - He complained that the code was so huge, convoluted, many-layered and poorly-done, it actually scared him to open the hood and work on it.

Case Study Background

- Working on the system left developers with feelings/memories like:



Case Study Background

- Developers gathered facts, presented to management and asked for permission to explore a rewrite.
- After a 2 week proof of concept (POC) exploration, developers claimed re-writing a screen using Angular was far faster and resulted in a smaller, cleaner, more maintainable system.
- UI developer promised management we would get done in the same amount of time, or less, as original budget.
- Management gave us permission to rewrite it as a modern, stateless, single-page application (SPA).
- What follows are the valuable lessons learned along the way.

The Tour

- Background
- **Old System**
- The Challenge
- Solution
  - Angular Angles
  - ~~ORMmmmm~~ Lightweight Services
  - Data Done Delightfully
- Lessons Learned

Old System: UI

- Web pages generated by JSF and Struts.
  - Cruft: Large, convoluted classes that had grown crusty
  - Too many cooks in the kitchen: Developers had applied the library du' jour and their unique style and approach over the years.
  - Result was old, bloated, obscure and fragile code.
- State and cache managed by Hibernate
- UI was OK, if a little dated.
- Some screens took 4s to 8s to load. One took 52s to 108s!

Old System: Middle Tier

- Java classes for UI, business and data layers
- Data layer Java used Hibernate…poorly.
  - Gigantic, slow object graphs loaded through Hibernate-generated HQL queries, holding most information needed for each missionary across all the pages in the app.
  - Hibernate wasn't finished inflicting its features upon us. HQL it issued was not in our control, awful, hard to debug and change.
  - For the worst page, on top of really complex business logic, we found Hibernate was issuing over 600 SQL statements under the covers.

Old System: Backend

- Oracle held the core missionary database tables, and a few tables unique to the senior missionaries.
- Five of the most complex queries were turned into views.
- Other than the views, nothing was abstracted or encapsulated.
- Oracle treated like a generic bit-bucket, most of the rich features and abilities of Oracle were ignored.
- Table names and column names were read, mapped and exposed by Hibernate, leaving the system tightly coupled and vulnerable to change.

The Tour

- Background
- Old System
- **The Challenge**
- Solution
  - Angular Angles
  - ~~ORMmmmm~~ Lightweight Services
  - Data Done Delightfully
- Lessons Learned

The Challenge

- Initial design time from June to August was well-spent.
- But 2.5 months were burned trying to refactor as-is and the POC.
- It was now the middle of October and the project was scheduled for completion end of December.
- Should have had 4 months to integrate the new concepts and business process, modifying a few pages in the process.
- Now we only had **2 months** left to rewrite 80% of the application.
  - Young client developer was sure we could make it
  - Seasoned Java and services developer estimated 5 months

The Challenge

- Unfortunately, the DBE was not invited to participate during POC
- I assumed that the developers' research and recommendations included the whole application stack, especially how to handle data access and changes
- Instead, the DBE was asked for data layer recommendation on the day the POC was to end!
  - Major panic and hyperventilation
  - Managed to buy us a week to investigate options
- Turns out the modern SPA architecture does not dictate a standard approach to data access.
- Oh no.

The Tour

- Background
- Old System
- The Challenge
- **Solution**
  - Angular Angles
  - ~~ORMmmmm~~ Lightweight Services
  - Data Done Delightfully
- Lessons Learned

Client: Angular Angles

- Java/JavaScript developer who championed the rewrite was very bright and enthusiastic, but inexperienced and too optimistic.
  - Original claims were ¼ hour to 2 hr re-write effort per page!
  - Estimates weren't even close. Real-world dev time to re-write was the typical 2 to 5 days per page, depending on complexity.
- Deep corners and angles of Angular weren't considered or estimated.
  - Data grid/list widget almost scuttled the whole project. Wasted 7 weeks.
  - Was tricky to get loading spinner right.
  - Grunt and Gradle and SSO headaches.
  - Cache http requests. Caching patterns not known during POC.

Client: Angular Angles

- Angular 1.X (2014) somewhat immature. Our enterprise framework team had already abandoned it and eagerly awaited Angular 2.
- Did it need to be a stateless SPA app?
  - Going stateless is fantastic for scalability.
  - But this app was only used by a handful of users at HQ.
  - Benefit of web access questionable. Cannot foresee it ever being mobile.
  - In fact, being a heavy data-entry app, probably should have been a fat client.
- **Stateless** means that, unless kept in the client, the old state of the data will not be known when a POST or PUT request is issued. This means the data must be re-read, potentially doubling DB roundtrips.

Client: Angular Takeaways

- POC should cover all major components/features required by the app.
- Newer tech requires trial and error and sometimes scrap and re-do.
- Easy to do things the wrong way. Maintain discipline.
- Split client and service layer into their own projects. Numerous benefits from this.
- Lots of built-in goodness inherited from the internally-developed web app framework and Spring frameworks (Spring Boot, REST, & JDBC).
- Angular is awesome for managing JavaScript libraries

Client: Now

- Even though stateless wasn’t a necessity, we are glad we went that route.
- UI is now
  - Cleaner
  - Looks and feels “modern”
  - More intuitive
  - Responsive in several ways
  - Fast
  - Modular
  - Easy to understand and maintain

Moving on to the middle tier, we come to Object-Relational Mapping.

Chant with me now

ORMmmmmm…

Just kidding

I never want to see another heavy ORM as long as I live

Middle: Data Layer Choices

- ORMs have been called the Vietnam of Computer Science
- Other articles call them anti-patterns, “the devil” and otherwise show no love
- Devs so fed up with the difficulty of fusing object and data domains, they invent new data access frameworks almost daily
- Can be done right, but usually requires an ORM expert

I read around 100 articles and forum threads to survey the current state of ORM use. One contributor summed it up nicely :

> “ORMs are great for simple CRUD operations. As soon as you want to do anything mildly complex or desire efficiency, you need to write SQL.”

Middle: Data Layer Choices

Even Gavin King, the creator of Hibernate, agrees its adoption was overdone:

"Just because you're using Hibernate, doesn't mean you have to use it for everything. A point I've been making for about ten years now." - Dec 9, 2013 Gavin King’s Google+ feed

Middle: Data Layer Choices

- Like most shops that went nuts with ORMs between 2005 to 2010, this app was plagued by the use, misuse and overuse of Hibernate.
- Our devs decided early to remove as much of Hibernate as possible.
- Hurray!
- But…

Middle: Data Layer Choices

- But when you remove an ORM, its basic functionality still needs to be handled by something.
- Even at their worst, big ORMs accomplish three basic things:
  1. Allow for rapid prototyping (irrelevant to this study)
  2. Manage entity relationships (map between object model & data model)
  3. Manage transactions and data changes
  4. Hibernate also gave us an easy caching mechanism which was heavily used

Middle: Data Layer Choices

- Being an Angular SPA app the use of RESTful Java services and JSON for the middle tier was a given. The POC had focused instead on:
  - Client-centric services for marshalling JSON data to and from the Angular UI
  - Business-logic services
  - Initial Angular pages and integration with our core app stack
- But the devs running the POC had forgotten about:
  - **Data access services**
    - **INSERT, UPDATE, SELECT, DELETE**
    - **Transactions**
    - **Stale data and changed data detection**

Middle: Data Layer Choices

- Exorcising the ORM meant we now had to manage relationships and CRUD operations in some other way. Nobody in the SPA community or our skunkworks team agreed how that should be done.
  - Which tier would build the simple and complex SQL statements?
  - Which tier would detect stale (aka dirty) data?
  - How should stale data detection be communicated and handled?
  - Which tier would detect and manage data changes?
  - How would optimistic locks be handled?
- These questions had to be answered in a week and became the genesis of this presentation.
- We began with all the options available at the time…

Middle: Data Layer Choices

- Custom POJOs and raw JDBC. *Fast execution, but slow to code.*
- EJB: *Ick. Nope. Never again.*
- Big ORM: *Can be done right. Usually isn't. Big nope.*
- Thin ORM: *Seemed viable. Little experience with them. Not approved.*
- SQLAlche[illegible]ntity Framewo[illegible] *Too new. Zero experience with these.*
- We chose our internal standard: Spring Framework [Boot, REST, JDBC]
- Our forward-thinkers: SOA (enterprise service bus of data services which abstract and secure queries, views, & stored procedures)

Middle: Data Layer Choice of Spring

- The data services would use Spring JDBC to talk to Oracle, but leave the heavy lifting to Oracle views and stored procs
- Discussed existing and new embedded SQL queries but devs and DBE agreed that SQL belongs in the database. (More on this in a moment.)
- DBE re-used existing views and moved embedded SQL and translated HQL into new stored procedures that abstract, encapsulate and protect the raw data structures and complexities of SQL queries.
- SELECT procs would return Oracle REFCURSOR (pointer to result set)
- Data services do row-mapping of DB call results to Java objects, and use Spring REST to marshal data to and from JSON objects consumed by Angular client.

DB: Data Done Delightfully

With the choice to leave SQL resident in the database, all that remained was deciding how to handle stale/changed data detection and the locking strategy.

DB: Data Change Detection

For stale and changed data detection, we had various options:

- Detection in the client
  - Client uses local storage to keep state in the browser and track what actually changed. DML done in middle or DB tier. DML layer must re-read DB for stale.
  - Loved this option. Wanted to use new HTTP PATCH verb, but no support then.
  - Developer swamped with Angular problems. Punted.
- Detection in the app server
  - Every field passed to and from client. Service tier re-reads DB for stale and compares current state with every field from client. DML can be here or in DB.
- Detection in the database
  - Every field passed from client. Still re-read DB for stale and data changes, but very fast as it avoids the network and object creation/population. SQL dead simple, instrumented, written by experts, monitored, tuned, etc. This became our best option given our constraints.

DB: Data Locking

- To ensure that newer data is never overwritten by older data, you can choose to pessimistically lock or optimistically lock.
  - Very few reasons for pessimistic, unless records are frequently changed at the same time by multiple users. These locks serialize data access and cannot scale.
- Optimistic locking assumes overwrite condition rarely happens.
  - When the data is first read and passed to the client, no lock is obtained. Instead a hash, version number, timestamp or similar signature based on the current state is passed along with the data.
  - Upon update, the program compares the current state of the data in the database to what it was when sent to the client. If the current state is newer, the client changes are not allowed to proceed.

DB: Data Locking

Various methods can be used to determine if the client's data is stale:

- Full re-select with serialized lock and new values passed in from the UI.
- Update of new values where the current database values match the old values passed back from the UI. If the update finds no rows to update, it is because the values in the row have changed since it was first read.
- Timestamp or counter that is maintained by app or trigger. Passed to UI during read. UI passes it back during write for comparison to current value.
- Checksum or hash generated using OWA_OPT_LOCK.checksum(), ORA_HASH, DBMS_CRYPTO.HASH or DBMS_SQLHASH.GETHASH.
- ORA_ROWSCN pseudocolumn.
  - Table must be created or re-organized with the ROWDEPENDENCIES clause, otherwise rows in same block share the same ORA_ROWSCN.

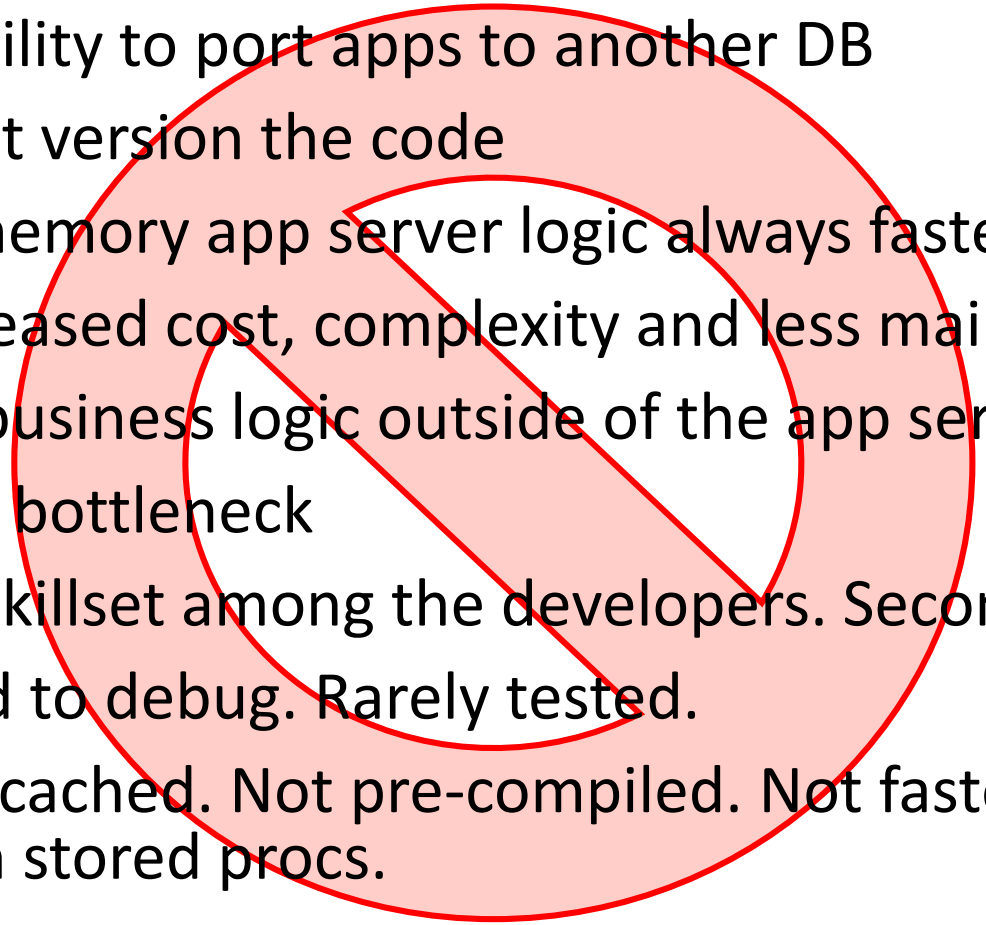
DB: Data Locking

- Personally prefer the timestamp as it gives us 3 for the price of 1:
  - we know the row has changed
  - we know when it changed
  - can serve double duty as both audit and versioning column
- In our case, we already had a numeric version column in place from Hibernate forcing it upon us. So timestamp was sadly jettisoned.
- We pass a counter (column jpa_version) back within query results. Client returns this counter value back to DB when updating data.
- Update API re-reads data, compares live counter to counter given by client, and raises exception with message if client data is stale.

DB: Data Access API

- With our data access strategy designed, just had to convince management.
- The biggest changes were removing Hibernate and moving SQL into PL/SQL APIs.
  - No trouble with Hibernate. Management had too many bruises. They wanted it gone too.
  - But very few PL/SQL cheerleaders. Their opinions about PL/SQL were based on hearsay from the uninformed.

DB: Arguments against PL/SQL



- Inability to port apps to another DB
- Can’t version the code
- In-memory app server logic always faster
- Increased cost, complexity and less maintainability
- No business logic outside of the app server
- DBA bottleneck
- No skillset among the developers. Second lang for app.
- Hard to debug. Rarely tested.
- Not cached. Not pre-compiled. Not faster. Dynamic SQL as fast or faster than stored procs.

DB: Arguments for PL/SQL as Data API

- Home for SQL in the DB where it belongs
- Agility when changing app development language/framework
- Continuous and easy deployment
- Speed and efficiency
- Set/Bulk transactions and processing
- ETL
- Security
- Abstraction and de-coupling
- Centralization of common logic and reusability
- Automation of routine data-centric tasks
- Built-in and bolt-on auditing and instrumentation

See Bryn Llewelyn's excellent Why use PL/SQL? for great pro-PL/SQL points

PL/SQL: SQL Placement

- Can feel like a religious debate, but there is only one truth:
  - All SQL should be kept behind an abstraction layer in the database (views, packaged cursors or packaged PL/SQL routines)
- SQL should be written by someone that understands relational, sets, SQL and your database's capabilities.
- Reviewed, tested and tuned by DBAs
- Can be easily instrumented for logging and debugging. Much easier to troubleshoot and monitor when kept inside the database, next to the data, where it belongs.
- Much more likely common SQL will be cached and re-used

PL/SQL: App Evolution

- How often do apps change a framework they're using? (POJO -> EJB -> Struts -> Ajax -> JSF -> Angular -> ?)
- Do any of them share the same DB access methods?
- How often do apps change the database?
- Keeping the SQL and data logic in the database means much less work when re-tooling an app

PL/SQL: Security

- If business requires stiff protection of data structures, don’t grant them to any accounts.
- Construct PL/SQL API for data access and manipulation
- Design security scheme for the stored procs that have access to the tables and views.
- Use DBMS_ASSERT to test for valid inputs and avoid injection
- Use new white-listing to further restrict access to PL/SQL routines

PL/SQL: SQL Injection-proofing

- SQL written by devs and kept in the middle tier is much more likely to suffer from predicate concatenation, performance, shared pool and security issues.
- SQL behind a PL/SQL interface guarantees no SQL injection (unless dynamic SQL is being written with parameters), and written by experienced DBAs, it is more likely to be correct and fast.

PL/SQL: Deployment

- Being an interpreted language, PL/SQL is very simple to deploy.
  - Compile the versioned source file into the schema. Done.
- Dependency model ensures package state isn't invalidated unless absolutely necessary.
  - App data layer should be amended to capture the "existing state of packages has been discarded" error and re-try the previous call.
- Abstract structures using views, ref cursor OUT parameter for query routines, and record-based DML.
- Voila! Self-adjusting code that can be continuously deployed to Prod

PL/SQL: Data Proximity & Network Latency

- Typical: Query DB, transport result set across the network, load objects, perform the operation(s) on the data in memory, unload objects, transport data back to DB, update and commit.
- Stored Routines: Call routine, query DB, perform operation(s) on the data in memory, update and commit.
- Eliminating the items above in red saves significant runtime when dealing with large result sets.

PL/SQL: Large Data Sets

- RDBMS are set-oriented
- Oracle is awesome at querying and manipulating large, relational result sets.
- Don’t treat DB like “dumb” persistence box
- Take advantage of the set and performance-oriented features of your DB
  - Doing as much as you can as one SQL statement
  - Partitioning and parallel processing
  - Pipelined functions
  - If PL/SQL is the solution, use bulk features BULK COLLECT and FORALL

PL/SQL: Extraction

- Use PL/SQL to extract data and return to client or write to file or other output mechanism.
  - This is especially appropriate if the extracting SQL query is complex and the resulting data is large.
- Pipelined functions deliver data to client of function as fast as possible

PL/SQL: Transformation & Loading

- If requirements need to pump large amounts of data into the database, in particular if calculations or derivations need to read the destination database while processing incoming data, do it in the database using external tables and pipelined PL/SQL functions.
- Can use EXTERNAL TABLE option and avoid copying/loading the data entirely

PL/SQL: Common Business Logic & Re-use

- Database processes, jobs, routines, triggers that need to use common business logic can't take advantage of web services or middle tier methods, which demands duplication and fragility.
- Critical algorithms that must be available to and used by multiple systems and tiers should be kept in the least common denominator (the database) as a stored routine.
- Design the PL/SQL interface well. Document it well. Publish it.
- If direct access to the DB is not available, the stored routine can be published as a web service just like any Java-based web service.

PL/SQL: Auditing & Logging

- SQL in the middle tier is notoriously done poorly (by devs or JPA engine).
- When things go wrong, it is difficult to debug, monitor and log.
  - Can be done right, but usually isn't
- SQL kept in stored packages trivial to instrument, monitor, debug, tune and re-deploy.

PL/SQL: Automation

- Most common use of PL/SQL
- Used often by physical/system DBAs to schedule routine reports, extractions, loads, monitoring tasks, etc.

PL/SQL: Final Considerations

- Time
- SQL expertise
- PL/SQL expertise
- Size and complexity of data model
- Portability
- Scalability and Performance
- Maintainability/Tuning/Troubleshooting

PL/SQL: Result

- Management agreed the data access, data locking and data change detection could be done in a database-resident PL/SQL API.

Case Study Results:

New System {code walkthrough, time permitting}

- UI is Angular 1.3 calling RESTful services
- RESTful Spring-based Services in Tomcat
- Oracle views and PL/SQL API for reporting, queries and updates
- Keeping static and dynamic SQL in views and procs very beneficial
  - One example: Able to test procs before REST services were ready. And were able to test REST services in browser before UI was done.

Case Study Results

client

Language	files	blank	comment	code
Javascript	128	900	653	7996
HTML	132	371	327	4599
SASS	31	395	40	1722
JSON	2	0	0	90
YAML	1	0	0	6
SUM:	294	1666	1020	14413

REST services

Language	files	blank	comment	code
Java	359	9203	4086	34371
XML	9	48	257	601
YAML	7	79	59	289
Maven	1	10	27	118
SQL	3	9	18	58
Bourne Shell	1	0	0	2
DOS Batch	1	0	0	2
SUM:	381	9349	4447	35441

Case Study Results

Link/Page	Filter Used	Avg Time (Old Sr. Asmt)	Notes (Old Sr. Asmt)	Avg Time (Jan 26)	Avg Time (Apr 17)	Notes (Finished Sr. Asmt)
Prepare Sr. Assignments (Candidate List)	Default (Not Finalized and all msny types)	6s		7 to 8s	5-6s 2s	Scrolling is now slower (necessary evil?) The 1s time is from Return to List from Summary.
Profile	enamem_id	4.5s		2 to 3s	2 to 3s	As fast. Old Hibernate left in place. Should be simplified with new architecture in future enhancement. As Ryan Allen found out with Screening, having two frameworks in place with two different transaction-management models, is not a good idea.
Recommendations	enamem_id	2.5s (0 rcmd) 3s (>0 rcmd)		2 to 4s	<=1s (0 rcmd) <=3s (> 0 rcmd)	A little slower with recommendations, but more accurate and functional too.
Summary	enamem_id	2.5s		2 to 4s	<=2s	As fast and more functional than old.
Needs List	Typical candidate default filters: All, 6 mo, $2000, Good care, LA - Exclude RV	6s		6 to 8s	<=6s (3.5s DB)	Slower due to far more "slot" needs and functionality. List of 227 needs.
	All, 12mo, $3000, Good care LA - Exclude RV	8s		6 to 8s	8s (6.2s DB)	Slower due to far more "slot" needs and functionality. List of 1027 needs.
	All (no high/low), no $, Poor care, LA - Include RV	13s		6 to 8s	15s (14.1s DB)	Slower due to far more "slot" needs and functionality. List of 4041 needs.
Add Rcmd: Select Training	Default	1 to 2s		1 to 2s	<1s	Fast, simple, better.
	All curriculum	2s	Found bug in attempting to go back to just the default curriculum.	n/a	<1s	Fast, simple, better.
Add Rcmd: Select Dates	12mo	52s to 108s	12mo is the default. Got error on first attempt after 108s	1 to 2s	1s	So sweet now. Reloading takes 3.5s, which is fine.
	12mo	35s	Reload previous	1 to 2s		
	6mo	60s	Doubled time asking for half the data. Odd.	1 to 2s		
	No training selected, 12mo	5s	Speedy. Shows radio button on days in the past for current week.	<1s		

The Tour

- Background
- Old System
- The Challenge
- Solution
  - Angular Angles
  - ~~ORMmmmm~~ Lightweight Services
  - Data Done Delightfully
- **Lessons Learned**

Lessons Learned

- Consider all UI, middle tier and backend components up front. Ensure application architecture is complete before estimations and recommendations.
  - SPA UI: Responsive design, navigation and paging, lists and grids, offline storage, etc.
  - Middle: App server, frameworks, REST, JSON, ORM, input sanitation, security, etc.
  - Backend: Persistence mechanism. Optimistic locking. Dynamic SQL. Updates.
- Question estimates that don’t feel right, especially if the tech is bleeding edge.
- If you do use an ORM (JPA), hire an ORM expert or model in the small
  - Grab only the data needed for the current page. Small object graphs that do one thing and one thing well. Consider micro services instead.
- Moving the SELECT SQL behind a PL/SQL API has many benefits

Lessons Learned

- Use timestamp or version column for optimistic locking
- Let a proc handle stale data detection
- Let middle tier handle changed data detection and generation of dynamic UPDATE statement.
- Use quantities and dollars with mgmt when arguing to keep SQL in the DB and removing abominations like Hibernate.
- Use the Agile practice of involving the users early and often
  - Seeing the cleaner, faster functioning initial screens motivated them to pay for and approve the additional months we needed to finish the rewrite.


ODTUG
Kscope16
CHICAGO, ILLINOIS · JUNE 26-30
PLEASE FILL OUT YOUR EVALUATIONS