
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: dbartisans.com & dbsherpa.com (soon)
- Twitter: @billcoulam
- Email: bill.coulam@dbartisans.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

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.

The Tour

- **Background**
- Old System
- The Challenge
- Solution
  - Angular Angles
  - ORMmmmm
  - 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.

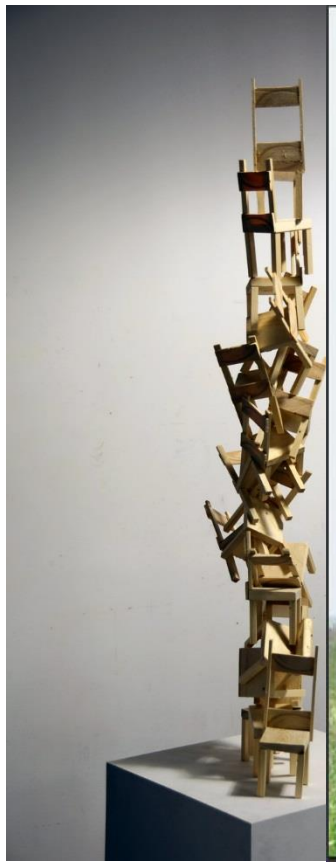
Application architecture mainly based on Java JSF, Hibernate and Oracle tables and views. Old, stateful, bloated, slow and crusty.

Case Study Background

- Due to a process redesign, new business requirements and desires for new features, we were given permission to rework pieces 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.
- Promised management we would get done in the same amount of time, or less, as original budget.
- Given permission to it as a modern thin, 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
  - Data Done Delightfully
- Lessons Learned

Old System: UI

- Web pages generated by JSF and Struts.
  - Large, convoluted classes that had grown crusty over the years.
  - Every developer that had a hand, applied their own unique style and approach to things.
  - Results were old, difficult to read and digest code.
- State 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, using 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
  - 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 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 dear.
- To the starting line…Runners take your marks! Go!

The Tour

- Background
- Old System
- The Challenge
- **Solution**
  - Angular Angles
  - ORMmmmm
  - 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 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.
- Devs decided early to remove as much of Hibernate as possible.
- But even at their worst, ORMs still accomplish three basic things:
  - Allow for rapid prototyping (irrelevant to this study)
  - Manage entity relationships (map between object model & data model)
  - Manage transactions and data changes
  - (Hibernate also gave us an easy caching mechanism which was heavily used)
- When you remove an ORM like Hibernate, its basic functionality still needs to be handled by something.

Middle: Data Layer Choices

- Being a SPA app the use of RESTful Java services for the middle tier was a given.
  - Client-centric services for marshalling JSON data to and from the Angular UI
  - Business-logic services
  - Data access services
- How we would approach managing data (with the ORM out of the picture) was up in the air.

Middle: Data Layer Choices

- Removing the ORM meant we now had to manage relationships and CRUD operations in some other way. Nobody in the industry 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.*
- Our current 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

- 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.
- Query 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 yet.
  - 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 old/new values passed in from the UI.
- Update of new values where the current database values match the old values passed in 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 trigger. Passed to UI during read. UI passes it back during write for comparison to current value.
- Checksum generated using OWA_OPT_LOCK.checksum(). Uses row's ROWID to access the column values.
- ORA_ROWSCN function.
  - 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 version column back within query results. Client returns this to DB when updating data.
- Update APIs re-read data, compares latest version to version given by client, and raises exception with message if client data is stale.

DB: Data Access API

- With Hibernate removed and our strategy decided upon to replace its core functions, all that remained was to convince management that it was worth our time to move SQL behind views and PL/SQL APIs (packaged functions and procedures)
- Very few PL/SQL cheerleaders. Management all too often gets their opinion second-hand from the jaded and uninformed.

DB: Arguments against PL/SQL

- Inability to port ther DB
- Can't vers
- In-mem erver logic a aster
- Increa lexity and aintainability
- No b s logi e of the a rver
- DBA neck
- No sk mong the d ond lang for app.
- Hard to Rarely teste
- Not cache ot faster. Dynamic SQL as fast or faster than stored pr

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

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
- See Bryn Llewelyn's excellent Why use PL/SQL? whitepaper

PL/SQL: SQL Injection-proofing

- SQL written by devs and kept in the middle tier is much more likely to suffer from concatenation problems, hurting performance, shared pool and inviting SQL injectors
- SQL behind a PL/SQL interface guarantees no SQL injection (unless dynamic SQL is being written with parameters)

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

Case Study Results:

Old System

- UI produced by JSF/Struts-based Java classes running in app server
- Hibernate ORM
- Persistence to Oracle as a dumb black box

New System {live 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

The Tour

- Background
- Old System
- The Challenge
- Solution
  - Angular Angles
  - ORMmmmm
  - 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, etc.
  - Backend: Persistence mechanism.
- Question estimates that don't feel right, especially if the technology is bleeding edge.
- If you do use an ORM, 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.
- Moving the SQL behind a PL/SQL API benefits everybody
- Where you put the INSERT, UPDATE and DELETE APIs is a toss-up
- 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.