Dev 2.0:
Living in the World of APIs

June 5, 2019

Grigoriy Novikov, Michael Rosenblum

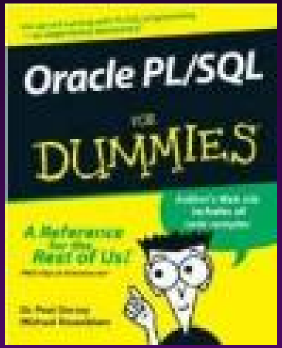
www.dulcian.com

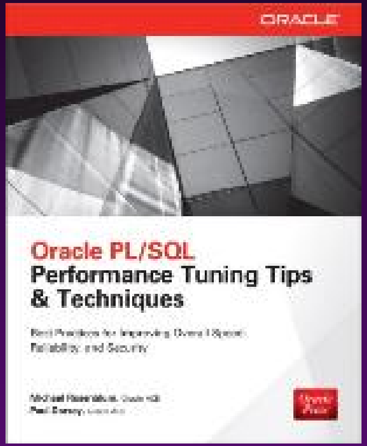
Who Am I? – "Grisha" (aka Grigoriy Novikov)

- OCA with 15+ years in Oracle development
- Areas of responsibility
  - Technical Leader for multiple HIPAA SaaS products
  - Participated in many government projects
- Loves new projects!

Who Am I? – "Misha" (aka Michael Rosenblum)

- Oracle ACE
- Co-author of 3 books
  - *PL/SQL for Dummies*
  - *Expert PL/SQL Practices*
  - *Oracle PL/SQL Performance Tuning Tips & Techniques*





- Known for:
  - SQL and PL/SQL tuning
  - Complex functionality
    - Code generators
    - Repository-based development

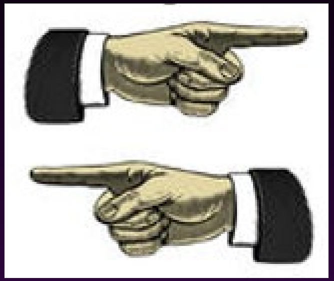
APIs

- Application Programming Interface:
  - Definition: Set of clearly defined methods of communication among various components
  - Introduced: 1968 and (ab)used ever since

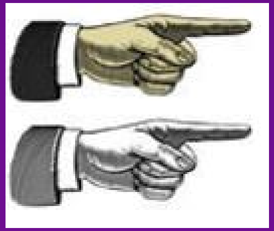
APIs?

- You can call them SOA/Micro-Services/new buzzword
  - … but they are somebody’s tool being used by somebody else
    ➔ Crossing boundaries is the key!
- Crossing boundaries always means crossing areas of responsibility
  - … but every issue should have a name assigned to it
    ➔ Higher management/control requirements

APIs…



- More involved parties = more “blame game”
  - … so, backside covering is the most critical survivability factor
- Service Level Agreements (SLA) are written by lawyers for lawyers.
  - … so, normal techies rarely understand what is/is not covered
- Efficiency is often the first victim of being “bullet-proof”
  - … so, performance tuning is viewed as an afterthought.



APIs!

- System tuning in any API-based system is very complex
  - … and often involves direct management intervention.
- You cannot build contemporary systems without APIs
  - … because too many moving parts are involved.
- API-based systems have to be properly built from the very beginning
  - … since architectural solutions are always more efficient than purely technical ones.

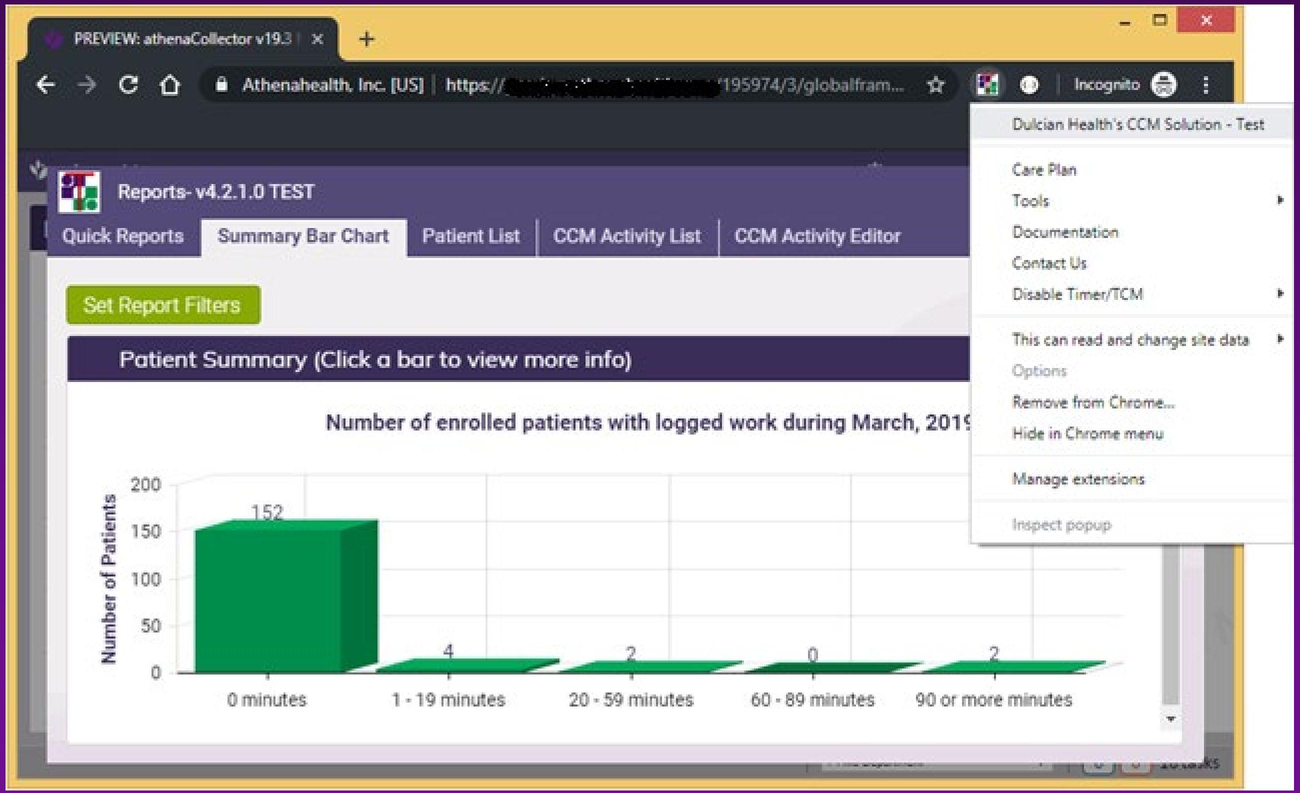
So?

- This presentation IS NOT about:
  - Finding "_RUN_FAST=TRUE" somewhere in undocumented list of parameters
  - Writing the most efficient APIs ever invented
- This presentation IS about:
  - Finding and solving real-world challenges of API-based systems
  - Making your system architecture API-friendly
  - Surviving when your system depends upon others

Real World System Example

About the System

- ◆Seamless integration with EHR
- ◆Provides additional functionality
- ◆Implemented as a Google Chrome Extension
- ◆API and SSO
- ◆Oracle + Formspider IDE



System Statistics

- ◆400,000+ API calls per day
- ◆~50 API calls per second during peak hours
- ◆~20 API calls per second (usual workload)
- ◆~1,000 active users every day
- ◆~500 active simultaneous **logical** sessions
- ◆~40 app server requests per second

System Structure

- Application server
  - 2 CPU
  - 8 GB RAM
- Database Server
  - 4 CPU
  - 16 GB RAM
- Stateless architecture

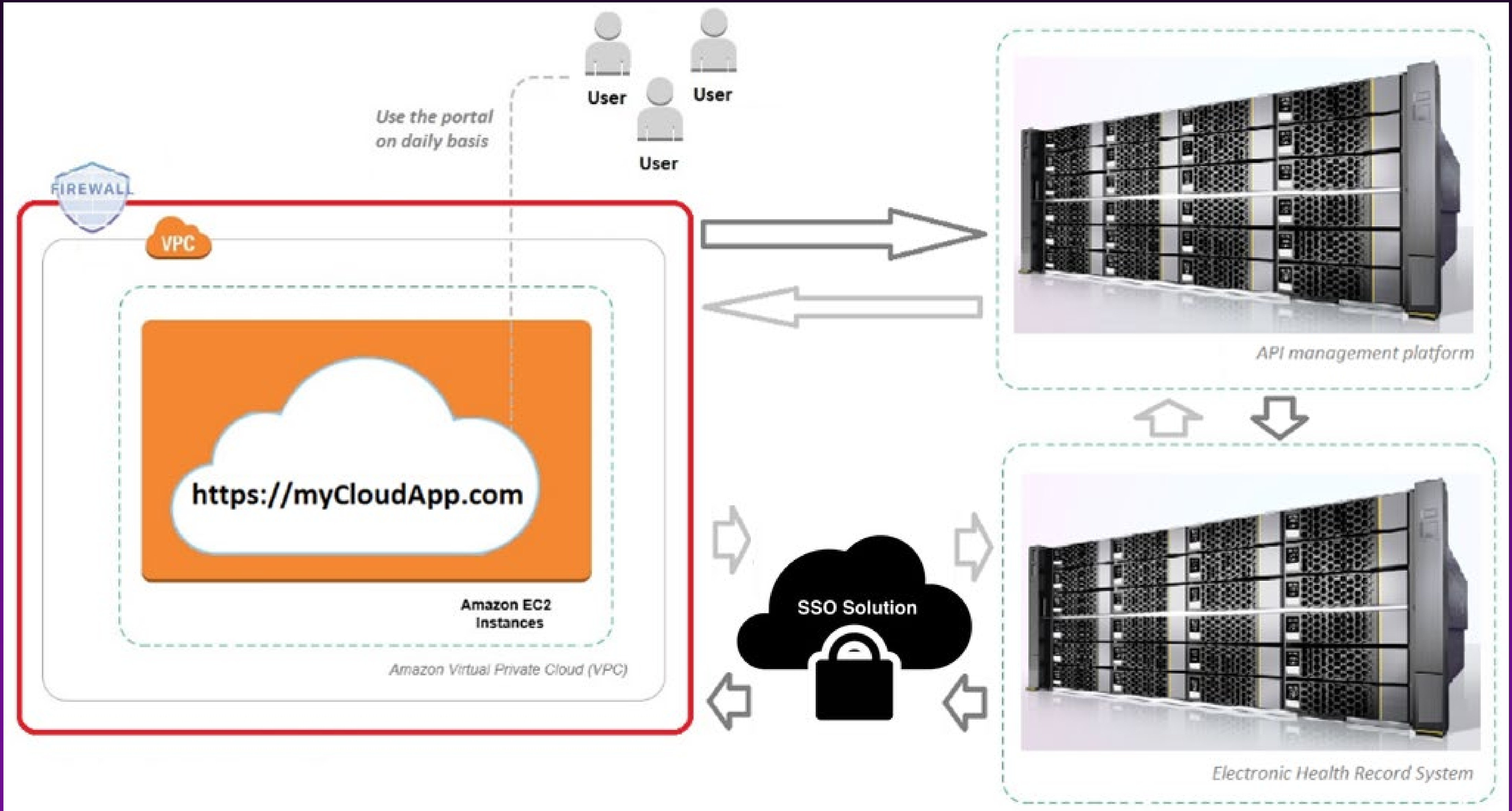
Stateless Architecture

- Core concept:
  - "Session" = set of activities between logon and logoff.
- Problem:
  - Rules applicable to 100 connections didn't work for 100,000 connections.
- Alternative:
  - Introduce logical/physical session separation

Why bother?

StateFULL Architecture	*StateLESS* Architecture
◆Logical session = Physical session ➢ … meaning lots and lots of database connections (irrelevant whether anything happens) → ▪ Risk: idle hardware ▪ Benefit: predictability	◆1 Logical session = 1..* Physical Session ➢ … meaning database connections are opened only as needed (to serve requests) → ▪ Risk: workload peaks ▪ Benefit: cost efficiency

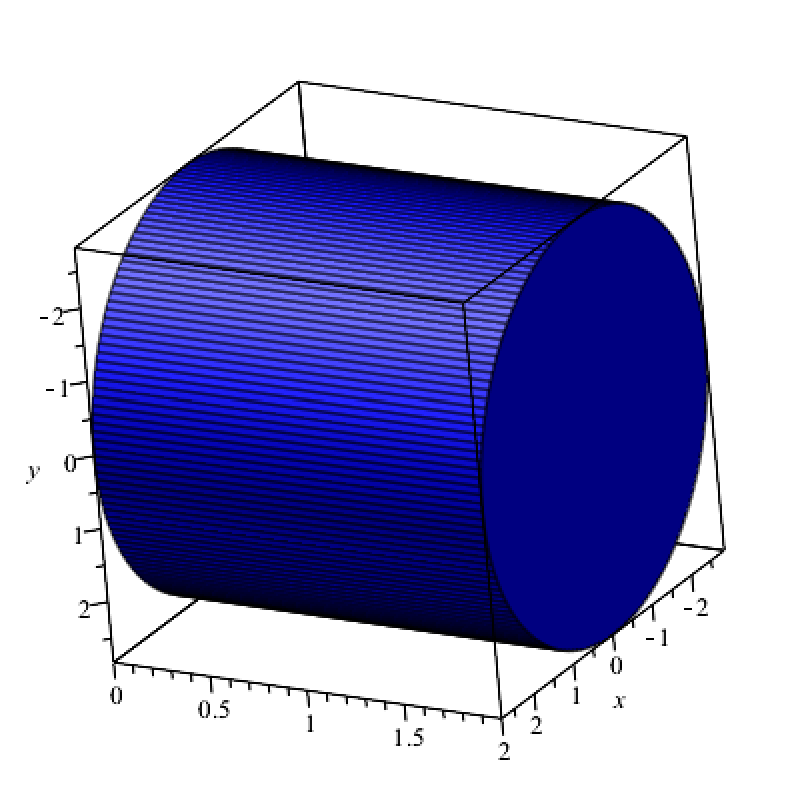
System Architecture



Real Use Cases!

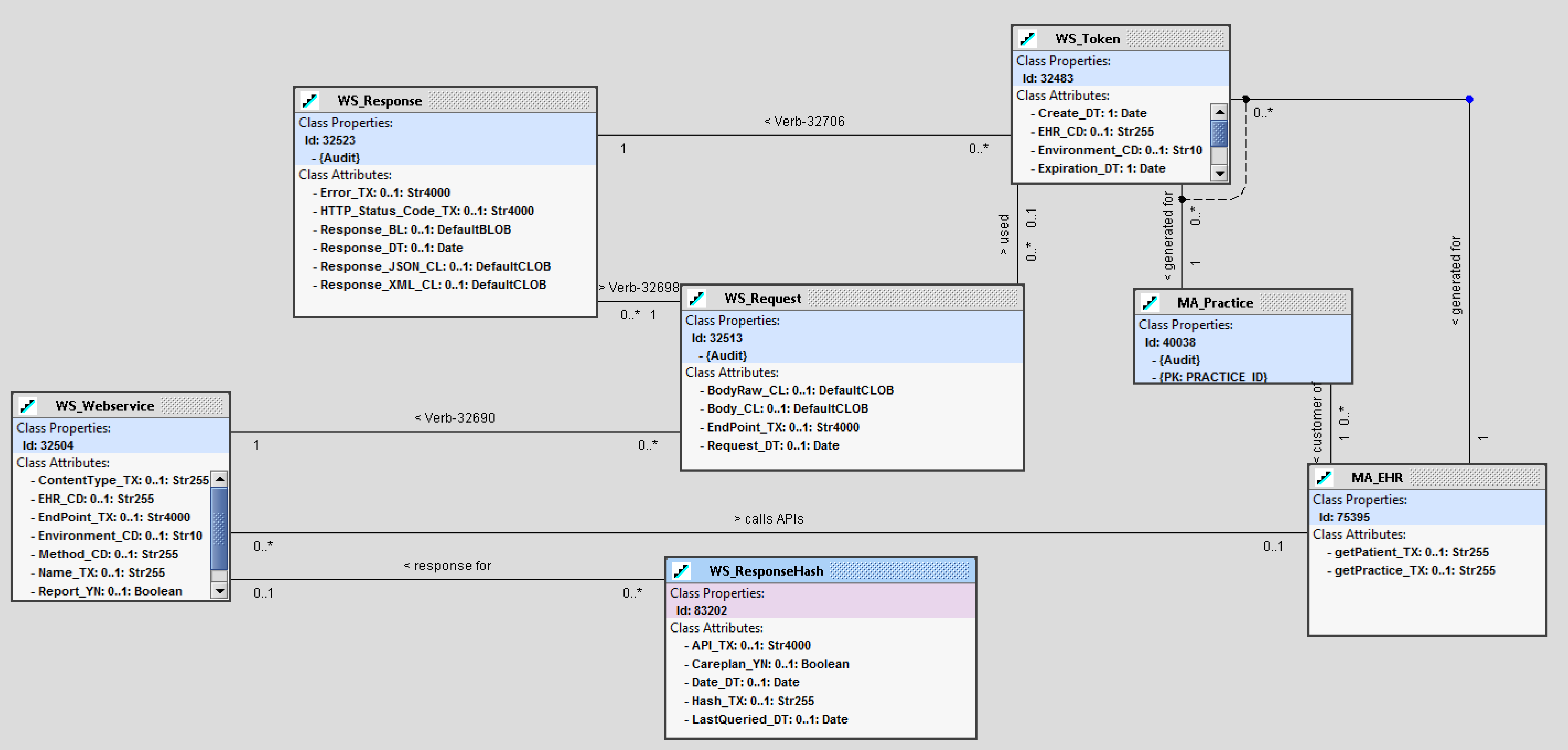
Dealing with Volume

- Data model
  - API footprints
  - Some tips
- Volume
  - How much is enough?
- Hashing
  - Save on resources
- Inactive clients
  - Regulations and contracts



Dealing with Volume: Data Model

- Data model – store all fingerprints



WS_Token
Class Properties:
Id: 32483
Class Attributes:
- Create_DT: 1: Date
- EHR_CD: 0..1: Str255
- Environment_CD: 0..1: Str10
- Expiration_DT: 1: Date

WS_Response
Class Properties:
Id: 32523
- {Audit}
Class Attributes:
- Error_TX: 0..1: Str4000
- HTTP_Status_Code_TX: 0..1: Str4000
- Response_BL: 0..1: DefaultBLOB
- Response_DT: 0..1: Date
- Response_JSON_CL: 0..1: DefaultCLOB
- Response_XML_CL: 0..1: DefaultCLOB

WS_Request
Class Properties:
Id: 32513
- {Audit}
Class Attributes:
- BodyRaw_CL: 0..1: DefaultCLOB
- Body_CL: 0..1: DefaultCLOB
- EndPoint_TX: 0..1: Str4000
- Request_DT: 0..1: Date

MA_Practice
Class Properties:
Id: 40038
- {Audit}
- {PK: PRACTICE_ID}

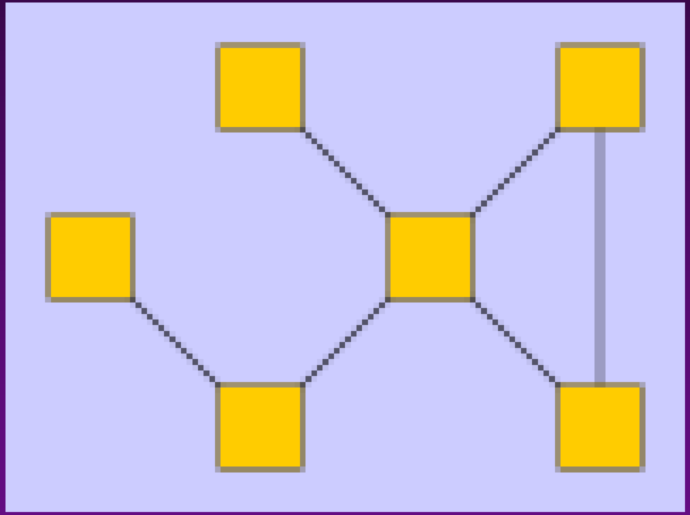
WS_Webservice
Class Properties:
Id: 32504
Class Attributes:
- ContentType_TX: 0..1: Str255
- EHR_CD: 0..1: Str255
- EndPoint_TX: 0..1: Str4000
- Environment_CD: 0..1: Str10
- Method_CD: 0..1: Str255
- Name_TX: 0..1: Str255
- Report_YN: 0..1: Boolean

MA_EHR
Class Properties:
Id: 75395
Class Attributes:
- getPatient_TX: 0..1: Str255
- getPractice_TX: 0..1: Str255

WS_ResponseHash
Class Properties:
Id: 83202
Class Attributes:
- API_TX: 0..1: Str4000
- Careplan_YN: 0..1: Boolean
- Date_DT: 0..1: Date
- Hash_TX: 0..1: Str255
- LastQueried_DT: 0..1: Date

Dealing with Volume: Data Model (2)

- Request
  - Web service method (GET, PUT, POST…)
  - Endpoint
  - All parameters including binary data
  - Timestamp
  - Environment (DEV, TEST, DEMO, PROD)
- Response
  - Full response
  - Response code
  - Timestamp
  - Error (API vs. System)



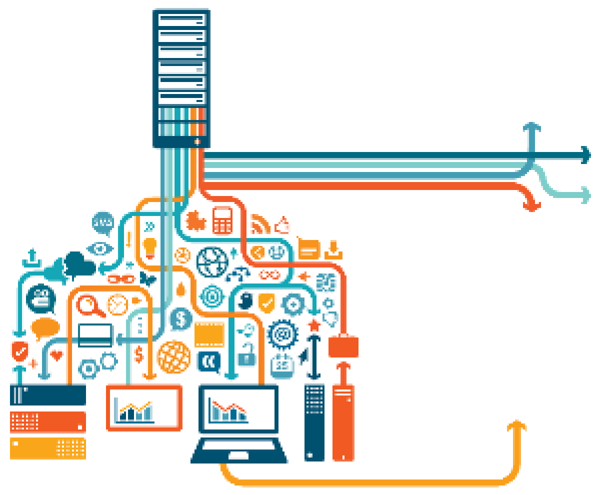
Dealing with Volume: Data Model (3)

- Tokens
  - API provider-specific
  - Client specific

- TIP: Store the last response separately (and associated with the core object)

Dealing with Volume: Use Case 1

- How much data do you really need?
  - Let’s grab it. We’ll decide later what to do.
- Original solution:
  - Request data from external source as much as possible
  - Synchronize your system with external source
- Pros:
  - All data up-to-date (almost)
  - Users run the reports in your system
- Cons:
  - A lot of API calls (and $$$)
  - You will always be one (or more) steps behind
  - Room for errors

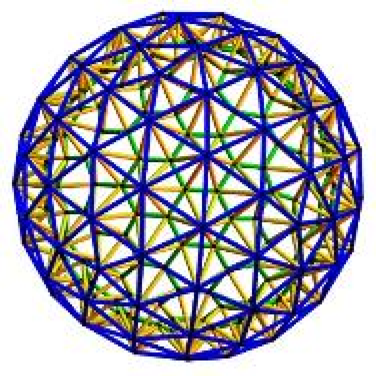


Dealing with Volume: Use Case 1 (cont.)

- Optimized solution
  - Request data only for objects of interest
  - Keep only important data synchronized
- Pros
  - Fewer API calls
  - Smaller workload (and less $$)
- Cons
  - You are still one (or more) steps behind
  - Room for errors

Dealing with Volume: Use Case 2

- Can we go even further?
  - Response can be quite complex
- Hash and cache the response
  - `update t_responseHash`
    `set hash_tx = hash(`*`response_object`*`)…`
- Calculate response hash and compare with previous one
- Be aware of the response timestamp



Dealing with Volume: Use Case 2 (cont)

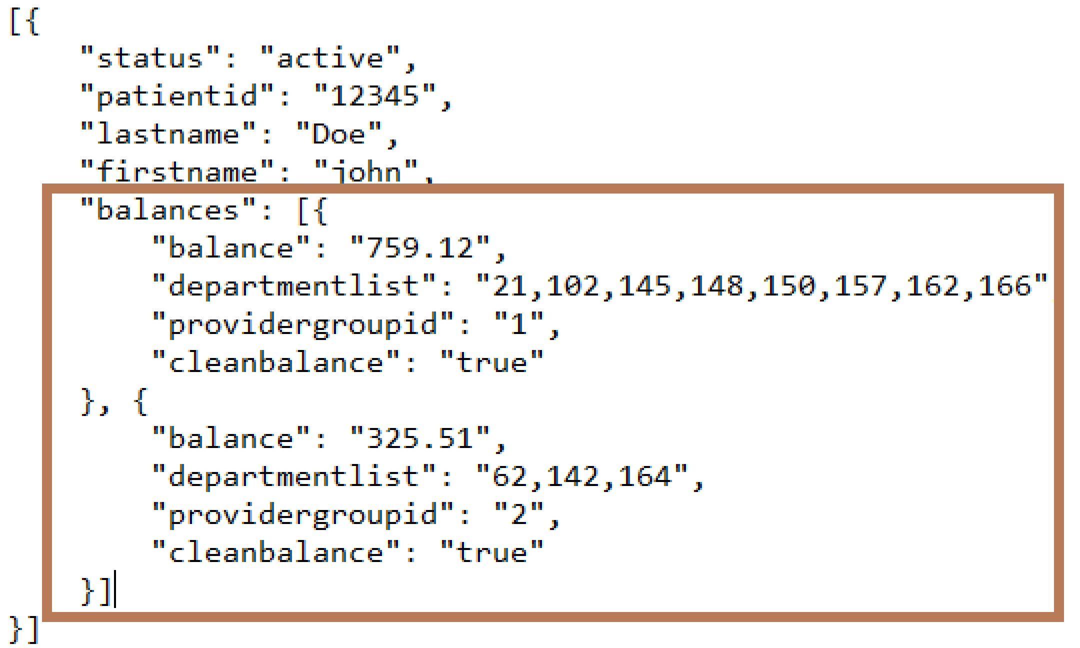
- Ask/Check for last_updated
- Be aware
  - Watch for complex responses

```
{
 "patient":12345,
 "first_name":"John",
 "last_name":"Doe",
 "insurances":[{***},{***}....],
 "claims":[{***},{***}....],
 "last_updated":"2019-02-21T13:28:06.419Z"
}
```

  - Need to test a lot

Dealing with Volume: Use Case 2 (cont.)

- Can we reduce workload further?
  - Check for API filter parameters (active, start and end date, etc)
    - `https://api.provider.com/v1/patients/?active=true&balance=true`
    - Do not see one? Log an enhancement request
  - Do not store if you do not need it



```
[{
    "status": "active",
    "patientid": "12345",
    "lastname": "Doe",
    "firstname": "john",
    "balances": [{
        "balance": "759.12",
        "departmentlist": "21,102,145,148,150,157,162,166"
        "providergroupid": "1",
        "cleanbalance": "true"
    }, {
        "balance": "325.51",
        "departmentlist": "62,142,164",
        "providergroupid": "2",
        "cleanbalance": "true"
    }]
}]
```

Dealing with Volume: Use Case 3

- The Timer must not pop up on certain pages.
- Original solution:
  - Repository-based system
  - Business rules in the database
  - Many round trips
  - Increased traffic and workload (and $$$)
  - Poor user experience

Dealing with Volume: Use Case 3 (cont.)

- Optimized solution:
  - Still a repository-based system
  - Read the settings and delegate some processing to the client box
  - 50x fewer round trips
  - Reduced traffic and workload (and $)
  - “Your system works much faster.”

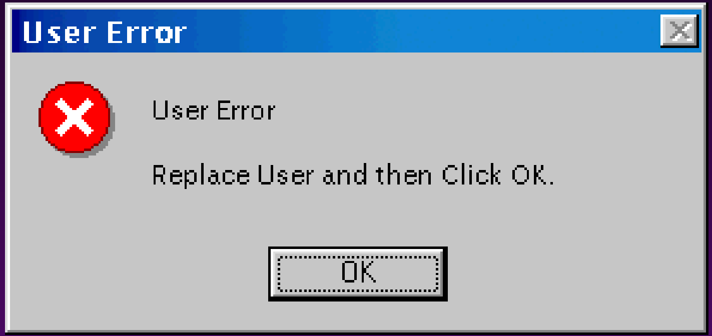
Dealing with Volume: Use Case 4

- How critical is it to keep data synchronized?
  - Update data overnight
  - Update on demand
  - Update while object is still an object of interest
- Immediate updates vs. data warehouse
  - Important changes – demog, insurance, etc.
  - Can it wait? – Staff performance reports
- Daily/weekly/weekend updates

Dealing with Volume: Use Case 4 (cont.)

- Exclude inactive clients
  - Retention period
  - Regulations
- Delete old data
  - Watch for SQL execution plans
  - Rebuild indexes
- Ask for API “GET /changed”
  - Less workload for API provider
  - Less workload for you

Handling Errors (1)


User Error

User Error

Replace User and then Click OK.

- Many points of failure in the architecture
- Network errors vs. API errors
- Some API errors are OK:
  - `GET /patient/9999`
  - `200 OK {“error”: “No patient found”}`
  - `404 Not Found {“error”: “No patient found”}`
- Be aware
  - Read the documentation – one error code, multiple meanings
  - API provider’s default error, i.e. “404 Not Found ”

Handling Errors (2)

- Why is there a 504 Gateway_Timeout?
  - Did you ask for the entire data set?
  - Narrow your search results
  - Break down to X number of calls
  - Find the timeout cutoff

Handling Errors (3)

- Log the error and repeat

```
request_attempt_nr := 0;
request_success := FALSE;
WHILE request_attempt_nr <= max_request_attempt_nr
      AND request_success = FALSE
LOOP
   BEGIN
     <API request>
     request_success := TRUE;
   EXCEPTION
      WHEN OTHERS THEN
           <log error>
           IF begin_request_attempt_nr = max_request_attempt_nr THEN
              raise;
           END IF;
   END;
   request_attempt_nr := request_attempt_nr + 1;
END LOOP;
```

Testing (1)

- Do not trust the API provider
  - Everyone makes mistakes.
- Look for:
  - API changes (without prior notice!)
    - "…It was a quick emergency fix for a specific client"
  - Data structure changes
  - Domain/Lookup changes
- Implement automatic testing
- Determine the official maintenance window

Testing (2)

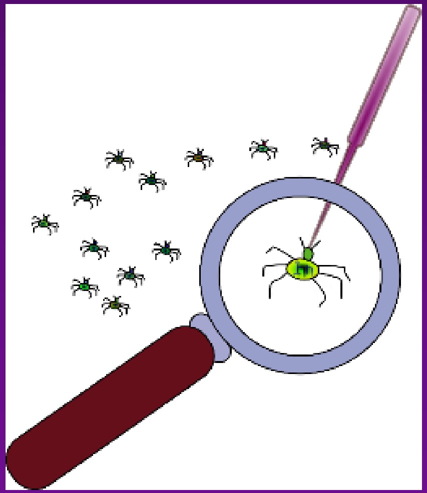
- Prepare “perfect” set
  - What is your “happy day scenario”
- Check API responses against the “perfect” set
  - Watch for response timestamp attribute
- Look for new/removed data elements
  - Find delta
- Ask for metadata/configuration API
  - https://api.provider.com/v1/chart/configuration/socialhistory

Testing (3)

- Ask for the release notes
  - … hopefully, PRIOR to the release!
- What if there are no API changes?
  - You must test - no matter what!
  - API developers and system developers – two different teams
- Watch for
  - Performance
  - Missing data

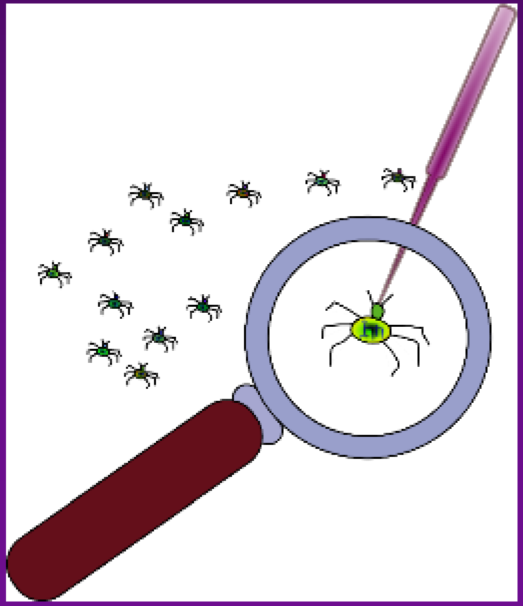
Debugging

- Get your own “playground”
- Keep your own log (both requests and responses)
- Add new parameter: your_request_id=1234
  - `https://api.provider.com/v1/patients/?active=true&request_id=1234`
- Report bugs
  - Replicate in TEST and PROD
  - API provider will be happy.
- Provide use cases to support your requests!



Coordinating

- Suggest enhancements
  - Engage your clients
  - Calculate ROI
- HIPAA-compliance
  - Watch for logs and screenshots



Backup plan

- Show must go on
  - Users do not care if APIs are down (they do not even know what API is)
- Check if a backup endpoints are available
  - Automatic and manual switch
- Allow manual entry with appropriate logging
- Report the issue with the Highest priority ASAP
  - API provider <u>must</u> know your users work 24x7

Health Checks

- Total API calls
- Make sure you know the cutoff timestamp
- API calls per second
- API calls per “client”
- Know your daily and per-second limits
- Group by response code
- Find anomalies

Summary

Why am I taking this course?
What am I supposed to do with all of this information?
How can I prove I know it?

- An efficient API-based system involves:
  - 1. Communication
    - … because when lots of people are involved – something will be “lost in translation”
  - 2. Being reasonably paranoid
    - … because everything that CAN change at some point MAY change
  - 3. Keeping ALL records
    - … because before blaming somebody else you should have proof!
  - 4. Holding your ground
    - … because everybody should be “not guilty” until proven otherwise

Contact Information

- Michael Rosenblum – mrosenblum@dulcian.com
- Grigoriy Novikov - gnovikov@dulcian.com
- Dulcian, Inc. website - www.dulcian.com