Top Performance Bottleneck Pattern Deep Dive

Andreas Grabner  - @grabnerandi
andreas.grabner@dynatrace.com
Why Performance?
- cars.com: 700 deployments / YEAR
- flickr: 10 + deployments / DAY
- Etsy: 50 – 60 deployments / DAY
- Amazon: Every 11.6 SECONDS
“We increased from monthly to 250+ deployments per week“
Not only *fast delivered* but also *delivering fast*!

<table>
<thead>
<tr>
<th>Response Time</th>
<th>Conversions</th>
</tr>
</thead>
<tbody>
<tr>
<td>↓ +100ms</td>
<td>↓ -1%</td>
</tr>
<tr>
<td>↑ -1000ms</td>
<td>↑ +2%</td>
</tr>
<tr>
<td>↑ -1000ms</td>
<td>↑ +10%</td>
</tr>
</tbody>
</table>

*Amazon.com*, *Walmart*, *STAPLES*
User Experience == Load Time + Error Rate + Bandwidth

#1: Which Geo has which “User Experience”?

#2: Who are these users?
New Deployment + Mkt Push

Overall increase of Users!

Increase # of unhappy users!

Spikes in FRUSTRATED Users!

Decline in Conversion Rate
Performance == Response Time | UX + Resource Usage

CPU & Mem Impact of updated Dependency Injection Lib
Performance == Scalability! What's the Cost per User?

App with Regular Load supported by 10 Containers

Twice the Load but 48 (=4.8x!) Containers! App doesn’t scale!!
Performance --> User Behavior: Click Heatmap Analysis
How to analyze perf?
**Time:** Wall Clock, CPU, I/O, Wait/Sync, Susp, Page Load

**Throughput:** # of Requests per Timeinterval

**Interactions:** # SQLs, # Messages, # Services, # Images, # CSS

**Resources:** CPU Cycles, Memory, I/O, Log Messages, ...

**Pools and Queues:** Sizes, Utilization, Acquisition Time, # Publishers vs # Subscribers, Process Time

**Errors:** Exceptions, HTTPs, TCP Packet Loss
Available Tools

Mission Control  
VisualVM  
New Relic  
JProbe  
YourKit  
AppDynamics  
JProfiler  
Dynatrace  
Solaris Studio  
NetBeans Profiler  
Honest Profiler  
XRebel

AND MANY MORE
THERE'S THE DOOR
„In Your Face“ Data!!

https://dynatrace.github.io/ufo/
Where do your Stories come from?
Method Level Hotspots

Arch Validation

+ Exceptions, Logs, Memory Allocation, Threads, Actual Code ...
Share PurePath Analysis

Andreas Grabner – agrabner@dynatrace.com
March 2015

Highlights
- Too many SQL Statements
  - Tip: Use Database Aggregation!
- Too many Exceptions
  - Tip: Use Exception Aggregation!
- Web Server has 690! Active threads – App Server has 240!
  Threads
  - Most of the wait time on Web Server probably due to waiting on the next free available thread on the AppServer
  - Scale your Web and App Server Infrastructure
  - Optimize App Server Code to handle more requests per minute

Threading Issues

"Only" 240 Active threads were
based on the Thread Count Measure.
Most of the wait time on the Web
Server could therefore be eliminated
because it needs to wait for the next
free thread on the App Server.
Increase Worker Threads on App Server, Add More App

Hidden Exceptions with Overhead

Runtime Exceptions with huge
Overhead that we didn’t capture
with Exception Server but with
AutoSensors

Learn more about PurePath Analysis by checking out this link:
http://www.dynatrace.com/2014/12/04/dynatrace-performance-tips
#taking-off-the-retire Weber
Frontend Performance

„We are getting FATer!“
m.pepsi.com during Super Bowl

**Find all Key Performance Indicators (KPI) for the selected page**

These values help you compare with other versions of the same page to identify problems or regressions. Learn more on Key Performance Indicators and how they get calculated.

<table>
<thead>
<tr>
<th>First Request</th>
<th>6ms</th>
<th>On Server</th>
<th>5859ms</th>
<th>DNS</th>
<th>0ms</th>
<th>Network</th>
<th>16307ms</th>
<th>Total Size</th>
<th>20814kb</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Impression Time</td>
<td>277ms</td>
<td>On Client</td>
<td>1490ms</td>
<td>Connect</td>
<td>0ms</td>
<td>JavaScript</td>
<td>3268ms</td>
<td># of Requests</td>
<td>437</td>
</tr>
<tr>
<td>OnLoad Time</td>
<td>2018ms</td>
<td>Ø Interactive</td>
<td>152ms</td>
<td>Transfer</td>
<td>3341ms</td>
<td>Rendering</td>
<td>3402ms</td>
<td># of XHR</td>
<td>2</td>
</tr>
<tr>
<td>Total Load Time</td>
<td>15065ms</td>
<td>Ø Wait</td>
<td>1973ms</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Total size of ~ 20MB**

Following table lists details about all 14 different mime-types and their impact on caching:

Analyze which types of resources are cached vs. not cached and how much size you can save by caching more of these resources.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TOTAL</td>
<td>434</td>
<td>21314149</td>
<td>398</td>
<td>21250483</td>
<td>-</td>
<td>238916</td>
<td>36</td>
<td>63666</td>
</tr>
<tr>
<td>image/jpeg</td>
<td>2</td>
<td>18387943</td>
<td>230</td>
<td>18387943</td>
<td>1</td>
<td>47789</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>image/png</td>
<td>4</td>
<td>424580</td>
<td>75</td>
<td>424580</td>
<td>0</td>
<td>908</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>image/gif</td>
<td>1</td>
<td>5762</td>
<td>50</td>
<td>5762</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Tip: Make F12 your friend!

433! Requests & 14.5MB Page Size

Lots of HTTP 403s
Tip: Monitor your Real End Users

#1: Which Geo has which “User Experience”?

#2: Who are these users?
Key Metrics

# of Resources
Size of Resources
Total Size of Content
HTTP 3xx, 4xx, 5xx
# of Domains
Backend Performance

“The Usual Suspects”
What are the typical root causes you most often experience? 

Figure 1.16

54.8% Slow database queries

28.4% Concurrency issues

18.5% Slow DB

27.6% Memory leak

13.3% Excessive disk IO

12.0% Slow/unreliable third party entities

12.9% Excessive network IO

17.9% GC pauses

23.4% Configuration issues*

10.6% Excessive memory churn

7.6% HTTP session bloat

1.7% Other

4.4% Don't know

*Answers were multiple choice, so the numbers don't add up to 100%. Deal with it ;)

cite the database as the most common challenge or issue with application performance.
#1: Watch out for Slow Single Statement

**Slow Single SQL Issue:** The root cause can be single SQL queries that block connections for a long time.

<table>
<thead>
<tr>
<th>SQL</th>
<th>Executions</th>
<th>Exec Avg [ms]</th>
<th>Exec Max [ms]</th>
<th>Exec Total [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPDATE <code>Fرص</code> LINE SET S.LINE = <code> فرص</code></td>
<td>2</td>
<td>982024.10</td>
<td>1048440.34</td>
<td>1964048.21</td>
</tr>
<tr>
<td>UPDATE <code>Fرص</code> LINE X SET X.QTY = 0 WHER</td>
<td>1</td>
<td>960979.07</td>
<td>960979.07</td>
<td>960979.07</td>
</tr>
<tr>
<td>UPDATE <code>Fرص</code> LINE X SET X.QTY = 0 WHER</td>
<td>19</td>
<td>700752.90</td>
<td>743943.66</td>
<td>13314305.07</td>
</tr>
<tr>
<td>UPDATE <code>Fرص</code> LINE X SET X.QTY = 0 WHER</td>
<td>24</td>
<td>214678.81</td>
<td>308914.09</td>
<td>5152291.53</td>
</tr>
</tbody>
</table>
Tip: Analyze Execution Plans

Focus on: Rows Processed, Bytes Read, Impact of Indices, ...
Tip: Analyze other Database Activity

Tip: Watch out for CPU, Disk, Rows Processed, Execution Time, Locks, …
#2: Too many Queries – Too much Data

- **Excessive SQL: 24889!** Calls to JDBC API

- **Database Heavy: 66.51%** (40.27s) Time Spent in SQL Execs
#3: N+1 Query Access Pattern

**Inefficient Pool Access:**

12444! individual connections

**N+1 Query Problem:** Same SQL with different bind value

Individual SQL are fast <1ms

No need to optimize on the database side
#3: N+1 Query Access Pattern (cont.)

**N+1 Query Problem + Excessive SQL:** Lazy Loading in Hibernate Executes 4k+ Statements

**Database Heavy: 2 SQL Queries executed 4k+ times totaling to 6s**

<table>
<thead>
<tr>
<th>SQL</th>
<th>Execs/calling</th>
<th>Executions</th>
<th>Preparations</th>
<th>Exec Avg [ms]</th>
<th>Exec Total [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>select history0.trialId as trialId</td>
<td>2178.00</td>
<td>2178</td>
<td>2178</td>
<td>1.31</td>
<td>2851.90</td>
</tr>
<tr>
<td>select events0.trialId as trialId</td>
<td>2178.00</td>
<td>2178</td>
<td>2178</td>
<td>1.48</td>
<td>3219.95</td>
</tr>
<tr>
<td>select trial0.id as id</td>
<td>13.00</td>
<td>13</td>
<td>0</td>
<td>2.74</td>
<td>35.57</td>
</tr>
<tr>
<td>select company0.id as companyId</td>
<td>11.00</td>
<td>11</td>
<td>11</td>
<td>2.70</td>
<td>29.74</td>
</tr>
<tr>
<td>select this.id as id</td>
<td>1.00</td>
<td>1</td>
<td>1</td>
<td>4.05</td>
<td>4.05</td>
</tr>
<tr>
<td>SELECT DISTINCT LOWER(u.user_name) as user_name, u.display_name</td>
<td>1.00</td>
<td>1</td>
<td>1</td>
<td>1.75</td>
<td>1.75</td>
</tr>
</tbody>
</table>
**#4: Prepared vs Unprepared**

**Candidate for Prepared Execution:** Watch out for a single SQL that gets executed more than once per request.

**Unprepared Statements:** Only 7 out of 4904 total executions across multiple requests got prepared!

### SQL Table

<table>
<thead>
<tr>
<th>SQL</th>
<th>Execs/calling Trans</th>
<th>Executions</th>
<th>Preparations</th>
<th>Exec Avg [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT [Version] AS COL1 FROM [CanvasPage3] WHERE [CanvasPage3] = @p0</td>
<td>41.50</td>
<td>83</td>
<td>4</td>
<td>2.04</td>
</tr>
<tr>
<td>SELECT b.[CanvasPage4] AS COL1, c.[CanvasPage5] AS COL2, c.[vca_class] AS COL3, c.</td>
<td>33.44</td>
<td>301</td>
<td>10</td>
<td>1.80</td>
</tr>
<tr>
<td>Security</td>
<td>8.22</td>
<td>74</td>
<td>74</td>
<td>72.24</td>
</tr>
<tr>
<td>SELECT a.[ResourceID] AS COL1, a.[Bucket] AS COL2, a.[mCountry] AS COL3, a.[Description] AS CO</td>
<td>7.00</td>
<td>35</td>
<td>14</td>
<td>1.12</td>
</tr>
<tr>
<td>if exists (select * from CanvasUser where CanvasId = @p0 and UserGuid = @p1) select 1 else select 0</td>
<td>4.85</td>
<td>262</td>
<td>6</td>
<td>1.57</td>
</tr>
<tr>
<td>SELECT TOP(@_TAKE) a.[MediaId] AS COL1, a.[Active] AS COL2, a.[AlbumId] AS COL3, a.[AlbumMediaType] AS</td>
<td>3.83</td>
<td>88</td>
<td>3</td>
<td>1.55</td>
</tr>
<tr>
<td>SELECT [Tag] AS COL1, [AlbumId] AS x1l, [seq] AS x12 FROM [AlbumTags] WHERE [AlbumId] = @p0 ORDER BY x</td>
<td>3.83</td>
<td>88</td>
<td>3</td>
<td>1.36</td>
</tr>
<tr>
<td>SELECT [val] AS COL1, [MediaId] AS xjL, [seq] AS xj2 FROM [AlbumMediaTags] WHERE [MediaId] = @p0 ORDER</td>
<td>3.83</td>
<td>88</td>
<td>3</td>
<td>1.33</td>
</tr>
</tbody>
</table>
**Tip: Impact on Execution Plan Generation**

*CPU intensive operation:* A new execution plan is generated for each unique SQL statements.

```sql
SELECT FirstName, LastName, Country
FROM Customer WHERE CustomerId = 8
```
Tip: Oracle v$ metrics to monitor overhead of parsing

High CPU caused by execution plan generation

Time Spent Breakdown show up to 25% of the time spent on Parsing
Tip: „Ad hoc“ vs Prepared Statements

Without bind variable:

```java
int customer_id;
Statement command = connection.createStatement("select FirstName, LastName, Country " + 
"from Customer where CustomerId = " + customer_id);
```

The database will generate a new execution plan for every value of customer_id

Using bind variable:

```java
int customer_id;
PreparedStatement command = connection.prepareStatement("select FirstName, LastName, Country " + 
"from Customer where CustomerId = ?");
command.setInt(1, customer_id);
```

The database will generate one single execution plan and reuse it for every value of customer_id
Proof: No Parsing Overhead!

Low CPU usage

Parsing doesn’t show up anymore on the Time Spent Breakdown
Proof: Significant Performance Improvement

Response Time with Ad-Hoc Queries per Transactions: **177.9 ms**

Using Bind Variables, Response Time reduced by **56% (78.64 ms)**
Tip: Analyze by SQL Type (SELECT, INSERT, UPDATE, ...)  

Which Types take how long?  

When do you have spikes?
Tip: Does our Data Caching work?

Do we see increased in **AVG # of SQL** Executions over Time when we have constant load?

Do **TOTAL # of SQL** Executions increase with load? Shouldn’t it flatten due to **CACHES**?
Lessons Learned – *Don’t Assume* …

• ... you know what code is doing you inherited!!
• ... you are not making mistakes like this 😊

• Explore the Right Tools
  • Built-In Database Analysis Tools
  • “Logging” options of Frameworks such as Hibernate, ...
  • JMX, Perf Counters, ... of your Application Servers
  • Performance Tracing Tools: Dynatrace, Ruxit, NewRelic, AppDynamics, Your Profiler of Choice ...
Key Metrics

# of SQL Calls
# of same SQL Execs (1+N)
# of Connections
Rows/Data Transferred
Pools & Queues
„Proper Sizing“
„Proper Usage“
#1: Pick Proper Pool Sizes

Do we have enough *DB CONNECTIONS* per pool?
#2: Monitoring Pool Usage and Impact

- **Connection Pool Usage**
  - Shows the saturation of this pool.

- **Connection Acquisition**
  - High acquisition times and multiple waiting threads are an indicator for a saturated pool or improper pool configuration (for example, low time to live).

- **Wait Time**
  - Indicates how long the threads have to wait for a new connection.

- **Thread Wait Count**
  - Shows the number of threads waiting for a new connection.

- **Connection Usage Time**
  - High usage times may be an indicator for leaking connections or long-running jobs.

- **Prepared Statement Cache Discard Rate**
  - Values above 0 indicate a sizing issue of the prepared statement cache or improper use of prepared statements.

- **Create and Close Count**
  - High create and close counts can indicate an improper pool configuration (for example, time to live).

- **How long to wait for a new connection?**
- **How long connections are in use?**
#3: Correct Load Balancing

- Increasing the maxThread number caused individual JVMs to overload and crash.
- After changing the load balancing strategy load was distributed evenly without any crashes.
#4: Excessive Thread Usage per Request

**Excessive Remoting:**
40! RMI Calls on 20! parallel threads

**Worker Thread Exhaustion:**
Excessive use of threads in App Server also causes backup of requests in Web Server. Watch your Worker Thread Pools!

**Excessive SQL:**
2790! SQL Calls Total
Tip: Follow Threads through PurePath

Async/Sync Nodes: RMI calls done on background threads

Thread Name: Count Unique Thread IDs
**Tip: Watch out for Sync / Wait**

### Method Breakdown by Wait Time

<table>
<thead>
<tr>
<th>Method</th>
<th>Wait Sum</th>
<th>Breakdown</th>
<th>Class</th>
<th>APIs</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wait(long)</code></td>
<td>1.63s</td>
<td>99.0%</td>
<td><code>java.lang.Object</code></td>
<td>P11a</td>
</tr>
<tr>
<td><code>park(boolean, long)</code></td>
<td>113ms</td>
<td>99.0%</td>
<td><code>sun.misc.Unsafe</code></td>
<td>P11a</td>
</tr>
</tbody>
</table>

*1.63s in `Object.wait`*  
Means that this thread is put to hold

### Caller Breakdown of `wait(long)`

<table>
<thead>
<tr>
<th>Method</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Object.wait()</code></td>
<td></td>
</tr>
<tr>
<td><code>Job&lt;ComplianceAuditService&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>GeneratedMethodAccessor$2.invoke(Object, Object)</code></td>
<td></td>
</tr>
<tr>
<td><code>MethodProxy.invoke(Object, Object)</code></td>
<td></td>
</tr>
<tr>
<td><code>Object.invoke(Object, Object)</code></td>
<td></td>
</tr>
<tr>
<td><code>CheckInvocation.invoke(Object, Object)</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$1.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$2.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$3.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$4.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$5.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$6.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$7.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$8.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$9.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$10.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$11.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$12.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$13.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$14.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$15.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$16.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$17.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$18.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$19.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$20.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$21.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$22.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$23.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$24.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$25.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$26.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$27.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$28.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$29.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$30.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$31.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$32.run()</code></td>
<td></td>
</tr>
<tr>
<td><code>ConnectorServer$SingleThreadExecutor$Task$33.run()</code></td>
<td></td>
</tr>
</tbody>
</table>

*Waiting on the next Connection to become available!*
Key Metrics

Pool and Queue Sizes
Time in Sync & Wait
# of Threads per Request
Balanced Requests
When Logging 
„Impacts Performance“

WE CAN LOG THIS!
#1: Log Hotspots in Frameworks!

callAppenders clear CPU and I/O Hotspot

Excessive logging through Spring Framework
#2: Debug Log and outdated log4j library

#1: Top Problem: log4j.callAppenders
   -> 71% Sync Time

#3: Doing “DEBUG” log output: Is this necessary?

#2: Most of logging done from fillDetail method
#3: Overhead caused by Exceptions

fillInStackTrace is Top 2 in CPU Hotspots
All these Exceptions that never show up in a log file are consuming all CPU
Tip: Too Many Exceptions vs Log Messages

Up to 20000 Custom Exceptions
That’s about 4000x the number of Exceptions per Log Message

2-5 Log Messages per 5 Min
Looking at the important (SEVERE, FATAL, ...) log messages written
Key Metrics

# of Log Entries
Size of Logs per Use Case
Ratio Exceptions to Logs
“(Micro-)Service Migration Mistakes”
Online Sports Club *Search Service*

1) Started as a small project
2) Slowly growing user base
3) Expanding to new markets – 1st performance degradation!
4) Adding more markets – performance becomes a business impact

4) Potentially start loosing users

Response Time

Users

20xx  |  2014  |  2015  |  2016+
Early 2015: Monolithic App

2.68s Load Time

Can’t scale vertically endlessly!

94.09% CPU Bound
Proposal: Service approach!
Testing the Backend Service alone scales well ...

7:00 a.m.
Low Load and Service running on minimum redundancy

12:00 p.m.
Scaled up service during peak load with failover of problematic node

7:00 p.m.
Scaled down again to lower load and move to different geo location
GO LIVE DAY – 7:00 a.m.

User experience index: 0.81
Response time: 2.82 s
User actions: 1.64 /min
Failure rate: 1.61 %

User satisfaction:
- Satisfied: 11.6k
- Tolerating: 6.33k
- Frustrated: 1.11k

Bounce rate: 20.1%
Conversion rate: 2%
GO LIVE DAY – 12:00 p.m.
What Went Wrong?
Single search query end-to-end

Architecture Violation
Direct access to DB from frontend service

26.7s Load Time
5kB Payload

33! Service Calls
99kB - 3kB for each call!

171! Total SQL Count
The fixed end-to-end use case
“Re-architect” vs. “Migrate” to Service-Oriented Architecture

- **2.5s** (vs 26.7) **5kB** Payload
- **1!** (vs 33!) **1kb** Service Call
- **3!** (vs 177) **3kB** Payload!
Key Metrics

# of Service Calls
Payload of Service Calls
# of Involved Threads
1+N Service Call Pattern!
You *measure* it! from Dev (to) Ops
## Metrics from and for Dev(to)Ops

### Scenario: Monolithic App with 2 Key Features

<table>
<thead>
<tr>
<th>Use Case Tests and Monitors</th>
<th>Service &amp; App Metrics</th>
<th>Ops</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Build #</strong></td>
<td><strong>Use Case</strong></td>
<td><strong>Stat</strong></td>
</tr>
<tr>
<td>Build 17</td>
<td>testNewsAlert</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td>testSearch</td>
<td>OK</td>
</tr>
<tr>
<td>Build 25</td>
<td>testNewsAlert</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td>testSearch</td>
<td>OK</td>
</tr>
<tr>
<td>Build 26</td>
<td>testNewsAlert</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td>testSearch</td>
<td>OK</td>
</tr>
<tr>
<td>Build 35</td>
<td>testNewsAlert</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>testSearch</td>
<td>OK</td>
</tr>
</tbody>
</table>
„In Your Face Data“

https://dynatrace.github.io/ufo/
Questions

Slides: slideshare.net/grabnerandi
Get Tools: bit.ly/dtpersonal
YouTube Tutorials: bit.ly/dttutorials
PurePerformance Podcast: bit.ly/purepeef
Contact Me: agrabnererand@gmail.com
Follow Me: @grabnerandi
Read More: blog.dynatrace.com
Andreas Grabner
Dynatrace Developer Advocate
@grabnerandi
http://blog.dynatrace.com