Chris Farnsworth & Sean Klink

Software Architecture

Assignment 4

Deploying and Monitoring PieMatrix as Microservices

**Introduction:**

This document introduces a strategy for deploying and monitoring a microservices architecture previously converted from a monolithic architecture: PieMatrix [1].

We split up similar or related microservices categorically based on how dangerous it is for the service(s) to be offline for any given amount of time. With security being paramount, the descending list of microservices categories (1 being most critical for uptime) is: 1.) Security, 2.) Core Data, 3.) Database Manager, 4.) Formatters for Frontend, and 5.) Auxillary Tools. Security is taken very seriously and nobody wants compromised data. The core data/messaging is the second most uptime critical component tightly coupled with the database manager and formatters for UI (User Interface). Auxillary tools may be absent relatively temporarily.

Within each numbered severity category are severity subcategories delineated by lowercase English letter a – z (as needed).

**Path to deployment:**

Employee Frederick finds a bug/issue worthy of revision. Frederick fixes the problem and the revision gets queued up in a QA (Quality Assurance) inbox. QA “black box” tests the revision and, if acceptable: deployment.

The above is the general approach for bugs. However, all changes to UI elements that are “pre-baked” (not custom) can be submitted, forgoing the QA process – it is assumed that the employee accepts full responsibility for their quality.

Additionally, major new features must pass an additional approval step – the customer must receive them as a beta release, following QA.

Automated testing is optional depending on the service. Data-critical services MUST be tested thoroughly. The submitting developer is responsible for writing the test that checks the sanity of data they receive/distribute.

**Monitoring:**

Monitoring: Server integrity/overall health, incoming and outgoing requests, maximum capacity for requests in current state, cost/risk analysis, scalability.

**Short term:**

Server integrity/overall health must be monitored, as well as our current usage vs how much throughput we can support. AWS will report this information for us. We will need to define what “healthy” means on a service-by-service basis, but a reasonable starting cap might be 50k requests per hour on each service. Any more, and we might begin to suspect an attack.

**Long term:**

Cost/risk analysis, scalability issues.

As we scale, we need to ensure we aren’t spending too much on any one service. We need to examine the trends surrounding each service – at what rate are we growing, and adjust instances to match demand. Additionally, once we observe the fastest-growing services, we can prioritize optimization. We should aim to reduce throughput as much as possible on the highest-demand services, while adjusting scaling rules to meet demand as it comes in the meantime. This can be configured via AWS.

**Errors/Warnings at Compile Time:**

Only senior-level staff cleared to make the decision on a persistent bug/problem will either clear a warning/error as erroneous or will declare it necessary to fix the error/warning. In general, warnings are not tolerated. An error or warning should only be cleared if we can confidently verify it’s a compiler issue with no reasonable way to get around. We should broach all such issues with the manufacturer of our compiler. The end goal is to have compilation return error and warning-free.

**General rules about updating:**

Upgrading any library or service will occur in real-time, without taking the system down. However, we will strive to upgrade services on comparatively “off” hours. (For instance, Sunday mornings at 3am EST). This will hopefully minimize interruptions to the service or library used.

**Third-party libraries:**

When upgrading versions of third party libraries or otherwise swapping a used third party library out for something else, a local copy of the library shall be stored in Datomic to be retrieved and incorporated locally if the original copy is taken down. Every time an external third party library is used, a local copy is generated and stored in Datomic.

The core issue with third-party libraries we must pursue is that they introduce incompatibilities. To pursue an upgrade at all, we must run quality assurance against all affected parts of the system when a third-party library is updated. We maintain a list of all services affected, directly or indirectly, by every third-party library we use.

The physical upgrade is then handled by directing all existing traffic to the “old” copy of a given library while we install new versions. Traffic will not be rerouted until all required library updates are complete – this is because libraries may rely on other libraries. Therefore, the new installs will not accept traffic until all components are in place.

**Security:**

**Automatic Startup**

We will rely on industry-standard software to manage startup passwords on all services. This software will handle encrypting the passwords locally and initializing them at runtime.

**Web Security**

The server(s) we maintain have two layers of differently optimized versions of Snort, an IDS (Intrusion Detection System)—one outside the router interfacing with the Internet, and one just inside the network.

We also use AWS (Amazon Web Services) for the abstractly physical locale of all the individual instances of our microservices being spun up in response to demand. Therefore, AWS is responsible for most of our system’s security.

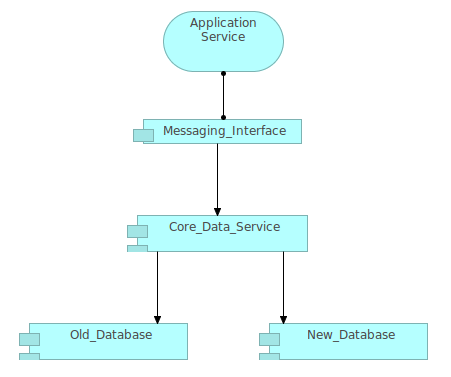
**Updating PieMatrix Services Intro:**

Services on our system are generally designed in a chain. At the very bottom is a database manager that interfaces with all necessary databases directly. At the top is a message acceptance service that receives and distributes a request (messages the services that return the specific XML it needs to return). We describe how to handle this upgrade for the different types of services at varying layers below. We do not describe all services individually, as the procedure for several would be redundant.

**Core Data:**

The core data and the messaging of this information is second to security concerns. The way in which we handle this scenario is to, first, update this entire package of services so that the collective can interface and communicate with any arbitrary database, not just its currently assigned one. Next, we create a new database, redirecting to the old database until the update completes. Finally, route new traffic to the updated service.

This handles any particular case. However, we must work up and down the chain, as it were. Consider the diagram below:



In general we take this approach with core data services: First, update the Core\_Data\_Service to address multiple instances of a database if it does not yet support multiple RESTful links already. Then, update the Database Managers in place, temporarily linking to the old copy of the database manager while the new one is updated. Finally, redirect all new traffic to the new database manager. Delete the old copy. Now, work up the chain and repeat this process.

1) Update the service linking to this service to address multiple databases

2) Update the underlying service, temporarily linking to the old copy of the service.

3) Delete the old service once all activity on it has ceased (use AWS’s monitoring tools to report when this condition is met).

These are examples of our services that touch the core data at a database level directly or indirectly and must thus be updated in this manner:

1. **Roles**
2. **Scanner**
3. **Version Verification**
4. **Project Data**
5. **Critical Path Management**
6. **User**
7. **Scheduler**

**Database Managers:**

The database managers will be swapped out following the procedure explained above. Note that in order to swap these services out, we must first update the services that address them to link to an old copy. Follow this general protocol for updating these:

1) Continue to allow all ongoing traffic to route through the existing database.

2) Insert the new database managers in place, with a new resource link.

3) When update completes, route all database traffic through the new address.

4) Delete all links to the old database once AWS reports all traffic to the old address has ceased.

**Formatters for Frontend:**

This package of microservices may be hot-swapped with lag time not to exceed 500ms. User, in this case, may only experience a slight jitter or automatic refresh depending on browser.

Please note that this category of microservice is technically part of the “chain” we discussed in the Core Data, and as such, it is part of the strategy for service upgrade explained there. However, it’s at the top of the “chain”, and therefore the least tricky to upgrade. It merely needs the ability to redirect to multiple “versions” of the message passers to the core data services – so that while upgrades are occurring,

However, note that while these are being replaced, we do need to temporarily redirect our users to a static set of pre-baked frontend data. We should cache a generic response for each user anyway – so, should a user access the frontend service while we’re migrating, merely direct them to their cached response, and update once the frontend formatter is back online.

These are the front-end pages that should be addressed in this manner:

1. **Social Feed View**
2. **Process Authoring**
3. **Process Execution**
4. **Message App**

**Auxillary Tools:**

This package of microservices (mainly by design of outgoing and incoming queues for various messaging) may be cold swapped with lag time not to exceed 1.5s. The user may experience some jitter or page reloading depending on tool being updated and what the user was doing at that particular moment.

These services, by their nature, do not touch other services – they merely receive a request with input and provide an output. If a user tries to access them while they are offline, the user should merely be informed that their request has been queued, but that they will have to wait for completion. This should be no more than inconvenient, but not dangerous.

a.) **File Encryptor/Decryptor**

b.) **Send Manual Notification**

c.) **Send Auto Notification**

d.) **Notification Sender**

e.) **File Uploader/Downloader**

f.) **File Database Updater**

g.) **File Database Accessor**

h.) **Search within File (if text file)**

i.) **Convert to Microsoft Project**

j.) **Decode from Microsoft Project**

k.) **Sync with Microsoft Project**

**Conclusion:**

With a team of 12 individuals, we can manage to update quickly and efficiently, monitoring largely being taken care for us by AWS. When an alert is issued by AWS, the team finds the bug/problem, fixes it, QAs it, and subsequently deploys it using either the no-lag technique described for the core data section, hot-swapping, or cold-swapping—each with its own lag-range. These techniques are not a hard-and-fast rule, but rather can be interchangeably used for the update at hand. For example, if a formatter for the frontend needs to be significantly overhauled, forcing refresh on many thousands of users may cost profit in the long-run. Therefore, we may use the technique described in core data with modification: we allow currently connected users the chance to stay on the original page until they terminate their session. New connections automatically go to the new page until the last of the old-pagers have terminated their sessions.

**References:**

1. https://www.piematrix.com/