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.

**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.

**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, incoming and outgoing requests, maximum capacity for requests in current state.

**Long term:**

Cost/risk analysis, scalability issues.

**Errors/Warnings:**

Only senior-level architects 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, bugs and warnings are not tolerated.

**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.

**Security:**

**Automatic Startup**

On automatic startup in the event of unexpected downtime, the AES encrypted password “IAMTHELORDTHYGOD” shall be placed much like mythical Bible code within a 15 GB file of gibberish. An additional algorithm is used to decompose and search the encrypted input for the correct, corresponding indices in the gibberish file. Once this phase is bypassed, the user then has access to all other AES encrypted passwords that are decrypted sequentially using traditional tools.

**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 interface 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.

**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.

**Database Manager:**

Database manager is third most critical and must be handled much the same way as the core data/messaging.

**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.

**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.

**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/