Chris Farnsworth & Sean Klink

Software Architecture

Assignment 2

Migrating PieMatrix to Microservices Strategy

**Introduction:**

This document outlines a strategy to shift PieMatrix’s monolithic application gradually to a network of microservices. The UI facing the user that is currently in place will remain largely the same – rather every component on the backend will change, one at a time, as possible.

Our strategy commences by converting one-off services like file uploading/downloading and their attached services to microservices. We begin with isolated, simple services like encryption to allow the development team to gain experience with this new style of thinking, before moving the remainder of services, organized by complexity/impact on the overall system.

The full list of services, in our proposed order of implementation, is highlighted below:

**Microservices:**

**Auxillary Tools**

**File Encryptor**

**File Decryptor**

Inputs: File to encrypt/decrypt

Outputs: processed file

Summary: This pair of services, very simply, accepts a file and transforms it by encryption or decryption.

**Send Manual Notification**

**-Severity assigner**

**-notification queue modifier**

Inputs: Assignation of severity of notification, e.g. ranges from 1 to 3, 1 being the most urgent, 2 being moderately urgent, and 3 being much less urgent.

Output: Notification placed in notification queue, propagating through the queue based on each notification’s severity level.

Summary: Sending a manual notification with a severity level triggers the notification queue modifier to order the notifications in order of severity.

**Send Auto Notification**

**-Get Current date**

**-Scan active projects for corresponding dates**

**-Severity assigner**

**-Notification queue modifier**

Inputs: Assignation of severity of notification corresponding to a specific inputted date, e.g. AutoSend(X, Date), where X ranges from 1 to 3, 1 being the most urgent, 2 being moderately urgent, and 3 being much less urgent.

Action: AutoSend() scans the current project for supplied date, assigning supplied severity level to each corresponding notification.

Output: Modified notifications within the notification queue propagate through the queue based on each modified notification’s severity level.

Summary: Sending an auto notification with a severity level and specific date triggers the notification queue modifier to reorder the notifications in order of the new severity for each modified notification.

**Notification Sender**

Input: Current notification queue.

Output: Notifications serially sent to respective destinations.

Summary: The notification sender will grab a high (1) severity notification from the notification queue and send it to the specified user until severity level 1 is depleted, then moving on to severity levels 2 and 3.

**File Uploader**

**-Version History by date**

**-Update file version**

Input: New version of a specific file.

Output: Updated and current versioned file server side.

Summary: User may upload file to the server to replace or modify current versioned file. History of modifications stored in datomic database residing on the server.

**File Downloader**

**-Version History by date**

**-Select specific file version**

Input: Query of datomic database by file and value (date and time).

Output: Specific versioned file.

Summary: User may query the datomic database residing server side to download a specific file as of a specific date and time. The cloud-based datomic interfaces with the server and the client. The user initiates an action which is XMLed to the server. The server then initiates datomic to send the modification to the client via XML. http://docs.datomic.com/

**File Database Updater**

Input: As query to datomic database, user must specify file (ENTITY), date and time (VALUE), and the specification of addition/retraction. Queries CAN be more complicated and sweeping, but for this use case we are limiting the queries to these three attributes for file updating.

**File Database Accessor**

See File Database Updater.

**Search within file (if text file)**

Summary: Once file has been moved from the cloud to the personal server(s), it can be (if text file) searched through via XML.

Use case: User inputs Ctrl-F which elicits input box client side. User inputs String or Char to be searched for in the file. XML ferries the query to the file server side and searches for the String or Char, highlighting and indexing the instances of each literal and then ferrying the highlighted and indexed file back to client side for review.

**Convert to Microsoft project**

**-Convert each XML element individually**

Summary: User may execute an option client side to have the file compatible with a Microsoft project. Server side, the XML is modified to be compatible.

**Decode from Microsoft project**

**-Decode each XML element individually**

Summary: Opposite process from Convert to Microsoft project.

**Sync with Microsoft project (do both)**

**Formatters for Frontend**

**Social Feed View**

**-Recent Project Summarizer**

Use Case: User may toggle I/O button labeled “Summary” on the penultimate position to the right on the “Options Bar”. This brings up a 33% dimensions panel in the upper right hand corner of the user’s screen. It lists the project name, the attached user’s and their hierarchy of privileges, and user provided summary description.

**-Recent Projects Attached to this user at a glance**

Hovering over the project name will initiate a pop-up window with all the same information

**-Convert project on the backend to Project view for the frontend**

First, deactivate the code responsible for the “Options Bar” and add in the microservice which accomplishes the same output. Next, link this new microservice to the existing monolith’s hover action. Finally, replace the code responsible for the hover action and add in the microservice which accomplishes the same output.

**-Recent Chain of messages from message app**

Use Case: User may toggle I/O button labeled “ShimdiggityWooWah” in the third place from the left of the user’s screen. This “pushes” the left side of the screen to the right for 20% the horizontal length of the screen, listing the user’s messages.

**Process Authoring**

**-Create Project UI (see core data, we need one UI service for each of the main categories that feeds user input to that category)**

**Process Execution**

**-Display Status for project elements**

**-Input project, return XML for elements with status**

**-Conglomerate XML into a complete display status**

**-Update Project UI (see core data)**

**-Determine what’s modified via XML element type**

**-Update service**

**Message App**

**-Get Messages**

**-Formatter**

**-Grabber**

**-Validate Recipient**

**-Database Queue Manager - Receiver (One instance per user)**

**-Send Messages**

**-Formatter**

**-Sender**

**-Validate Recipient**

**-Database Queue Manager – Sender**

**Dashboard**

**-Project Summarizer**

**-Projects View Constructor**

**Core Data**

**How it works:** Client initiates XML to server to initiate/query datomic cloud service to ferry new XML to client.

**Roles**

**-Role Determiner (Accepts Project Element, User)**

**-Role Modifiers**

**-Updater**

**-Deleter**

**-Accessor**

**Version Verification**

**-Major Version**

**-Minor Version**

**-Validate Project**

**-Determine least version in project chain**

**-Validate all against version (by walking until all elements are visited)**

**Project Data (access, update, delete, create, construct XML for each)**

**-Project**

**-Process**

**-Stage**

**-Box**

**-Step**

**-Infinite chain of substeps**

**-Data looks like:**

**Database ID / String ID / Fields as needed / Status / start date / duration / link to issue table**

**-XML from datomic looks like:** (XML setup without data fields)

<Project\_ID>

<Process\_ID>

<Stage>

<Box>

<Step>

<Step>

<Box>

<Stage>

<Process\_ID>

</Project\_ID>

**-Get Issues for element (for the severities user wants)**

**-Get severity for issue**

**-Determine end date for element (using duration and start date)**

**-XML Translation per element**

**-Children per element**

**-Parents per element**

**-Attached Files Per Element**

**-Import from template (one service per main element)**

**-Load Template**

**-Export to template**

**-Save Template**

**User (w/ schedules)**

**-Create, Update, Delete, Access**

**-Assign User to project element**

**-Determine if user is available at time block**

Input: From user, input is in the form of a button press that Java “listens” for. Users are hierarchical and the “meta-user” may add users with their credentials to a project element.

Output: Adds user to time block, if available. Otherwise alerts current user that user is not available during that specific time.

**Scheduler**

**-Construct schedule**

**-Validate times for all elements of a project**

**-Determine date**

**-Display actual progress of project**

**-Display projected progress of project**

Procedure: Scheduler will continuously check the “currentness” of each project element and update if necessary. This is accomplished with Java listeners that will send request XML to the server upon user input (e.g. button press/out-of-dateness). The server will then pull from the datomic cloud an XML destined for a specific client ID. Datomic keeps a history of all database states. On our server, we have a dedicated CQRS database that holds only the overall values of each user inputted action (without varying XML versions, as in the cloud) and a running time. With this information, we can process Java graphs and/or trees (both?) that can complimentarily predict the projected progress of an individual project (both based on the history of the given project with running time and global project histories and their final times).