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.

NOTE: Some services could be fairly complex and entangled, even within our ordering. However, as the function they represent is less critical to the core data, we address them earlier, as “getting it right” is more of an annoyance than data-critical. (Additionally, it’s probable leaving all the entangled services for last will produce its own complexities). If this doesn’t actually make sense at implementation time, this ordering may be seen as more suggestion than assertion.

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.

Entanglement with: Send Auto Notification and Notification Sender.

**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.

Entanglement with: Send Manual Notification and Notification Sender.

**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.

Entanglement with: Send Manual Notification and Send Auto Notification.

**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.

Entanglement with: File Downloader, File Database Updater and File Database Accessor.

**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/>

Entanglement with: File Uploader, File Database Updater and File Database Accessor.

**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.

Entanglement with: File Uploader, File Downloader, and File Database Accessor.

**File Database Accessor**

See File Database Updater.

Entanglement with: File Uploader, File Downloader, and 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)**

Input: Interval to sync on, in minutes.

Summary: Maintain one additional service that monitors for changes to the project on either side (Microsoft Project or PieMatrix files) and perform “Convert to Microsoft Project” or “Decode from Microsoft Project” on a set interval.

Entanglement with: These three microservices dealing with Microsoft project are heavily entangled.

**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.

Note: What follows involving the UI and core data are all heavily entangled monolithically.

**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**

**-Pass to core data Update services**

**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**

**Database Manager**

**-Database Management System Access**

**-Database Management System Update**

These services should themselves maintain links to databases on a per-customer level. (There’s update and access to provide new database tables/etc as needed.)

**Core Data**

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

There is a queue between “layers” of the Core Data that propagates requests to update between the layers. Functions that access are kept separate from functions that update at the data level. This means there is a separate queue between the accessor data and databases, and the updater data/databases.

**Roles**

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

**-Role Modifiers**

**-Updater**

**-Deleter**

**-Accessor**

**Scanner**

This service takes in an element of any type, and returns the collection of elements that will be affected should we choose to update this element.

**Version Verification**

The version verification services are by their nature extremely entangled. This design passes a query to the Scanner (which itself sends requests between the core data queues). We then use the major/minor versions of each element, determine the “least” version in the collection, and ensure that all higher versions are marked as compatible with this version.

**-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**

Note: The data representation in all databases will include more fields as needed for services

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

<Project\_ID>

<Process\_ID>

<Stage>

<Box>

<Step>

<Nested substeps go here as needed>

<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**

Note: Children and Parent elements are used to work “up the chain” as needed. Datomic’s structure assists, as the database is returned as an object.

-**Update Queue for Element**

**-Access Queue for Element**

**-Send/recieve for each queue**

**-Attached Files Per Element**

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

**-Load Template**

**-Export to template**

**-Save Template**

**Critical Path Management**

First, use the Scanner to return all related objects to an element when its critical path is modified. This lets us know which elements we need to touch.

The critical path is perceived as important enough so that it gets its own tables in the access and update databases. These tables follow the same structure we’ve used thus far; they’re disparate but have an update queue between them to be kept “in-sync”.

-**Critical Path Manager**

This formats and returns XML information regarding the current state of the critical path. It also could be used as a waypoint for updating the critical path.

**-Critical Path Elements Access**

**-Critical Path Elements Update**

As is per usual for this software, these services have a queue between themselves and their databases.

**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).

Note also that the scheduler is called when an element’s timing is updated in any way. If any conflicts are reported, an error flag for all elements in conflict is set, with links to the elements of core data said element is in conflict with. (A link to the ID).

**Security**

Security should be the last thing to move over – this is because it’s important to get it right, and it works as-is. In fact, there could be an argument to be made to modify security functions as little as possible. With this in mind, we would suggest stripping out all security functions, leaving them as a stand-alone monolithic app, and running copies on enough servers to satisfy load. Rather than suggest specific microservices, here is our recommended approach:

-Strip out all security functions into a monolithic app (do not modify the internals, it’s imperative they work as they always have)

-Create “bridge” functions to pass needed data into this application

This leaves security functionality intact, but provides compatibility with the new architecture.