IUMTWEB Assignment Report

Leoluca D’Atri | 976459

Introduction

This is the report detailing the implementation of the IUMTWEB assignment.

As of writing, some statistics do not compute in the notebook, they error out or their actual visualisation is buggy. As far as I could tell, this is due to the very varying ways data can take shape and or missing data not properly being handled in the visualisations notebook. I must admit that I did not have enough time to properly implement these visualisations with the care that I had wished for, due to focusing heavily on a frontend design as well as the fact that I did not want to hand in the assignment based on the diminished requirements, as I thought having to manually run queries in the notebook was horrible UX. Alas, this came with its cost and they are quite evident. At the end of the development cycle, I realised many things and false assumptions that I had made. It was a tremendous learning experience and when the next big project comes up, I now know how to properly handle it. Among these things is not tackling a large project on a tight deadline alone, no matter how skilled one is, managing so many infrastructures and choices and development is simply too difficult to handle at the same time for a junior developer. Another more technical thing I have learned is that for projects of this scale, monorepos are the way to go as they provide a centralised place for everything, making issue tracking and PRs much easier to deal with, not to mention the implications the multi-repo approach has had with Docker and handing the assignment in.

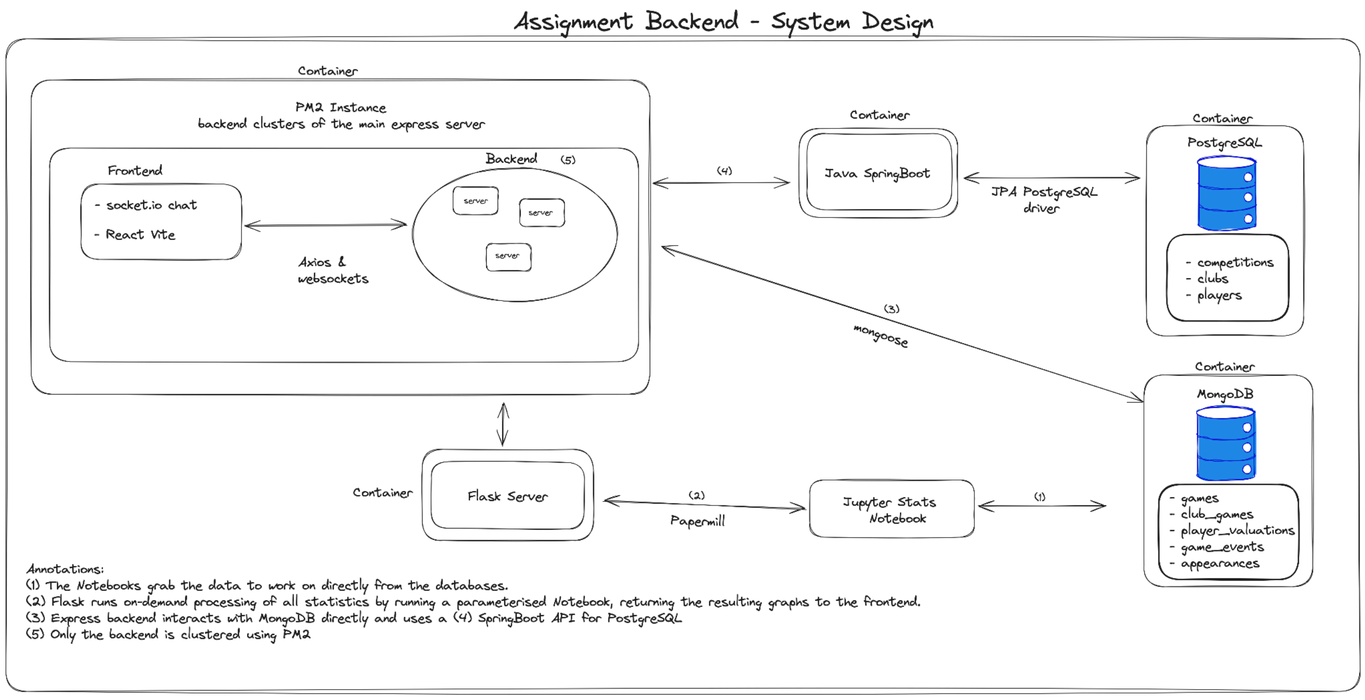
Another technical fault within the project is that GridJS, the library used to display data in a table, seems to have a few issues of its own, namely not properly handling server-side search merged with sort and displaying errors about fetching data when those did not occur. Often the GridJs Plugins (built-in) also crash and a page reload is needed to make search and sort functional again or to make the error at the bottom of the table disappear.
Minor issues are also related to CSS styling. For what it’s worth, in the upcoming days I will be deploying some fixes to address as many issues as I can. Consider this more of an MVP in its deliverable timeframe.

All that being said, I came out of it much wiser than before and that is the most important thing.

Let’s now move on to the more fun part of this report.

Technical Specifications

System Design



Annotations:
(1) The Notebooks grab the data to work on directly from the databases.
(2) Flask runs on-demand processing of all statistics by running a parameterised Notebook, returning the resulting graphs to the frontend.
(3) Express backend interacts with MongoDB directly and uses a (4) SpringBoot API for PostgreSQL
(5) Only the backend is clustered using PM2

Overall additions to the requirements include:

- the use of Docker containers for deployment,
- PM2 for load balancing,
- a Flask server that uses the Papermill API to generate pre-determined statistics,
- React Vite as a frontend framework,
- and PNPM as the faster package manager of choice.

Frontend

Solution

- The Design was first created in Figma. Because I am a very, very bad designer I chose to create a very, very simple frontend (despite this it took 80% of the time to do).
- The web app is built using React Vite in Typescript because I was more familiar with the stack and React provides many advantages as opposed to regular vanilla HTML/JS, such as state management, native and easy routing as well as modularisation into components, making putting together pages much simpler.
- Vite was chosen because it provides faster spin-up times. It inherently does not build the entire application before serving but divides them into dependencies that are bundled using esBuild (10-100x faster) and source code. Since the dependencies are static they can be cached. The source code is served over native ESM. Vite also provides less waiting time to reflect file updates because of its Hot Module Replacement in ESM and also has improved build performance thanks to its many out-of-the-box optimisations.

- GridJs was used to display database data in tables (clubs, competitions, players, etc). Server-side pagination for games was done due to the vast amount of data.

Issues

- The UI could be seen as very poor, depending on who you are asking. It's nothing fancy at all. GridJS breaks (I think) when hot-swapping pages too fast, interrupting a previous load perhaps. Sometimes search and sort breaks due to a tracked issue in react environments. Overall CSS might be icky.

Requirements

- The page enables querying and exploring data and the minimum page requirement is met. The chat system is also implemented.

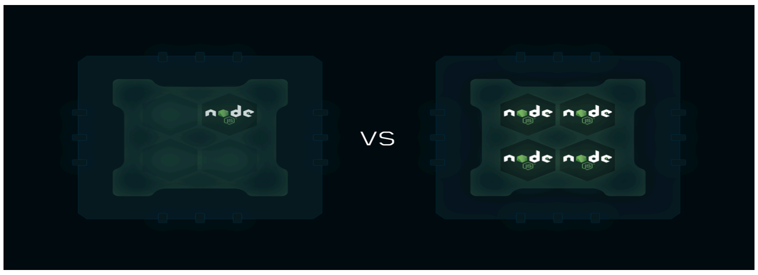
Limitations

- I think the inherent nature of websites built in React is being extensible and modular. The design makes no specific references to the subject of this assignment, so it can be easily adapted to fit other steps or more of them. No immediate scenarios where the frontend could not handle itself come to mind.

Express API and Java SpringBoot API

Solution

- The Express API is load-balanced using PM2. It keeps applications alive forever, reloads them when necessary without causing downtime and distributes the load evenly across all available CPUs by creating child processes sharing the same server port. A pretty simple caching strategy is implemented for the data loaded from the databases. Responses are cached for a specified amount of time before being re-fetched. In an enterprise environment, especially when loading millions of rows of data, this, together with pagination, is crucial in ensuring a smooth UX. Pagination is used for the games data as it is a quite large dataset.



- The SpringBoot server was implemented as was shown in the lectures, using Spring Initializr with the required dependencies. Uses the MVC model and has the database entities defined within, adapted to an **already loaded database schema**.

Issues

- No issues explicitly caused by the Express API that I am aware of. There is however a major problem with the server-side pagination and sorting for the games tables. More details are provided in the frontend repo issues and they have a partial fix in a separate branch. Thanks to its direct connection to MongoDB, things have less of an overhead, especially in latency, and are quicker to implement. SpringBoot has been a bit of a nightmare since you need to use Data Transfer Objects if you want a subset of data or a join between entities. A lot of time was spent on getting it all to work, but, at least now I'm fairly confident with the environment.

- SpringBoot queries are done using JPQL for the ones not natively supported. This might imply a performance tradeoff as opposed to native queries.

Requirements

- The Express API clusters can handle heavy workloads as requested. It communicates with the other two servers as an intermediary for the frontend.
- SpringBoot is used to fetch the data saved in the PostgreSQL database.

Limitations

- SpringBoot isn’t clustered or anything of the sort, so it’s going to take quite a hit under heavy loads. This is however also heavily mitigated by the caching strategy in the express server and the data being static in this assignment. Eliminating the server and integrating the data fetching into the clusters of express servers would probably improve latency and system stability tremendously.

Flask

Solution

- The Flask server implements a few key things:
    - When run in a container, it is started using a WGSI server (gunicorn), fit for production environments.
    - When the request from the express API arrives with the arguments specifying a type of visualisation to run, it uses the Papermill API to run a parameterised Jupyter Notebook. The Notebook has a first parameterised cell, then goes through all cells skipping those not corresponding to the requested visualisation using if statements. The visualisations once generated are exported in an output directory as SVG files. Flask, after the notebook finishes executing, either returns the SVG directly or, if multiple, zips them and returns the zip file. SVGs include a tag to classify the statistic as “advanced”, for the frontend to interpret.
    - The route checks if the visualisation already exists, if they do it returns them directly, saving a lot of processing time.
    - A scheduler task clears the output directory once every 24 hours, to reduce disk space usage

Issues

- Thankfully, having already documented myself on the Papermill API back when the requirements were released, implementing the Flask server with this functionality was easy. The notebook unfortunately has been a big challenge to get right. The dataset is distributed in two databases, which makes querying and joining the data quite tricky, but, more so, defining appropriate statistics and classifying them between standard and advanced, due to being left as one of the last things to do, has been hectic and difficult to properly element, hence the occasional visualisation errors with certain sets of data. I apologise for this.

Requirements

- This implementation complies with the original requirements and I’m quite proud of that.

Limitations

- The notebooks do not always process the data correctly and error out. Much more testing is needed to make this as robust as possible.

Database Structure

Solution

- The dataset was divided into static and dynamic data based on reasonable expectations for data to be updated and its frequency. Players, clubs and competitions were saved in PostgreSQL, whereas the other data is saved in MongoDB. The databases were loaded after a data analysis and preprocessing task in the Jupyter notebooks.

Issues

- PostgreSQL is the worst DBMS I have ever worked with and I will never touch that again. Never seen a worse implementation both for the end-user and for the actual database processing. If you export data from the database for importing elsewhere, even in a blank database, you better be sure to test the importing regularly. For whatever reason exporting regularly by using the proprietary GUI or even just SQL causes infinite syntax errors. Exports are a breeze in alternatives such as MySQL and MariaDB.
- MongoDB doesn't escape these issues but was much easier to properly figure out and implement reliably. Exporting was a breeze and so was importing, so long as the user with appropriate permission existed in the database before importing data to it.

Requirements

- The division of data should respect the requirements.

Limitations

- Having to divide the data into different databases, for this particular use case has been quite a challenging environment and much more research is needed to go into how to deal with the data, whether it respects a more structured approach or is more in line with NoSQL. This division was solely based on the update frequency.

Data Processing and Analysis

Solution

- The entire dataset was loaded into a notebook, one for each CSV file and was analysed in a storytelling fashion, checking missing data values, and immediate outliers through exploratory data analysis.

Issues

- Storytelling is still storytelling and everyone can extrapolate a different or even objectively better story from the same dataset. I've tried my best to make sure the dataset is as usable as possible. One quite important thing to note is that the dataset seems to stem from a mid-production version. Looking around in the GitHub repository for the dataset and opening a discussion question, there is an extra column `n`, which is only used by the dataset maintainer to deduplicate records during data preparation stages.

Requirements

- I think the storytelling approach leaves quite a broad spectrum of interpretation but I believe to have done an adequate job in this step of the process.

Limitations

- Not realistically applicable

Conclusions

Although I have already gone off on a tangent with my conclusions in the introduction to this report, I'll re-emphasise the key aspects and things I have learned throughout this project's development.

The challenges faced, overall have provided quite valuable insights into the complexities of a multi-service modern web application.

Importance of comprehensive planning
The difficulties encountered in implementing visualizations and handling varying data formats highlight the necessity for thorough planning, especially in data handling and visualization strategies. This underscores the need for a more robust planning phase in future projects Although I had planned to use Docker from the get-go, leaving it as one of the last things to add to the project was a big, big mistake. But hey, if I were to do it again it would take me an hour or two at most, not 4 days of troubleshooting.

Time Management and Focus
The significant time invested in frontend design came at the expense of other project aspects. This stresses the importance of a balanced time allocation across all project components, proportional to the developer's skill levels, who, ideally wouldn't tackle all of this alone.

Teamwork over solo efforts
The challenges faced in managing the breadth of this project alone have underlined the value of teamwork. Collaborative efforts, especially in large projects, not only distribute the workload but also bring diverse perspectives and expertise, enriching the development process.

Technical Choices and Their Impact
The selection of "add-on" technologies like Docker, PM2, React Vite, and GridJS had substantial impacts on the project's execution. Each technology brought its strengths and limitations, teaching the important lesson of always carefully evaluating and choosing the right tools for the job.

So, in essence,
 The project, while meeting its core objectives, leaves room for future improvements, but shows a good understanding of where the problem points are and how to improve on them. Developing this has been a journey of professional and personal growth. The lessons learned extend beyond the technical skills and all build up to making me a better developer, in every way.

Extra Information - How to Run
I'm running out of the allowed page limit, but running the project is straightforward. Start by cloning the hand-in repository at https://github.com/Rithari/FDP_Handin. The readme provides further details and necessary steps to take to make sure the containerisation works. The other repositories should be used as a reference for the development cycle.