

Web Development Environment 2

Final Project

Proof of Concept : Marathon Website

Group 2:

Ashley Arenas

Brendan Delaleu

Sarah Gisondi

Presented to:

Silviya Lyubomirova Paskaleva

Roadblock #1:

At the beginning of this project, we underestimated the scope and depth for this project. We simply knew that this project would be a big learning curve in our programming journey, which was exciting and intimidating at the same time. We simply started working on the project without really understanding the objective and goals for this POC.

Problem Statement

Context:

In this project, we are simulating a scenario where a marathon website's registration system is not functioning as expected, leading to issues with form submissions. The system fails to handle user registrations properly and given that marathon registration systems often deal with thousands of users within a short period of time, such a flaw can cause serious issues to event's success.

The Importance of a Good Registration System:

With major marathon events, in which the demand often exceeds available spots, the registration process is the most critical aspect for runners to get a chance of participating in the race. It is to note that there are multiple ways to be qualified for a major race. Elite runners are automatically qualified, but for regular runners wanting to participate, they have to enter a lottery system in which the event uses a random draw system from the registrations to select the participants. This further indicates that a fast and reliable registration process is essential for such a website.

For instance, more than 200,000 people applied for a place in the NYC Marathon lottery this year, but only around 2%-3% of the applicants will be accepted through the general draw (Reuters, 2025, para. 1).

Objective for POC

The objective for this proof of concept is to ensure fast and reliable form submissions, meaning there should be no crashes or lost submissions during the registration process. The system must also be able to handle traffic spikes, with the capability to manage 1000 concurrent requests efficiently. Clear and immediate communication with participants is essential, so instant user feedback on their status (such as success or failure messages) should be provided. The database handling must be reliable, ensuring that all entries are processed correctly and stored securely. Finally, Scalability is another key goal, with the system designed to handle massive spikes in traffic without failure.

The Learning Objective For This Project

For this group project and for learning purposes, we decided that building a simple marathon registration page would be a good place to start with. It gave us a clear, real-world scenario that's visually easy to understand, which made it ideal for experimentation.

We wanted to use this opportunity to explore different tech stacks and gradually apply concepts and tools we learned in class, such as backend routing, database integration, and Docker-based deployment.

Roadblock #2:

Choosing a stack that we have some or experience with. Because we touched on Node.js and MongoDB in this class, as well as Python and Sqlite in our dynamic web programming class, we decided to stick to these technologies for the scope of the POC. However, there might be other technologies more conducive to this project that we didn't explore. Even with some experience, we encountered multiple hours of debugging sessions, integration challenges, difficulties with stress testing and deployment issues.

Develop a Prototype

How We Approached This Project

This POC will explore and compare two versions of the same marathon registration system concept, comparing a Node.js backend and a Python backend, each connected to a shared MongoDB database. Also, for the frontend side, we use the same form for both versions using Bootstrap to simulate real user interactions with jQuery and AJAX to handle form submissions.

To be able to compare which stack is more efficient at handling requests, we ensured that both implementations used the same elements:

- Identical endpoints (POST /register)
- Identical data models (participants: name, email, age, race distance, medical conditions)
- Same frontend form structure to ensure consistent input and behavior
- Simulate same traffic load

The Choice Behind the Stack Used

Why Node.js and Python as backend?

For our group project, we chose to use Node.js and Python as backend technologies primarily because we wanted to apply what we learned in a practical context. Furthermore, both technologies are widely popular and used in real world applications which makes it easier to find solutions to bugs during development as they have lots of documentation.

According to the Node.js' official website (About Node.js., n.d., para 1), some key pros of Python include:

- Asynchronous, event-driven architecture:
 - Handles multiple connections efficiently without blocking.
 - Avoids deadlocks.
 - Supports real-time, low-latency communication.
- Seamless integration with JavaScript:
 - Enables full-stack development using one language.
- Ideal for building scalable network applications:
 - Makes it a strong choice for fast, scalable web systems.

- Well-suited for use cases like a marathon registration system

According to Python's official website (*What is Python? Executive Summary*, n.d., para. 1), some key pros of Python include:

- Easy to learn and read syntax, making it ideal for beginners and maintaining code.
- Rapid application development due to dynamic typing and quick edit-test-debug cycles.
- Modularity and code reuse supported by modules and packages.
- Strong debugging support with built-in tools.
- Freely available and cross-platform.

Thus, with our problem statement in mind, we believe that Node.js and Python are effective backend options for prototyping and integrating different systems.

Why MongoDB as a database?

We selected MongoDB instead of SQLite because MongoDB is designed for scalability and flexibility as opposed to SQLite, which is a lightweight, file-based relational database best suited for small-scale applications. As we wanted to simulate a real-world scenario with our marathon registration system, we believe that MongoDB would be the best option for large applications managing massive amounts of data (Ferrell, 2023, para. 10).

Roadblock #2.1:

One of the key challenges that we faced was that, although the backend server was running successfully, the registration form kept failing to send data to the backend and ultimately to the database.

Why Docker for deployment ?

For this POC, Docker was used to simplify deployment and testing of the backend services. It allowed us to containerize both the application (Node.js or Python) and the MongoDB database, ensuring consistency across development and testing environments. Docker made it easy to replicate the same setup on different machines without worrying about environment configuration or dependency issues. This was especially valuable during load

testing and stack comparisons, as it enabled quick setup, teardown, and scalability testing (Developers bring their ideas to life with Docker, n.d., para. 3).

Roadblock #2.2:

The hardest part of the project was containerizing the entire application and deploying it to a cloud environment instead of running it locally. While Docker made it possible to bundle the backend and database services into separate containers, there were several challenges in setting up the necessary environment for production deployment. When testing docker individually, only one team member was able to successfully open the project, but the other encountered many problems.

Test the Prototype

Once we were finally able to deploy the project using Docker, the next step was to test the performance of the marathon registration system under load. To simulate real-world traffic and ensure our system could handle a high number of concurrent users, we used the load testing tool Artillery to :

- Simulate thousands of concurrent requests to the registration endpoint.
- Measure system performance, including response times and any potential bottlenecks.

Roadblock #3:

During the load testing phase with Artillery, we encountered a significant issue that hindered our ability to effectively test and compare the different stacks.

When we conducted the tests, we received 200 OK response for only one request, while the rest returned 400 Bad Request errors. We discovered that the problem stemmed from a constraint in our backend that only allowed unique email addresses to register. Further investigation revealed that MongoDB was receiving the email string correctly, but because the test was not generating unique emails as expected, the load test failed to properly simulate randomized user registrations. This problem impeded our ability to conduct accurate stress tests on the system.

```

tests > ! load-test-python.yml
1  config:
2    target: "http://localhost:5000"
3    phases:
4      - duration: 10
5        arrivalRate: 1
6    defaults:
7      headers:
8        Content-Type: application/x-www-form-urlencoded
9
10   scenarios:
11     flow:
12       - post:
13         url: "/register"
14         formType: urlencoded
15         body:
16           name: "Test User"
17           email: "{{ randomString(8) }}@example.com"
18           age: "30"
19           distance: "10K"
20           medical: "None"

```

```

_id: ObjectId('67f89ada18747e44b6c81a35')
name: "Test User"
email: "{{ randomString(8) }}@example.com"
age: "30"
distance: "10K"
medical: "None"

```

Stress Test - Python

Objective	Result Summary
Fast & reliable form submissions	1459 successful form submissions with 0 crashes. 41 timeouts (2.7%) under peak.
No crashes or lost submissions	No crashes recorded. All completed requests either succeeded (200) or timed out — no silent failures.
Handle traffic spikes (~1k+ concurrent requests)	Successfully simulated 1500 virtual users. ~97% completed, but 2.7% timeout under peak load.
Instant user feedback (success/failure)	Success (200) and failure (timeout) were clearly logged;
Reliable database handling (correct, secure entry storage)	All 1459 completions likely stored data.
Scalability (handle massive spikes without failure)	Handled 1500 VUs with minor timeouts

Summary:

- The system handles high concurrent load well, with no crashes and solid completion rates.
- ETIMEDOUT (41 users) during peak indicates backend/server scaling limits which could use improvements.
- No loss of data or silent failure suggests the backend is robust.

Stress Test - Node.js

Objective	Result Summary
Fast & reliable form submissions	1500 form submissions were sent and all 1500 were successful (HTTP 200).
No crashes or lost submissions	No user failures or dropped requests. 0 failed users, 0 timeouts.
Handle traffic spikes (~1k+ concurrent requests)	Simulated 1500 virtual users, and all handled successfully.
Instant user feedback (success/failure)	Every request received an HTTP 200.
Reliable database handling (correct, secure entry storage)	No errors indicate that submissions were stored properly.
Scalability (handle massive spikes without failure)	This test suggests strong scalability potential. More intense bursts would confirm horizontal scaling ability.

Summary:

- Perfect reliability and zero errors under test.
- Solid capability for high concurrency (1500 VUs).
- All HTTP responses were successful (200 OK), implying good backend handling.

The raw results for the stress test are provided in the load-test-node-results.json and load-test-python-results.json files

Node.js vs Python

Criteria	Python App	Node App
Total Requests Sent	2970	1500
Successful Submissions (HTTP 200)	1459	1500
Failed Submissions	41 (Timeouts)	0
Completion Rate	~97.3% (1459/1500)	100%
Errors (ETIMEDOUT, etc.)	41 (ETIMEDOUT)	0
Average Response Time	~1.8s (best run) to ~6.5s (slowest run)	~26ms
Scalability Potential	Handled up to 1500 VUs but with some failure under load	Handled 1500 VUs with no degradation
Stability under Load	Some instability under higher load (timeouts, slower)	Stable under test (no timeouts/errors)

Final Recommendation

Given that the frontend (HTML/CSS) and database (MongoDB) remain constant, the comparison between Python (Flask) and Node.js (Express) for the backend centers on performance, reliability, and scalability under load.

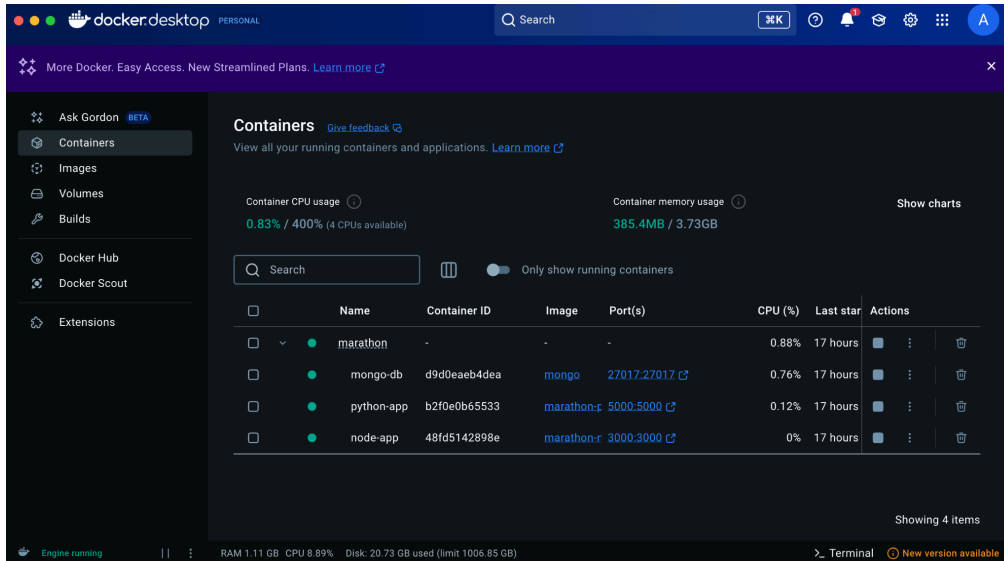
In the POC, Node.js significantly outperformed Python in terms of stability and responsiveness. The Node backend processed 1,500 concurrent requests without any failures, achieving a 100% success rate with consistently low response times. In contrast, the Python backend experienced 41 timeouts out of 1,500 users, suggesting potential issues under high concurrency.

Therefore, for a high-traffic registration system requiring real-time feedback, stability, and quick response under load—Node.js is the better backend choice. It integrates well with MongoDB and offers better scalability with form submissions. Sticking with Node.js for the backend will maximize system reliability and performance while keeping the tech stack simple and efficient.

Caveat

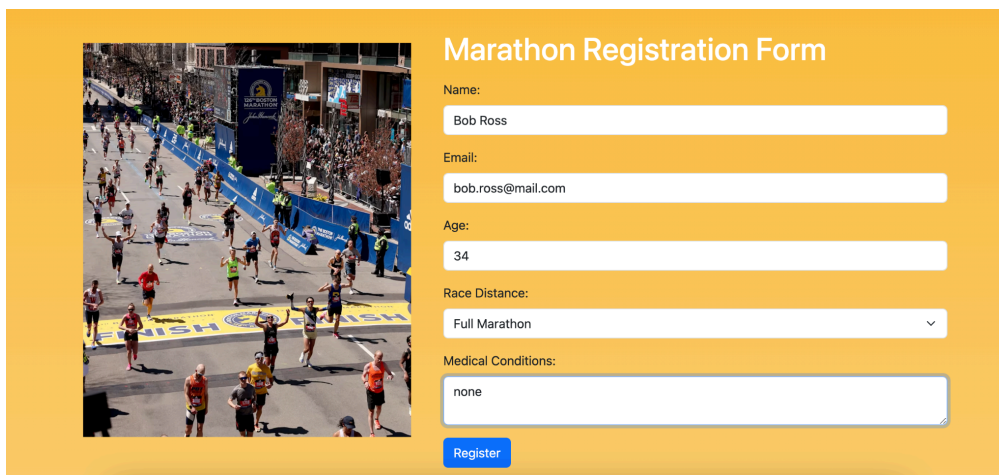
Since our main goal for this project was to validate functionality and scalability, we utilized available resources without analyzing the financial implications. A detailed cost analysis would be more appropriate in a future phase, once a preferred stack is chosen for full-scale deployment.

Visual Overview of POC



The screenshot shows the Docker Desktop interface. The left sidebar contains navigation options: Ask Gordon, Containers, Images, Volumes, Builds, Docker Hub, Docker Scout, and Extensions. The main area is titled 'Containers' and displays a table of running containers. The table has columns for Name, Container ID, Image, Port(s), CPU (%), Last start, and Actions. Four containers are listed: 'marathon', 'mongo-db', 'python-app', and 'node-app'. The 'marathon' container is highlighted. Below the table, it says 'Showing 4 items'. At the bottom, there is a status bar showing 'Engine running', RAM usage (1.11 GB), CPU usage (8.89%), and disk usage (20.73 GB used of 1006.85 GB limit).

	Name	Container ID	Image	Port(s)	CPU (%)	Last start	Actions
<input type="checkbox"/>	marathon	-	-	-	0.88%	17 hours	
<input type="checkbox"/>	mongo-db	d9d0eae4dea	mongo	27017:27017	0.76%	17 hours	
<input type="checkbox"/>	python-app	b2f0e0b65533	marathon-p	5000:5000	0.12%	17 hours	
<input type="checkbox"/>	node-app	48fd5142898e	marathon-r	3000:3000	0%	17 hours	



Marathon Registration Form

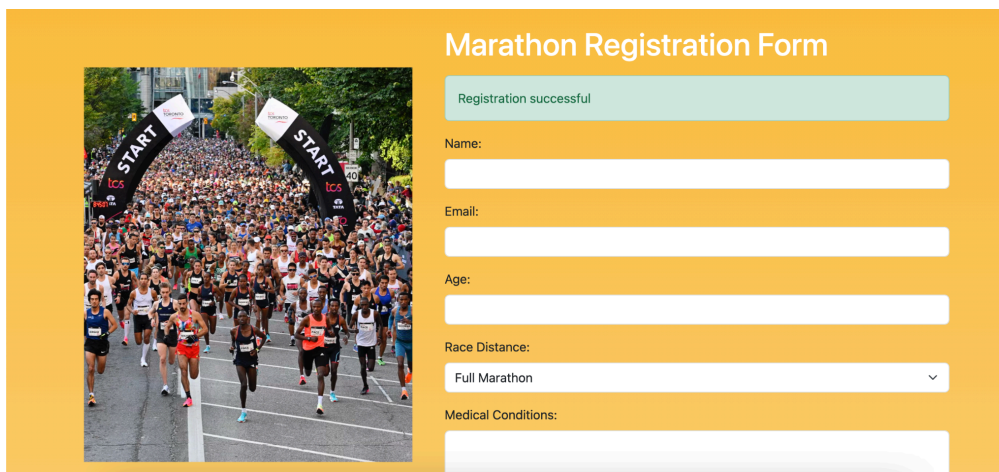
Name:

Email:

Age:

Race Distance:

Medical Conditions:



Marathon Registration Form

Registration successful

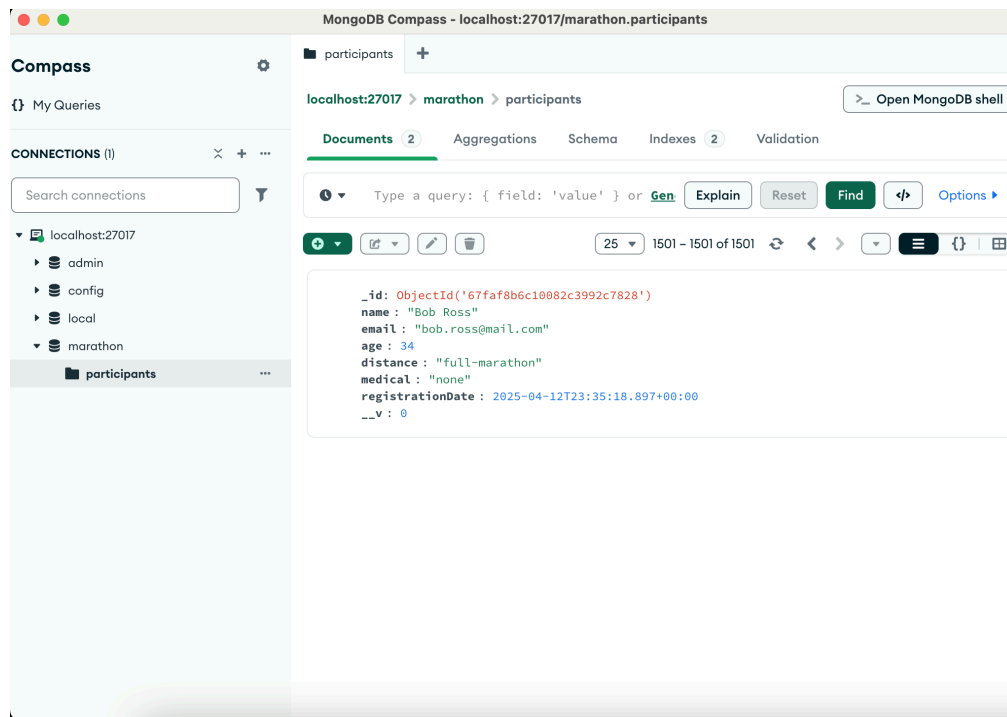
Name:

Email:

Age:

Race Distance:

Medical Conditions:



References

- *About Node.js*. (n.d.). Retrieved from <https://nodejs.org/en/about>
- *Developers bring their ideas to life with Docker* (n.d.). Retrieved from <https://www.docker.com/why-docker/>
- Ferrell, R. (2023). *SQLite vs mongoDB: Comparing Two of the Most Popular Databases*. Retrieved from <https://sqldocs.org/sqlite-database/sqlite-vs-mongodb/>
- Reuters (2025). *NYC Marathon draws record applications of more than 200,000*. Retrieved from <https://www.reuters.com/sports/athletics/nyc-marathon-draws-record-applications-more-than-200000-2025-03-05/>
- *What is Python? Executive Summary*. (n.d.). Retrieved from <https://www.python.org/doc/essays/blurb/>