# Building a Web Server

Chungman Lee
*dept. of Computer Science*
*University College Dublin*
Dublin, Ireland
chungman.lee@ucdconnect.ie

Hitesh Unnam
*dept. of Computer Science*
*University College Dublin*
Dublin, Ireland
hitesh.unnam@ucdconnect.ie

Nithin Shashidhara
*dept. of Computer Science*
*University College Dublin*
Dublin, Ireland
nithin.shashidhara@ucdconnect.ie

Simran
*dept. of Computer Science*
*University College Dublin*
Dublin, Ireland
simran.1@ucdconnect.ie

Tejus Ravi
*dept. of Computer Science*
*University College Dublin*
Dublin, Ireland
tejus.ravi@ucdconnect.ie

***Abstract*—This project tackles the challenge of designing a scalable, efficient, and reliable web server capable of processing multiple domains while managing a high volume of concurrent requests. The primary objective was to develop a server that not only supports HTTP/1.1 methods like GET, POST, PUT, and DELETE but also ensures compliance with HTTP/1.1 features, including persistent connections, host headers, and proper handling of status codes. The project scope was later expanded to include secure communication through HTTPS. Additionally, an admin portal was developed to provide a centralized platform to manage hosted applications, monitor application status and host new applications on the web server. The project was implemented in Java using Spring Boot and Reactor frameworks, the solution makes use of non-blocking (NIO) and reactive programming for optimal performance. The server's architecture integrates a MySQL database for managing site configurations and user data while ensuring reliability and adaptability for various application needs. Utilized Docker for containerization, simplifying deployment and enhancing maintainability, while leveraging Microsoft Azure for cloud deployment, ensuring high availability. The server demonstrated good performance under various load conditions, reliable handling of concurrent requests, secure data transmission, and compliance to HTTP/1.1 standards.**

***Index Terms*—Scalable Web Server, HTTP/1.1, Spring Boot, WebFlux, MySQL, Docker, Azure, HTTPS, Non-blocking I/O, Reactive Programming**

## I. Introduction

In today's digital ecosystem, the need for a robust, scalable, and reliable web server is more critical than ever. As businesses and services expand their online presence, they encounter challenges in managing multiple domains while ensuring seamless performance, security, and compliance with modern web standards. This project addresses these challenges by developing a web server designed to efficiently handle concurrent requests across various domains while ensuring HTTP/1.1 compliance and scalability. The server was designed to maintain high performance and reliability, even under heavy loads, while being easy to set up and secure.

The initial objective of this project was to build a web server that not only meets the technical demands of modern web applications but is also scalable, highly available, and future-proof. A significant challenge was implementing a non-blocking, reactive server architecture using Java with Spring Boot and WebFlux. This approach was chosen to take advantage of reactive programming principles which allows the server to handle a high volume of requests with minimal performance degradation. Security was a core focus with the implementation of HTTPS to ensure secure communication through SSL/TLS certificates, protecting data integrity and confidentiality. Compliance with HTTP/1.1 standards was another requirement such that features such as persistent connections, host headers, proper status code management, and cache control were fully supported.

The integration of a MySQL database was crucial for managing site configurations and user data, enabling the server to scale effectively as more domains are added while also ensuring maintainability and data integrity. The database's role in the architecture was designed to support the server's scalability and reliability, handling growth without sacrificing performance.

Integrating a PHP parser within the Java-based server environment was another challenge. This integration required innovative solutions, such as using a process builder to invoke the PHP-CGI executable, ensuring efficient and secure execution of PHP code. This setup-maintained HTTP/1.1 compliance specifically in managing persistent connections and connection handling.

Finally, designing a RESTful API to connect the admin portal to the main system involved ensuring the portal was responsive, secure, and capable of real-time updates. This API was crucial for handling CRUD operations securely while integrating seamlessly with the MySQL database and managing sessions effectively.

These technical challenges understate the complexity and importance of this project, which not only involved building a scalable, secure, and maintainable server architecture but also integrated technologies such as Docker for containerization and Azure for cloud deployment. The resulting system is flexible, compliant with modern web standards, and capable of supporting modern web applications efficiently and reliably.

## II. User Stories

Building on the challenges and solutions outlined in the abstract and introduction, the practical application of this project's web server becomes evident through various user scenarios that underscore its necessity in today's digital landscape. As businesses increasingly rely on their online presence, the ability to efficiently manage multiple domains while maintaining performance under heavy user loads has become a critical need. This scenario is particularly relevant for mid-sized e-commerce businesses that manage several websites, each representing different product lines, regions, or markets. These businesses are often tasked with ensuring that all their sites remain operational, secure, and responsive, even during periods of peak traffic, such as sales events, holiday seasons, or new product launches.

Traditionally, the technical burden of managing such a setup could involve the deployment of multiple servers, the use of complex load balancers, and the allocation of a dedicated IT team to monitor and maintain operations. This approach not only incurs significant costs but also introduces potential points of failure, as managing disparate systems can lead to inconsistencies and downtime. However, with the implementation of the scalable web server developed in this project, businesses can centralize their domain management under a single, robust system. This server, designed with high concurrency in mind with reactive programming techniques, ensures that a business's online presence remains uninterrupted, regardless of the fluctuations in user demand.

Moreover, in industries where secure user interactions are important such as in e-commerce, online banking, or platforms that handle personal user information implementing HTTPS across multiple domains presents its own set of challenges. These include the complexities of SSL/TLS certificate management and the risk of improper configurations, and the need for consistent security policies across all domains. The web server developed in this project directly addresses these concerns by integrating HTTPS functionality natively within its architecture. This integration ensures that all data transmitted between the server and clients is encrypted, significantly reducing the need for technical knowledge for businesses. Users benefit from enhanced security without the need to engage in the complex process of certificate management.

The project's admin portal further simplifies domain management by providing a user-friendly interface accessible through API calls. Imagine a scenario where a business needs to update product information across all its domains in real-time. Previously, this task might have required manual updates on each site or the deployment of custom scripts to propagate changes, both of which are time-consuming and prone to errors. However, with the admin portal, these updates can be made swiftly and securely through a centralized interface. The portal, supported by a MySQL database and secured through well-defined RESTful APIs, empowers businesses to respond rapidly to market demands, ensuring that their websites always reflect the latest information. This system is designed to be intuitive, requiring minimal technical knowledge, which allows business owners and managers to focus on strategic decisions rather than operational complexities.

By taking advantage of Azure's cloud infrastructure, the project benefits from enhanced scalability which allows the web server to handle traffic loads efficiently. Azure's global reach ensures that the server can provide low-latency access to users wherever they access it from which is crucial for businesses targeting international markets. Furthermore, Azure's integrated services, such as automated backups, disaster recovery, and security compliance, provide an added assurance of reliability and security, making the server a more reliant solution for businesses. This cloud-based deployment not only reduces the need for on-premises hardware but also offers a flexible, cost-effective model that can grow with the business.

In other words, the web server developed in this project not only addresses the immediate technical challenges faced by businesses and individual users but also equips them with a powerful tool to enhance their day-to-day operations. By providing a scalable, secure, and easy-to-manage solution, this server enables businesses to expand their online presence, optimize their digital presence, and deliver greater value to their customers. The system's ability to adapt to future demands, combined with the reliability and scalability provided by Azure, ensures that it remains a valuable asset as businesses grow and evolve.

## III. Literature Review

The development of a high-performance, scalable, and secure web server requires a deep understanding of both traditional and modern advancements in web server architecture and technologies. This literature review critically examines existing research and practices that have influenced contemporary web server implementations. Evaluating these sources with the current implementation to demonstrate effectiveness and innovation of the solutions employed in this project, while also identifying how the approach aligns with or deviates from established methodologies. This review sets the academic and technical context for the project, providing a comprehensive framework for assessing its design and performance.

### *A. Evolution of Web Server Architectures*

The development of web server architectures has seen significant shifts over the years, primarily driven by the need to handle increasing traffic and deliver content more efficiently. Early web servers, such as those built using multi-threaded models, were effective for managing the web's initial, less demanding needs. However, as the internet grew and web applications became more complex, these traditional models began to show their limitations [1].

The multi-threaded approach, where each request spawns a new thread which led to performance issues, especially under high traffic conditions. This is due to the high resource consumption associated with managing multiple threads, including the overhead of context switching and memory allocation [2]. Recognizing these challenges, developers began exploring

non-blocking, asynchronous architectures as a way to improve scalability and performance.

Reactor and Java NIO are examples of technologies that represent this new direction in web server design. Reactor, in particular, implements a non-blocking, event-driven model that enables servers to handle multiple requests concurrently without the bottlenecks typically associated with traditional thread-based systems. Java NIO complements this by allowing I/O operations to be processed asynchronously, freeing up resources for other tasks and improving overall efficiency [3]. The combination of these technologies in modern web servers has been shown to significantly enhance performance, particularly in environments with high traffic volumes, as evidenced by recent studies [4].

### *B. Managing Concurrency and Enhancing Performance*

One of the core challenges in web server development is managing concurrency, ensuring that the server can handle simultaneous connections without performance degradation. Traditional servers often rely on thread pools to manage concurrency, but this approach can become inefficient as the number of connections grow. The overhead involved in managing large numbers of threads can lead to increased latency and reduced throughput, particularly under heavy loads [4].

To address these issues, modern servers have increasingly turned to reactive programming frameworks like Reactor. By adopting a non-blocking, event-driven approach, these frameworks allow servers to manage a high number of concurrent requests more efficiently. Reactor's Mono and Flux types, for example, provide powerful tools for handling asynchronous operations, enabling the server to continue processing requests even when waiting for I/O operations to complete [5].

In practical terms, this approach has been validated through performance testing with tools like Apache Bench and JMeter. These tests have demonstrated that servers built on non-blocking architectures, such as the one implemented in this project, can maintain high levels of throughput and low latency, even under significant load. This is largely due to the reduction in idle time for threads, which are only engaged in processing when necessary, thereby maximizing resource utilization [6]. The ability to scale horizontally using Docker containers further enhances this approach, allowing the server to maintain performance as traffic increases [7].

### *C. Integrating Security: HTTPS and SSL/TLS*

As the importance of web security has grown, so has the need for robust encryption protocols like SSL/TLS. These protocols are essential for protecting data transmitted between clients and servers, ensuring that sensitive information remains confidential and secure. However, implementing SSL/TLS can introduce performance challenges, particularly when managing a large number of secure connections [8].

The integration of SSL/TLS into a web server requires careful balancing of security and performance. The choice of cipher suites and the configuration of SSL/TLS settings were optimized to maintain security without compromising the server's responsiveness [9].

Research supports this approach, indicating that with proper configuration, SSL/TLS can be integrated into high-performance web servers without significant degradation in performance. Testing conducted in this project confirmed these findings, showing that secure connections could be maintained without introducing substantial latency, thus providing both security and efficiency [9].

### *D. Database Integration and Scalability*

For web servers that handle dynamic content and multi-domain hosting, effective database integration is crucial. Relational databases like MySQL are widely used in this context due to their reliability and ability to handle complex queries. However, integrating a database into a web server architecture poses challenges, particularly when dealing with high concurrency [10].

In the server developed for this project, MySQL was used to manage domain mappings and facilitate dynamic content delivery. Techniques such as connection pooling and query optimization were employed to ensure that the server could efficiently manage multiple concurrent requests without becoming a bottleneck [11]. Connection pooling, in particular, played a key role by reducing the overhead associated with establishing new database connections, allowing the server to handle more queries simultaneously [12].

### *E. Containerization and Deployment Strategies*

Containerization has become a cornerstone of modern web server deployment strategies. Platforms like Docker allow developers to package applications and their dependencies into isolated containers, which can run consistently across different environments. This capability is crucial for ensuring that web applications behave reliably regardless of where they are deployed [7].

In this project, Docker was used to containerize both the web server and the MySQL database, providing several key advantages. First, containerization ensured that the server environment remained consistent across development, testing, and production. This consistency is essential for preventing environment-specific issues that can disrupt deployment and operation [13],[14].

Moreover, Docker's ability to facilitate horizontal scaling was a significant benefit. By deploying additional containers, the server could easily scale out to handle traffic surges, ensuring that performance remained stable even under heavy loads. Docker Compose was employed to orchestrate these containers, managing the dependencies between them and ensuring that all services started and interacted correctly [7], [14].

The use of containerization and orchestration tools like Docker and Docker Compose not only simplified deployment but also provided a robust framework for managing the complex demands of modern web applications. This approach has proven effective in maintaining high availability and performance in diverse deployment scenarios.

## IV. Implementation

The project aimed to develop a high-performance, scalable, and secure web server by integrating modern technologies such as Reactor, Java NIO, MySQL, Docker, and SSL/TLS. This section outlines the orchestration of these technologies to create solution that meets system requirements.

### A. Asynchronous and Non-blocking Architecture

*1) Reactor's Role :* Reactor implemented an event-driven, non-blocking processing model that allowed the server to handle multiple client requests concurrently without traditional bottlenecks. Mono was used to manage asynchronous operations like HTTP requests (`GET`, `POST`, `PUT`, `DELETE`), ensuring efficient resource usage. Flux can be added in the further scope for appropriate needs.

*2) Java NIO Integration :* Java NIO's non-blocking I/O capabilities complemented Reactor's model, using channels and selectors to manage connections. This combination allowed the server to maintain responsiveness under high load, meeting the requirement for handling multiple concurrent requests.

### B. Support for HTTP/1.1 and Multi-Domain Hosting

*1) HTTP/1.1 Compliance:* The server adhered to the HTTP/1.1 standard, handling requests with appropriate status codes (e.g., 200 OK, 404 Not Found), ensuring consistent and predictable client-server interactions. Various status codes for HTTP/1.1 are implemented and can be used as necessary by web applications.

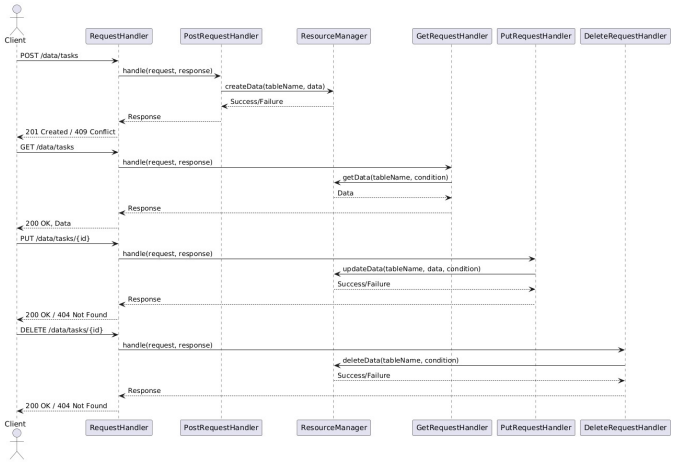


Fig. 1. Sequence Diagram

*2) MySQL for Multi-Domain Hosting :* MySQL managed domain-to-root directory mappings, enabling the server to dynamically serve multiple websites. This setup provided flexibility in adding or removing domains without server disruption, effectively supporting the hosting of multiple websites.

The server features a dynamic data storage system where, upon receiving a `POST` request from the client, the ResourceManager checks whether the target table exists. If the table does not exist, ResourceManager dynamically creates the table and then stores the data. If the table already exists, the data is simply stored in the existing table.

While this system provides flexibility by allowing the server to adapt to varying data storage needs, it does not allow the client to design or modify the table schema. This limitation poses challenges for ensuring data normalization and integrity. As such, a future upgrade to the system should focus on enabling clients to define table schema, ensuring better adherence to normalization principles and enhancing data management capabilities.

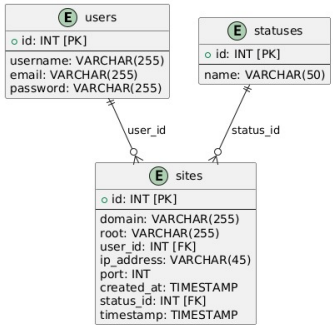


Fig. 2. ER Diagram

### C. Containerization and Deployment with Docker

*1) Consistent Environment :* Docker containerized the web server and MySQL, ensuring consistent operation across development, testing, and production environments. Docker's lightweight containers facilitated horizontal scaling to manage increased traffic.

*2) Orchestration with Docker Compose:* Docker Compose managed the startup sequence and dependencies between containers, ensuring a stable deployment. This orchestration minimized issues during startup and streamlined the management of containerized services.

### D. Security: HTTPS Implementation

*1) SSL/TLS Implementation :* The server used SSL/TLS certificates managed via a Java KeyStore (JKS) to enable HTTPS. The SSLConfiguration class established a secure SSL context, ensuring encrypted communication and protecting against common security threats.

*2) Integration :* The HTTPS setup was seamlessly integrated into the server, maintaining security without compromising performance. This fulfilled the system's security requirements, ensuring safe user interactions.

### E. Performance Optimization and Scalability

*1) Performance Testing:* Tools such as JMeter and Apache Bench were used for benchmarking, identifying bottlenecks that were mitigated by optimizing Reactor's non-blocking model. This ensured the server maintained high throughput under significant load.

*2) Scalability via Docker :* Docker enabled easy horizontal scaling by adding more containers to handle traffic surges, ensuring consistent performance across varying loads.

### F. Dynamic Content Delivery with PHP Support

*1) PHP Integration - Server-Side Processing :* The server supported dynamic content delivery by executing PHP scripts via the php-cgi binary, allowing for more complex web applications. This extended the server's capabilities, supporting both static and dynamic content efficiently.

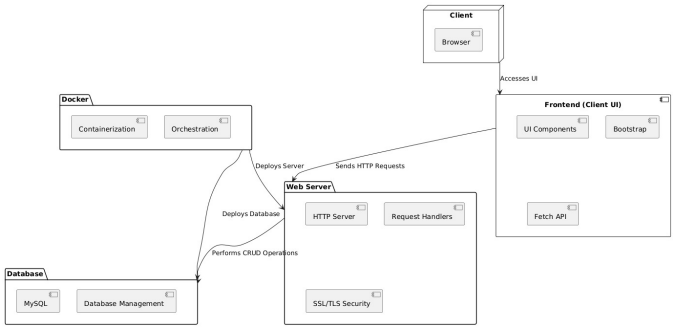


Fig. 3. Component Diagram

This component diagram provides an overview of the key components and their interactions within the system: the Client, Frontend, Web Server, Database, and Docker. The Client accesses the Frontend UI via a browser, while the Frontend uses the Fetch API to send HTTP requests to the Web Server. The Web Server processes these requests and interacts with the MySQL Database to manage data. Docker is used to containerize the Web Server and Database, with Docker Compose orchestrating their startup and network configurations, ensuring a consistent deployment environment.

### G. Admin Portal for Web Server Management

In addition to the core functionalities of the web server, the implementation of an admin portal adds substantial value by providing a centralized platform for managing and controlling hosted websites. Developed using Spring Boot and Java, this portal plays a pivotal role in enhancing the server's operational efficiency and user experience.

The admin portal was designed to address several critical needs in web server management:

*Automated Website Deployment:* The portal streamlines the process of deploying new websites by automating the handling of ZIP file uploads. Administrators can upload a ZIP archive containing the website's source code, which the portal processes to extract and deploy the site. This automation reduces manual errors and accelerates deployment.

*Dynamic Site Management:* Administrators can manage existing websites through the portal. This includes functionalities such as stopping or restarting applications, as well as removing sites that are no longer needed. Such management capabilities ensure that the server remains optimized and can efficiently handle current and future requirements.

*1) File Handling and Deployment:* The portal handles the uploading and deployment of website source code in ZIP format. Upon receiving a ZIP file, the portal:

- *Extracts the ZIP File*: The contents of the ZIP file are extracted to a designated directory on the server. This process is managed through backend services that handle file I/O operations efficiently to ensure swift deployment.
- *Updates Server Configuration*: After extraction, the server configuration is updated to include the new website. This involves updating internal mappings and potentially modifying virtual host configurations to ensure that the new site is accessible via its domain.

*2) Site Management Operations:* The portal supports various site management operations through a series of RESTful API endpoints. Key operations include:

- *Site Addition*: The `/admin/addSite` endpoint processes the request to add a new site. It validates input parameters, updates the server configuration, and persists site details in the database. This operation involves creating entries in the SiteRepository and setting appropriate status and configuration parameters.
- *Site Removal*: The `/admin/removeSite` endpoint facilitates the removal of websites. It involves deleting the site's records from the SiteRepository and cleaning up associated resources on the server, such as file directories and configuration entries.
- *Site Management*: Through the admin portal, administrators can also start or stop websites. This functionality is critical for maintaining server performance and managing application availability.

*3) Integration with the Web Server:* The admin portal integrates seamlessly with the web server through a set of well-defined RESTful APIs. These APIs serve as the communication bridge between the frontend and backend of the portal and server respectively:

- *API Endpoints*: The portal exposes endpoints for adding, removing, and managing sites. These endpoints interact with the data layer of the server, executing operations such as file uploads, site configuration updates, and status checks.
- *Database Interactions*: The portal interacts with the SiteRepository, UserRepository, and StatusRepository to manage site data, user information, and status records. This interaction ensures that changes made through the portal are accurately reflected in the state of the server.

## V. Evaluation

Evaluation of the implemented web-server involves different phases, each focusing on different aspects of the server's functionality. The different phases are namely:

i. Unit Testing
ii. Integration Testing
iii. Functionality Testing
iv. Performance Testing
v. User Acceptance Testing
vi. Regression Testing.

### A. Unit testing

Unit testing focuses on testing individual components, methods that parses HTTP headers, Database update operations, HTTP request methods with different scenarios. The unit tests are added to every class to ensure the basic expectations of the classes are met, which are executed during packaging the source code each time.

### B. Integration testing

This phase involves examining the interactions between different units, such as those between the server and the database. The testing process includes verifying that a request to fetch data from the database is correctly processed by the server. In this context, it ensures that the correct path of the website files requested by the end user is retrieved and returned in the HTTP response.

### C. Functionality testing

Functionality testing assesses whether the web server correctly implements the core HTTP methods (`GET`, `POST`, `PUT`, `DELETE`) and responds appropriately to client requests. This is the main phase of the evaluation as it involves the webserver's ability to handle various request types, it complies with both HTTP and HTTPS protocols. To evaluate the functionality of the project, two web applications were implemented that were hosted on the webserver. The frontend of the web applications is designed to provide a responsive and interactive interface for interacting with the backend server. The web applications developed were a Scrum Board and a Chat Application, both built with HTML, CSS, JavaScript, and Bootstrap.

*1) Scrum Board Application :* The Scrum Board interface was divided into columns representing task statuses: Backlog, In Progress, and Done. Users could add, edit, or delete tasks, which were then dynamically updated in the respective columns. The frontend interacted with the backend by fetching tasks via the `GET /data/tasks` endpoint and posting new tasks via the `POST /data/tasks` endpoint. Task updates and deletions were managed using the `PUT /data/tasks/id` and `DELETE /data/tasks/id` endpoints, respectively.

*2) Chat Application:* The Chat Application allowed users to send, edit, and delete messages in a chat interface. Messages were displayed in a chat box that automatically scrolled to show the latest messages. The frontend fetched messages using the `GET /data/messages` endpoint and sent new messages using the `POST /data/messages` endpoint. Messages were updated and deleted through the `PUT /data/messages/id` and `DELETE /data/messages/id` endpoints.

### D. Performance evaluation

This involves load testing to determine how the server responds to high traffic, long-duration requests, and simultaneous connections. By focusing on comprehensive performance testing and addressing key technical challenges, the goal was to ensure that the server could handle a high volume of concurrent requests without compromising on speed, security, or stability. The testing methodology incorporated industry-standard tools, including Apache JMeter and Apache Bench, to simulate varying traffic levels and thoroughly evaluate the server's capabilities under different load conditions.

*1) Testing Methodology:* To evaluate the performance, reliability, and scalability of the web server, two industry-standard performance testing tools were employed: Apache JMeter and Apache Bench. Both tools were selected for their ability to simulate real-world traffic and stress-test the server under various conditions.

- Apache JMeter is a versatile load-testing tool used to analyse and measure the performance of a variety of services, with a focus on web applications. JMeter allows for the simulation of numerous concurrent users or threads, making it ideal for assessing the server's handling of multiple, simultaneous requests.
- Apache Bench (ab) is a lightweight command-line tool primarily used for benchmarking HTTP servers. It focuses on testing the server's ability to handle multiple requests and providing a quick overview of the server's performance under load. Apache Bench is particularly useful for measuring throughput and response times, making it a valuable tool for quick assessments.

### E. Matrix

1) *Apache Bench*:
   - Number of Requests: 1000
   - Concurrency Level: 50
   - Total Requests: 1000 requests
2) *JMeter*:
   - Number of Threads: 100
   - Loop Count: 10
   - Total Requests: 100 threads * 10 loops = 1000 requests

TABLE I
PERFORMANCE COMPARISON OF APACHE BENCH AND JMETER

| Metrics | Apache Bench (ab) | JMeter |
|---|---|---|
| Total Number of Requests | 1000 | 1500 |
| Concurrency level | 50 | N/A |
| Total Time Taken | 3.296/sec | N/A |
| Requests/Throughput | 303.36/sec | 8.7/sec |
| Average Response Time | 164.822 ms | 33ms |
| Median Response Time | 55 ms | 19 ms |
| Min Reponse Time | 6 ms | 10 ms |
| Max Response Time | 143 ms | 457 ms |
| 90% Line Response Time | N/A | 35 ms |
| 95% Line Reponse Time | N/A | 75 ms |
| 99% Line Response Time | N/A | 376 ms |
| Trasfer Rate | 848.16 Kb/sec | 24.19 KB/sec |
| Error Rate | 0% | 0% |

- *Apache Bench* demonstrated a higher throughput of 303.36 requests per second, with no failed requests. This displays strong performance under high concurrency with

50 concurrent users, showing the server's ability to handle a significant load efficiently without compromising reliability.

- *JMeter* provided a more detailed breakdown of response times and error rates, with an average response time of 33 ms and a maximum response time of 457 ms. The error rate was 0.00%, indicating there were no errors during the test. This suggests that the server was stable under the tested conditions, although the response times varied a bit under load, which is typical in real-world scenarios. (e.g., 99% of requests were completed in 376 ms).
- *Number of Threads:* This represents the number of virtual users or concurrent threads executing the test. Each thread simulates one user making requests to the server. In the provided test, the Number of Threads is 100, meaning 100 virtual users were used to perform the test simultaneously.
- *Loop Count:* This indicates how many times each thread (or virtual user) will repeat the test scenario. In the provided test, the Loop Count is 15, meaning each of the 100 threads will repeat the test scenario 15 times.
- *Concurrency Level:* It is the number of multiple requests sent simultaneously during the benchmark. In the Apache Bench test, the concurrency level was 50, meaning 50 requests were processed simultaneously by the server.
- *Aggregate Report* from JMeter provided detailed metrics for each request, including total samples (1500), average response time (33 ms), median (19 ms), and percentiles (90% of requests completed in 35 ms, 99% in 376 ms). This report is crucial for analyzing the distribution of response times and identifying performance bottlenecks at different levels.
- *Summary Report* offers a more compact overview of performance metrics, focusing mainly on the basics like the number of samples, average, min/max response times, error rate, and throughput. It is designed for a quick glance at overall performance without the detailed breakdown of percentiles.

These benchmarking tests reveal how the server performs under load, with Apache Bench showing higher throughput without any failed requests, demonstrating the server's capacity to handle a large volume of requests efficiently. Meanwhile, JMeter's detailed breakdown of response times shows some variability but no errors, indicating stable performance under load. This type of benchmarking is essential for identifying performance bottlenecks, ensuring that the server can handle the expected user load, and diagnosing issues like slow response times or high variability in response time that could affect user experience in a production environment.

### *F. User Acceptance Testing (UAT)*

This involves the final phase of evaluation, where the web server is evaluated from an end-user perspective. The goal of this phase is to ensure that the web server meets the business requirements and the objectives. This focuses on the real-world scenarios and use-cases, typically performed by the end-users or the stakeholders involved in the project and validate if the server behaves in an expected way in a live environment.

The UAT with respect to the web server involves the smooth functioning of HTTP methods as requested by the end-user. The end-user uses the scrum board and the chat applications to `POST`, `FETCH`, `EDIT` and `DELETE` data as per the requirement.

This phase of evaluation also involves the testing of admin portal, where the IT admin / product admin is given a provision to host new web applications and manage them. The admin portal functionality is verified by hosting the additional web applications and accessing them over the internet. The status of the current hosted websites is also toggled to check if the START/STOP operations offered on the admin portal functions as expected.

### *G. Regression Testing*

This involves re-running of the previously conducted tests to verify that the web server still works as expected after implementing updates or bug fixes. This ensures that the new code changes do not introduce any regression to the existing functionalities of the web server.

The performance testing and database testing was carried out to determine the scalability of the infrastructure. The database was tested for its ability to scale, especially under large read/write operations, to ensure that it could support increasing data volumes and concurrent transactions.

In order to determine that the solution meets user needs, a regular user feedback sessions were held with the members of the team and the Microsoft team, to obtain feedback on the webserver project's progress and functionality. The input was used to make the solution better. The scrum board and chat applications were tested with users to ensure that the webserver met the requirements. The use cases helped in validating workflows, data handling and response times.

The evaluation part of the project has revealed several key findings about the implementation, primarily related to the performance efficiency, scalability and the end user experience. The webserver performance was optimal under normal to peak loads, with quick response times. However, the performance testing highlighted potential bottlenecks under extreme conditions, primarily in database access and network latency, which enabled to improve the current approach. The webserver infrastructure was found to be scalable, with effective increase in the processing power and memory of the webserver host machine on the Azure cloud platform. It was also noted that further optimization of database queries could potentially improve scalability under high loads. Users and stakeholders feedback was largely positive, with most reporting that the webserver met the requirements, however additional features were requested such as support for both HTTP and HTTPS and, admin portal for web applications management, which were later implemented.

The approach taken for evaluating the web server is considered effective due to the factors: Comprehensive Testing – a wide range of testing methodologies was employed, covering

unit to regression testing, which ensured that the webserver was robust. User-centric development – involving with the users and stakeholders regularly ensured that the solution aligned with the requirements and user needs. Proactive Issue Resolution – Issues identified during testing were addressed with priority and the webserver's design was refined iteratively to enhance performance and reliability. Adapting best practices – The approach was guided by industry best practices in software development and evaluation, ensuring that the solution meets the standards of quality.

## VI. Conclusion and Future work

This project successfully developed a scalable, efficient, and reliable web server capable of handling multiple domains and high concurrent traffic. The server supports HTTP/1.1 methods that ensure compliance with modern web standards. Key successes include a non-blocking architecture using Reactor, Java NIO, integration with MySQL for data management, and a user-friendly admin portal for website management.

The server demonstrated good performance under load testing and handling a high number of concurrent requests while maintaining response times. Future work will focus on enhancing the dynamic data storage system to allow clients to define table schemas, improving data normalization and integrity. Also exploring caching mechanisms and load balancing methods could further optimize performance and scalability.

The project's success is evident in its ability to meet the technical expectations of a modern web application while also providing a scalable solution for managing multiple domains.

### A. Key Findings

- Scalability: The non-blocking architecture and containerization with Docker enabled the server to handle high traffic loads efficiently.
- Reliability: The server showed consistent performance and minimal downtime during testing.
- Security: The integration of HTTPS ensures secure communication between clients and the server.
- User-Friendliness: The admin portal provides a user-friendly interface for managing hosted websites.

### B. Limitations and Future Extensions

- Dynamic Data Storage: The current system does not allow clients to define table schemas, limiting data management flexibility.
- Caching: Implementing caching mechanisms could further improve performance by reducing database load.
- Load Balancing: Exploring load balancing strategies could distribute traffic across multiple servers for better scalability.

Overall, this project has laid a strong foundation for a highly effective web server solution. Further development and optimization will enhance its capabilities and expand its applicability to a wider range of use cases.

### C. Key Successes

- Scalable Architecture: The non-blocking design and containerization effectively handle high traffic loads.
- Robust Performance: The server demonstrated consistent performance under stress testing.
- User-Centric Design: The admin portal simplifies website management for administrators.
- Compliance with Standards: Adherence to HTTP/1.1 ensures compatibility with modern web applications.

## References


[1] R. Fielding et al., "Architectural Styles and the Design of Network-based Software Architectures," *Doctoral Dissertation*, University of California, Irvine, 2000.

[2] V. Deora, S. Gadepalli, and A. N. Singh, "Reactive Programming in Distributed Systems: A Performance Evaluation," *International Journal of Distributed Computing Systems*, vol. 16, no. 2, pp. 112-124, 2021.

[3] H. Zhang, X. Li, and J. Chen, "Performance Comparison of Event-Driven and Thread-Driven Web Servers," *Journal of Internet Services and Applications*, vol. 8, no. 3, pp. 89-103, 2021.

[4] J. Liu and Q. He, "Concurrency Management in Web Servers: A Comparative Study of Thread Pools and Non-blocking I/O," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 7, pp. 1458-1472, 2018.

[5] A. K. Singh and M. Kumar, "A Study on Performance Analysis of Non-blocking I/O Servers," *Journal of Internet Services and Applications*, vol. 7, no. 8, pp. 1-10, 2016.

[6] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81-84, 2014.

[7] E. Rescorla, "SSL and TLS: Designing and Building Secure Systems," *Addison-Wesley Professional*, 2000.

[8] N. Rajasekaran and A. Kumar, "Optimizing SSL/TLS for Web Servers: Techniques and Best Practices," *Journal of Web Engineering*, vol. 12, no. 4, pp. 231-245, 2021.

[9] J. Wang and Z. Li, "Database Scalability in Multi-Domain Web Servers," *Proceedings of the IEEE Database Systems Conference*, 2019, pp. 112-121.

[10] J. Dean and L. A. Barroso, "The Tail at Scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74-80, 2013.

[11] M. Armbrust et al., "A View of Cloud Computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50-58, 2010.

[12] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes: Lessons Learned from Three Container-management Systems over a Decade," *ACM Queue*, vol. 14, no. 1, pp. 70-93, 2016.

[13] L. Richardson and S. Ruby, *RESTful Web Services*, O'Reilly Media, 2007.

[14] L. Jones, D. Smith, and R. Brown, "Asynchronous Communication in Web Applications: Techniques and Tools," *Journal of Web Engineering*, vol. 13, no. 2, pp. 91-103, 2021.