Niels Roefs

Abstract

A research document answering the research questions for the MyWatchList project.

Research Document

Individual Project

Contents

[Problem Definition 2](#_Toc169116812)

[Main Question 2](#_Toc169116813)

[Sub Questions & Methods 2](#_Toc169116814)

[Results 2](#_Toc169116815)

[Conclusion 2](#_Toc169116816)

[Bibliography 2](#_Toc169116817)

# Problem Definition

MyWatchList should be able to handle a million concurrent users. There are various aspects that need to be addressed to allow this to happen. Some of these aspects are scalability and performance optimization. In this research plan, I want to learn more about how to make MyWatchList scalable and how to keep high performance.

# Main Question

How can MyWatchList be effectively prepared for deployment while considering diverse factors such as scalability, and performance optimization?

# Sub Questions & Methods

Scalability:

1. What are the key principles and strategies for designing a scalable architecture and how can MyWatchList benefit from these?

Design pattern research (Library)

IT architecture sketching (Workshop)

1. What are the best practices for load balancing and auto-scaling configurations?

Best good and bad practices (Library)

Non-functional Test (Lab)

Benchmark test (Showroom)

1. How can cloud-based infrastructure contribute to the scalability of MyWatchList?

Available product analysis (Library)

Multi-criteria decision making (Workshop)

Performance:

1. What are the common performance bottlenecks in web applications and how can they be mitigated in the MyWatchList application?

Community research (Library)

Problem analysis (Field)

1. How can caching mechanisms and CDNs enhance MyWatchList’s performance?

Component testing (Lab)

Literature study (Library)

Prototyping (Workshop)

# Results

## What are the key principles and strategies for designing a scalable architecture and how can MyWatchList benefit from these?

**Design Pattern Research (Library)**

After doing research, I found the following key principles and strategies:

1. **Microservices Architecture**:
   * **Principle**: This involves breaking down the application into smaller, loosely coupled services that can be developed, deployed, and scaled independently. Each service encapsulates a specific business function and communicates with other services through well-defined APIs. (Lewis & Fowler, 2014)
   * **Benefits**:
     + **Independent Scaling**: Each microservice can be scaled independently according to its load and resource requirements. This is particularly useful for services that experience variable loads.
     + **Fault Isolation**: Failures in one microservice do not directly impact others, enhancing the overall resilience of the application.
     + **Technology Diversity**: Different microservices can be developed using different programming languages and technologies, allowing the best tool for each job.
   * **Cons**:
     + **Increased Complexity**: Managing a large number of microservices can be complex, requiring robust orchestration and monitoring tools.
     + **Inter-Service Communication Overhead**: Communication between microservices typically occurs over the network, which can introduce latency and increase the risk of partial failures.
2. **Horizontal Scaling**:
   * **Principle**: Horizontal scaling involves adding more instances of a service to handle increased load, as opposed to vertical scaling, which involves adding more resources (CPU, memory) to a single instance. (Digital Ocean, n.d.)
   * **Benefits**:
     + **Cost-Effectiveness**: Commodity hardware can be used to scale out, which is often more cost-effective than scaling up.
     + **Flexibility**: Easily accommodates growth by adding more instances as needed.
     + **Fault Tolerance**: Distributing the load across multiple instances reduces the impact of a single instance failure.
   * **Cons**:
     + **Load Balancing Required**: Effective load balancing strategies are essential to distribute traffic evenly across instances.
     + **Data Consistency Challenges**: Maintaining data consistency across multiple instances can be challenging, particularly for stateful applications.
3. **Asynchronous Communication**:
   * **Principle**: Utilizing asynchronous communication mechanisms, such as message queues, to decouple services and improve performance. This allows services to communicate without waiting for each other to process requests. (System Design by CHK, 2023)
   * **Benefits**:
     + **Improved Performance**: Reduces latency and improves throughput by allowing services to continue processing other tasks while waiting for responses.
     + **Decoupling**: Services are loosely coupled, making it easier to update or replace them without affecting the entire system.
     + **Scalability**: Asynchronous messaging systems can handle a large number of messages, supporting high loads and bursty traffic patterns.
   * **Cons**:
     + **Complex Error Handling**: Managing failures and retries in an asynchronous system can be more complex compared to synchronous systems.
     + **Message Ordering**: Ensuring the correct order of messages can be challenging in a distributed system.
4. **Load Balancing**:
   * **Principle**: Distributing incoming network traffic across multiple servers to ensure no single server becomes a bottleneck. Load balancers can use various algorithms such as round-robin, least connections, or IP hash to distribute the load. (Newman, 2024)
   * **Benefits**:
     + **Enhanced Availability**: Improves system reliability and uptime by ensuring traffic is spread evenly across available servers.
     + **Scalability**: Makes it easy to add or remove servers based on demand without affecting the end-user experience.
     + **Performance Optimization**: Prevents any single server from being overwhelmed, thereby maintaining optimal performance.
   * **Cons**:
     + **Single Point of Failure**: The load balancer itself can become a single point of failure if not properly managed or duplicated.
     + **Configuration Complexity**: Configuring load balancers to handle different types of traffic efficiently can be complex.
5. **Database Sharding**:
   * **Principle**: Partitioning the database into smaller, more manageable pieces (shards), which can be spread across multiple database servers. Each shard contains a subset of the data, and queries are routed to the appropriate shard. (Awati & Denman, n.d.)
   * **Benefits**:
     + **Improved Performance**: Distributes the load across multiple servers, reducing the query load on any single server and improving response times.
     + **Enhanced Scalability**: Allows the database to scale horizontally by adding more shards as data volume grows.
     + **Fault Isolation**: Issues in one shard do not affect the entire database, enhancing overall system resilience.
   * **Cons**:
     + **Complex Query Logic**: Queries that span multiple shards can be complex to implement and may require additional coordination.
     + **Operational Overhead**: Managing multiple database instances and ensuring data consistency can increase operational complexity.

**IT architecture sketching (Workshop)**

See architecture diagrams in the Technical Design Document under ‘Scalable Architectures’.

**Conclusion**

Based on these findings, MyWatchList adopts a microservices architecture to leverage the benefits of independent scaling, and fault isolation. Horizontal scaling is implemented in the Kubernetes cluster to manage increased loads efficiently. Asynchronous communication using RabbitMQ enhances performance and scalability, supporting load balancing as well. Ocelot .NET serves as the API gateway, providing load balancing capabilities. While database sharding is not currently implemented, Azure CosmosDB PostgreSQL supports sharding when it becomes necessary, ensuring future scalability and performance optimization.

# Conclusion

# Bibliography

Awati, R., & Denman, J. (n.d.). *sharding*. Retrieved from TechTarget: https://www.techtarget.com/searchoracle/definition/sharding

Digital Ocean. (n.d.). *Horizontal scaling vs vertical scaling: Choosing your strategy*. Retrieved from Digital Ocean: https://www.digitalocean.com/resources/article/horizontal-scaling-vs-vertical-scaling#horizontal-scaling-vs-vertical-scaling

Lewis, J., & Fowler, M. (2014, March 25). *Microservices*. Retrieved from martinFowler.com: https://martinfowler.com/articles/microservices.html

Newman, D. (2024, January 16). *Load Balancing Fundamentals: How Load Balancers Work*. Retrieved from Akamai: https://www.linode.com/docs/guides/load-balancing-fundamentals/

System Design by CHK. (2023, August 12). *System Design —A Comprehensive Guide on Synchronous & Asynchronous Microservice Communication*. Retrieved from Medium: https://medium.com/@systemdesignbychk/system-design-a-comprehensive-guide-on-synchronous-asynchronous-microservice-communication-8bda324943b8