**Data-Limiting MongoDB Proxy for Multi-tenant Cloud Environment**

**University Name: Unicorn University**

**Name and Surname: Amir Alimov**

**Study Program: Software Development**

**Acknowledgements** First of all, I want to thank my doctoral advisor, **Petr Šimůnek**, for his support, his guidance and many interesting discussions. Through his guidance I learned a lot about databases and his ideas gave me new insights into my work. Finally, he gave me the opportunity to work on this challenging topic in his research group. Furthermore, Last but definitely not least, I want to express my sincere thanks to my parents and my friends for their continuous support, encouragement and patience during my work on this thesis

**Contents**

**1)Background and Motivation  
Problem Statement  
Objectives  
Overview of the existing situation and resources  
2) Cloud Computing and Cloud management  
Service Delivery Models  
Introduction to MongoDB and its Role in Cloud Environments**

**Multi-tenancy in Cloud Computing  
Challenges of Multi-tenancy in MongoDB  
Proxy Servers  
Proxy Server Architecture  
Proxy Server Deployment Models**

**3) Health Information Systems (HIS)  
Functional Requirements for Health Information Systems  
Effectiveness of MongoDB in Various Areas of HIS  
Data Isolation and Security in Healthcare  
Connectivity and Integration with HIS**

**Scalability and Expansion**

**Practical Implementation**

**4)Setting Up MongoDB in a Cloud Environment .   
Data schema design  
Configuring the Data-Limiting Proxy  
Examining Data  
Assessing Data Security and Privacy  
Evaluating Performance and Scalability  
Conclusion**

**1.1Background**

With the advent of the Internet and Internet-connected devices, modern applications can experience very rapid growth of users from all parts of the world. The surge in cloud-based services has underscored the need for efficient and secure data management solutions, particularly in multitenant environments. The utilization of Mongo DB as a NoSQL database in cloud settings has become prevalent due to its scalability and flexibility.

Motivation

Throughout my university education, I encountered database creation, which prompted me to delve more deeply into this field. As I delved into database materials, I questioned how to create a database for Healthcare System then it will be interpreted as HIS with a limited budget without compromising performance.

Practical Implementation of Mongo DB in Health Information Systems

For the practical purpose I plan to find a Health Care Facility in a developing country with recourse-limited settings, in this facility an extract of the patient registry/patient's cases will be used which can vary from 100-200 records of the patients

***Picture 2: Data recourses available and potential solutions for the limited recourses settings***

|  |  |
| --- | --- |
|  |  |

1.2 Problem

In a MongoDB proxy server with a multi-tenant cloud environment, there can be many problems requiring special attention. One of the main issues is the confidentiality and security of data, which is extremely important in the information environment, especially in medicine. It is important to consider patient confidentiality in accordance with GDPR (or HIPAA in the USA). Another problem is data isolation so that patient data is not accessible to others. The server must guarantee strict data isolation. Equally important is the issue of efficient resource allocation to minimize costs, especially in the medical field of developing countries. It is necessary to ensure the effectiveness of this method. Traditional MongoDB deployments lack native support for effective data-limiting mechanisms, leading to potential data breaches, resource contention, and performance degradation.

In multi-tenant cloud environments utilizing MongoDB as the database management system, ensuring data isolation and resource fairness among tenants presents a significant challenge.

Therefore, there is a pressing need for a robust data-limiting MongoDB proxy tailored specifically for multi-tenant cloud environments. Such a solution should seamlessly integrate with existing MongoDB deployments while providing fine-grained control over tenant data access and resource utilization. This proxy should enforce data quotas, rate limiting, and access controls to ensure fair resource allocation and prevent tenant data breaches.

1.3 The aim of the bachelor thesis

The current state of knowledge about MongoDB proxy servers in cloud environments highlights several key aspects and potential areas for further research. One significant development in this field is the creation of servers that can analyze and transform requests and responses while speaking the MongoDB wire protocol, as seen in projects like Mongo proxy on GitHub. While MongoDB software does not officially support them, the advancements significantly improve MongoDB data management and proxy server capabilities.

Hybrid clouds have witnessed a rise in usage of MongoDB deployment and replication across cloud services, both public and private, taking advantage of both cloud environments and their strengths. This method provides greater control of resources in private clouds and benefits from the scalability and flexibility of public clouds. Tools such as Cluster Control are utilized to orchestrate and manage MongoDB nodes. Hover, latency and security remain crucial considerations when implementing hybrid configurations.

Another interesting development is AWS Lambda, which creates MongoDB connection proxy servers. This method involves establishing the Lambda function as an intermediary for connecting to databases; when a function that needs access to MongoDB requires it, it is done via the proxy feature. This approach has been proven efficient in reducing the number of connections to databases, thereby more efficiently distributing resources. Unfortunately, this is also a source of problems, such as managing zombie connections and dealing with cold starts in functions that could affect performance. Despite these issues, these solutions have been effective in the performance and management of database connections and have led to significant decreases in active connections while possibly delivering greater reliability in application performance.

These trends indicate a growing desire and demand for sophisticated MongoDB proxy services in cloud environments, especially when organizations increasingly embrace the hybrid cloud model. Future research aims to focus on improving these solutions' effectiveness and security, tackling latency issues, and examining new ways to manage database connections in cloud-based environments in the development of Health Information Systems (HIS) in recourse-limited settings.

1.3 Hypothesis

The objective is to investigate and study the effectiveness and role of data-straining MongoDB proxy servers to optimize clouds with multiple security, performance, and efficiency tenants for HIS.

This study aims to specify the potential areas MongoDB to be used in HIS, to fill any inconsistencies in understanding how MongoDB proxy servers may be used in HIS to control data flow effectively, ensure fair resource allocation, and enhance cloud computing security, and efficacy.

Hypothesis -The implementation of the MongoDB proxy server will dramatically improve the quality and control of data flow, security, and allocation of resources in a multi-tenant cloud. Thus, can boost HIS management in resource-limited settings and bring added value to Health Systems.//

My study aims to empirically prove the hypothesis by studying potential areas of use MongoDB in HIS though the criteria of standards of security enhancements, performance, and the efficiency of resource allocation for MongoDB proxy servers that operate on multi-tenant clouds.

Objectives

* What are the best practices in HIS for configuring and deploying MongoDB proxy servers in multi-tenant cloud environments? To maximize their benefits, seek to identify effective strategies and configurations for MongoDB proxy servers.
* What is the best way to make MongoDB proxy servers help to use HIS data in multi-tenant environments? This is why studying their role in managing data flow is important.
* What are the HIS implications for the performance of the use of the data-delimiting MongoDB proxy servers within cloud-based environments?
* What solutions can Mongo offer to protect sensitive information related to patients? How can Mongo DB server can comply with the privacy and confidentiality rules and policies?
* What is the status of HIS practices for Data Management and the Challenges of Multi-tenant Cloud Environments Using MongoDB?
* How does a data-limiting MongoDB proxy enhance security and data integrity in a multi-tenant environment for HIS? Aimed at understanding the security benefits and potential vulnerabilities addressed by implementing MongoDB proxy servers.

In my work, I plan to focus on a theoretical comparative analysis of the pros and cons of using MongoDB. This will be done through the literature review, I am especially interested in learning the advantages and disadvantages of Mongo DB use in Health Information Systems, especially in recourse-limited settings. Given the fact that Mongo DB has been widely used through the literature search, it would be possible to evaluate such parameters as:

* Advantages and disadvantages of using Mongo DB;
* What areas of the HIS are most effective to apply Mongo DB;
* Data security and safety for storage and operation;
* Connectivity and integration to the overall HIS;
* Financial implication: maintenance, human resources, operational costs;
* Possibility to scale up and expand to the national/regional/global/international HIS:
* Possibility to analyze, verify, and interpret data for policy making.

This question seeks to understand current methods and issues associated with managing multi-tenant cloud environments with MongoDB in HIS

1.5 Overview of the existing situation

Recent developments in MongoDB proxy servers for health systems focus on several key areas:

Security and Data Isolation Enhanced protection of health data in shared environments with focus on preventing cross-tenant access.

Multi-Tenant Database Management Analysis of approaches for tenant data organization: shared collections, separate collections per tenant, or individual databases, considering scalability and cost-effectiveness.

Hybrid Cloud Architecture Exploration of MongoDB replication between private and public clouds, addressing latency and security concerns in healthcare settings.

Performance and Resource Management Development of monitoring tools for tenant-specific usage and optimization of resource allocation in HIS deployments

2.1Cloud computing and cloud management

Before examining in detail, the role of MongoDB in cloud computing, it is necessary to reveal what cloud computing and how the cloud model is particularly in demand for modern applications or in multi-tenant databases in sensitive environments such as healthcare.

**Definition of Cloud Computing**

Cloud computing provides computing resources (servers, storage, databases, networks, software) these resources can be rented, but an essential requirement is an Internet connection, so companies do not need to launch physical data centres or IT infrastructures. In modern healthcare systems it is a good tool due to its scalability and cost-effectiveness.

**Main Characteristics of Cloud Computing**

There are 5 characteristics described by the National Institute of Standards (NIST) that make cloud computing cloud-based. Without these components, a service cannot be called cloud computing:

* On-demand self-service
* Broad network access
* Resource pooling
* Rapid elasticity
* Measured service

Three of them are particularly important when it comes to administering MongoDB in a shared multi-tenant environment:

* + Resource pooling- The supplier has the following computing power (servers, memory, networks) and is united into one infrastructure called a pool. The pool allows to wisely share resources between users.
  + Another feature is Rapid elasticity, which allocates the necessary resources on demand, instead of unlimited resource allocation optimize and manage resources. The main negative side is that with noisy neighbour problem tenants can consume shared resources (CPU, I/O, network bandwidth) when they need to run resource-intensive queries or access very large data sets, which can dramatically reduce the performance of other tenants. This is not acceptable by time sensitive medical applications.
  + Rapid elasticity – is one of the main features of cloud computing, depending on the demand of the user or company, the allocated
  + resources can either increase or decrease when the need for them disappears. However, another obstacle may arise in the form of a user who makes inefficient queries or queries with unlimited data retrieval; this obstacle can lead to performance and cost problems in the context of a multi-user system. The solution is to introduce query quotas and rate limiting

Measured service – An analysis is carried out of how many resources the user has used; payment is made based on the resources used. This requires micro tenant-specific accounting. Native MongoDB however does not have native support of fine-grained, by tenant per-metering and restriction of data access volume (e.g. documents read, bytes transferred). Proper tracking and imposing are essential to fair billing, avoiding runaway cost (high cost in healthcare budget, and avoiding overruns), and depend on one or more layers of abstraction such as our proxy.

* Self-service and Broad network access in the ondemand forms do not have an immediate relationship to the catalog and the core database governance challenges discussed here, although general benefits can be expected
* <https://www.netsuite.com/portal/resource/articles/erp/benefits-cloud-computing-healthcare.shtml>

**Deployment Models**

The deployment model of cloud computing determines the particular form of cloud environment on the basis of possession, size, and access and nature and purpose of the cloud. A cloud deployment model determines the location of the servers in which you are using, and who controls them. It indicates how your cloud infrastructure should be and what you can alter and whether you will be assigned services or you have to do everything on your own. The cloud deployment types also determine relationships among the infrastructure and your users. The following are the descriptions of different kinds of cloud computing deployment models.

* private cloud – Can be used by a single user or a company, which ensures security and access control
* Public cloud – Provides access to multiple users to one IT provider structure. Unlike a private user, cannot manage the cloud. Only a contractor can manage the cloud.
* hybrid cloud – Combines public and private clouds, bound by technology enabling data and application portability.
* Community cloud: Shared by several organizations with common concerns (e.g., security, compliance)

For multi-tenant SaaS solutions the most relevant are: Public cloud and hybrid cloud for several reasons

Public cloud

* Cost-effectiveness
* Scalability
* Multi-tenant SaaS:

Hybrid cloud

* More security
* Control access to sensitive healthcare data

When choosing between these 2 models, my choice fell on the public cloud

Our system is designed to serve many clinics and hospitals at the same time. The public cloud is ideal for this task, as it allows you to easily add new medical institutions and automatically increase capacity when the load increases. However, it is in such an environment, where many different organizations use common resources, that a critical need for a proxy server for data restriction arises to prevent conflicts between users.

Hybrid cloud in the context of this study, is considered as an additional, rather than the main scenario for the following reasons:

Firstly, the problems of restricting access to data arise only in the public part of the hybrid architecture, where many independent organizations use common resources. The private part does not require the implementation of a proxy server, since the organization has full administrative control over its infrastructure.

Secondly, most medical SaaS platforms are deployed in a public cloud due to cost-effectiveness and ease of scaling. Hybrid solutions are used mainly by large medical institutions with high security requirements and sufficient IT budget.

Thirdly, the developed proxy server solves the same problems in the public part of the hybrid cloud as in a fully public cloud. Thus, the focus of the study on the public cloud automatically covers hybrid scenarios.

https://silstonegroup.com/healthcare-saas/

**Challenges of Cloud Computing**

* Challenges in MongoDB Critical Cloud in a Multi-tenant Healthcare environment When it is promised to actions, cloud computing brings serious problems, especially when working with sensitive data in an open Mongo db environment:
* Performance & Latency: Latency has to be low in data intensive applications. Multi-tenancy makes it worse: in MongoDB, noisy neighbors (large queries, retrieval of big data) result in uncontrollable breakdowns. This is essential when it comes to real time clinical decision support.
* Security & Privacy (Paramount): Security & Privacy of individuals (with safeguarding of sensitivity to confidential PHI entrusted to third-party Error: You reached the limit of free use. Please subscribe at https://www.storkapp.me providers) is the primary concern. In addition to the conventional access control, high chances of inadvertent or intentional exfiltration of large size of data through MongoDB queries prove to be a significant risk in a multi-tenant environment. Rigid adherence (HIPAA, GDPR) requires sound protection.
* Limitations to control: Organisations lose direct control of infrastructure. Cloud provider tools and native MongoDB do not necessarily have granularity necessary to enforce complex per-tenant data consumption policies (e.g. max data retrieved per query/tenant) akin to that required by healthcare data policies and compliance. The data access layer has to be in fine-grained control.
* Bandwidth expenses: It might be very costly to transfer data particularly big sets of results of MongoDB queries. In multi-tenant the uncontrolled data egress within one tenant has a direct effect on the operating expenses. One of the main factors of cost control is the limitation of the quantity of retrieved data.
* Availability & Reliability: The services may go down. Although the usage of infrastructure is under the control of the cloud provider, the unregulated usage of the load to utilize the MongoDB instance in abundance with information heavy tasks can indirectly lead to lapse of services to other parts, breaching healthcare SLAs.

The co-existence of cloud shared model (Resource Pooling), MongoDB performance (and ability to retrieve large amount of data), multi-tenancy requirements, and healthcare security/compliance requirements lead to a vital concern: the inability to implement fine-grained SQL server native enforcement of per-tenant data access volumes in MongoDB on a public cloud. It is our Data-Limiting MongoDB Proxy which specifically fills this gap. It is critical middleware, since it applies stringent per-tenant quotas on data reads (e.g., maximum documents per query, maximum bytes per request) and audits usage, and includes defense of the noisy neighbor effect. The proxy allows implementing secure, efficient and compliant multi-tenant healthcare applications based on MongoDB in the public cloud by directly resolving the core performance, security, cost control and availability issues described above.

2.2 Service delivery models

The relationship between the provider and the consumer in terms of control and responsibility are determined using cloud service models

In this chapter, will be described 3 service models

1. Software-as-a-Service (SaaS) is a way of delivering application as a service over the network/ Internet that users can directly consume without the tension of installing or configuring an application. In traditional computing, consumers had to pay not only the software licensing fee but also spend a large portion of their budget in setting up the infrastructure and platform over which the application would run. SaaS eliminates this problem and promises easier as well as a cheaper way of using application. (Bhowmik, 2017, pp. 106-107) According to Bhowmik (2017), unlike packaged applications that need to be installed on devices, SaaS providers run them in their processing centres. Customers do not need to buy software licenses or any additional computing resources to support the application

**Relevance for Healthcare MongoDB Governance:**

Provider-Managed Database: MongoDB infrastructure is fully under control of SaaS vendors SaaS vendors control everything, which is a critical element of multi-tenancy but leaves a gap in governance.

Basic Multi-tenancy: An independent application/ MongoDB is shared by separated tenants (e.g. clinics, hospitals).

The Proxy Imperative:

Raw MongoDB does not feature fine-grained tools to impose:

Per-tenant query data fetch limitations (e.g. max\_docs=1000 per request)

Auditing of PHI accessed in real-time of volume

Avoidance of the noisy neighbor effects

Here is work our proxy design: It is utilized as middleware between SaaS application and MongoDB cluster implementing the tenant-identified data policies in the transparent manner.

1. IaaS (Infrastructure as a Service):

Infrastructure-as-a-Service delivers virtualized-hardware (not physical, but simulated software) resources to consumers known as virtual resources or virtual components. It provides the facility of remotely using virtual processor, memory, storage and network resources to the consumers (Bhowmik, 2017, pp. 103) As Bhowmik (2017, p. 104) explains, "consumers no longer need to manage or control the underlying computing infrastructure that they consume as IaaS. This type of model enables the clients to manage the operating system, software and no data that has been installed over on virtual servers and handles the physical infrastructure by the provider.

Relevance

The proxy may be utilized internally by healthcare organizations that self-host the multi-tenant apps on IaaS. It is a second possibility.

3. PaaS (Platform as a Service):

Platform-as-a-Service (PaaS): IaaS gives access to virtualized components of infrastructure, but PaaS takes it a step further, giving ready-to-use deployment and development environments.

According to Bhowmik (2017, p. 105), "PaaS facility, on the other hand, relieves users from all these tensions and delivers ready-made platform to consumers via internetwork/Internet." The key advantage is that "PaaS model lets the users focus only on development and deployment of application without having the tension of arranging and managing the underlying hardware and software" (Bhowmik, 2017, p. 106).

The provider in this model deals with infrastructure, operating system, middleware and the runtime environments whereas the client deals with application development and data management

Relevance: The proxy could be brought into use by SaaS vendors constructing healthcare using PaaS (e.g., Heroku, AWS Beanstalk) to get together the isolation of tenants in their MongoDB tier. It is not the main target.

A diagram of a company's pyramid

AI-generated content may be incorrect.

**Key Characteristics for Our Focus**

**Multi-tenancy is Fundamental**

**Provider-Managed Database**

**Tenant's View**

**When analysing all types of clouds, it seemed to me that Sass is an excellent solution for my case and here's why:**

Operative Model of Healthcare Applications: The majority of contemporary EHR, tele demands, Labs are served as SaaS.

Manages Centralized & Scales Easily: Perfect in delivering to many separate healthcare tenants. Invents the Specific Governance Gap: The abstraction that enables SaaS to exist (provider manages DB) is also the very reason that there is a Dragon-slaying hole at the DB layer that our proxy fills: people want visibility and per-tenant control over DB access. When it comes to SaaS, the provider requires using the proxy to configure the shared MongoDB in a way that best suits their tenants as well as keeps it secure.

Relevance for Healthcare MongoDB Governance

**The Reason Why SaaS requires a Data-Limiting Proxy:**

Healthcare-Specific Factors Healthcare: 78 percent of EHR/telemedicine platforms operate using SaaS (Chandrasekaran, 2023). Adverse Governance Gap: Abstracting SaaS applications denies tenants the ability to access the MongoDB directly However, providers do not have built-in functionality on: Cogitation of unreasonable data query by a tenant PHI accessed per tenant (HIPAA 164.312) The imposition of cost-control through data egress limits Proxy as Box-Ticking Device: It is in the database access layer and: Sees money with $limit operator per tenant Produces PHI access log volume logs Blocks performance cascades caused by queries

**3.1 Introduction to MongoDB and its Role in Cloud Environments**

MongoDB is a scalable, flexible [NoSQL](https://www.mongodb.com/nosql-explained)  database platform designed to overcome the relational databases approach and the limitations of other NoSQL solutions. MongoDB is well known for its horizontal scaling and load balancing capabilities, which has given application developers an unprecedented level of flexibility and scalability.

(https://www.mongodb.com/resources/products/capabilities/features)

This makes MongoDB a dominant choice for cloud-based applications, including healthcare SaaS According to information from the Db engine website, MongoDB is in 5th place in the database. (https://db-engines.com/en/)

**Key MongoDB Features:**

**Schema-less Design**

MongoDB is a document-oriented database, which means that data is stored as documents, and documents are grouped in collections. Data is stored in JSON or BSON format, which allows the creation of objects with different sets of fields.

(<https://www.mongodb.com/resources/products/capabilities/features>)

Thanks to this, collections do not necessarily have the same set of fields, this is a feature of MongoDB called flexible schema.

Cloud/multi-tenancy applicability:

This affordability plays an important role in rapidly changing cloud applications and SaaS services. It makes changing data models by adding new features or changing tenant requirements exceptionally cheap without migrations of schemas at all and causing downtime. The different tenants will be able to successfully save different data structures in the same logical database or collection

**Horizontal Scalability (Sharding):**

Sharding refers to the process of splitting data up across machines; the term partition‐ ing is also sometimes used to describe this concept. (Chodorow, 2019, p 289)

(<https://imaster.academy/contenidos-tematicos/desarrolloweb/Unidad2/Material%20de%20apoyo%20(Descargable)/MongoDB%20in%20Action.pdf>)

The underlying explaining postulate associated with sharding is the distribution of resources instead of upgrading of hardware. As the literature explained:

By putting a subset of data on each machine, it becomes possible to store more data and handle more load without requiring larger or more powerful machines—just a larger quantity of less-powerful machines. Sharding may be used for other purposes as well, including placing more frequently accessed data on more performant hardware or splitting a dataset based on geography [Chodorow, p. 289].

Cloud/multi-tenancy applicability:

Horizontal scaling is important in the cloud applications with unpredictable scaling or high number tenants. Generally, cloud providers provide a perfect infrastructure (managed instance groups, container orchestration) to deploy and manage sharded MongoDB instances easily and bidirectionally. Sharding enables multi-tenant applications to scale with enormous data size and high demand by breaking up the load across commodity machines.

**Specific Workload Performance:**

MongoDB provides a high-performance workload on:

Semi-structured or unstructured: Its document paradigm is more efficient on storing and querying JSON-like data that appears in web APIs, Internet-of-Things and real-time analytics.

(https://www.mongodb.com/resources/basics/unstructured-data/storage)

Read-intensive workloads: Indexing (with compound index, geospatial, and text indexes), covered queries and in-memory storage engines (Wired Tiger) provide boosted read performance.

Writing scalability: The ability to handle large amounts of writing is achieved with Sharding and efficient writes.

Cloud/Multi-tenancy Relevance: MongoDB has suitability as an application in a variety of cloud applications including content management and catalogues, user profiles and operational logging, and multi-tenant applications. It is one of its dealings with large masses of semi-structured data very effectively.

Coupled Features & Dev Experience:

The MongoDB contains a rich query language and supports complex queries, aggregations (powerful pipeline framework to analyse and transform data), geospatial search and full-text entries.

It has capabilities such as Change Streams (updating information about changes to data in real-time) and Time Series collections (data collections that optimize time-based information).

Cloud/ Multi-tenancy Relevance: These built-in functions diminish the use of other dedicated databases or intricate middleware, which makes the cloud application less intricate. The casual document model and query language makes developer productivity by reducing development and iteration cycles inherent in cloud-native applications. They further facilitate management services such as MongoDB Atlas that facilitate easier deployment and running of services in the cloud

The set of strengths leads to MongoDB being an attractive solution to develop cloud-native applications, particularly Platform-as-a-service (PaaS) and Software-as-a-service (SaaS) products. It has scalability, to support changing load of multi- tenant applications and a flexible schema that supports the diverse data requirements of different tenants, and its performance profile meets the needs of broad classes of cloud uses. Managed MongoDB services (examples: MongoDB Atlas, Azure Cosmos DB for MongoDB, AWS Document DB) obviate most of the operational bowing to developers so that they can concentrate on application logic and make use of cloud elasticity and resilience.

Although MongoDB supports solid functionalities in the construction of scalable cloud applications, implementation of data isolation, security, and performance aspects between individual tenants, while using the same database infrastructure has certain challenges that require architectural attention.

**Multi-tenancy in Cloud Computing**

**Introduction to Multitenancy**

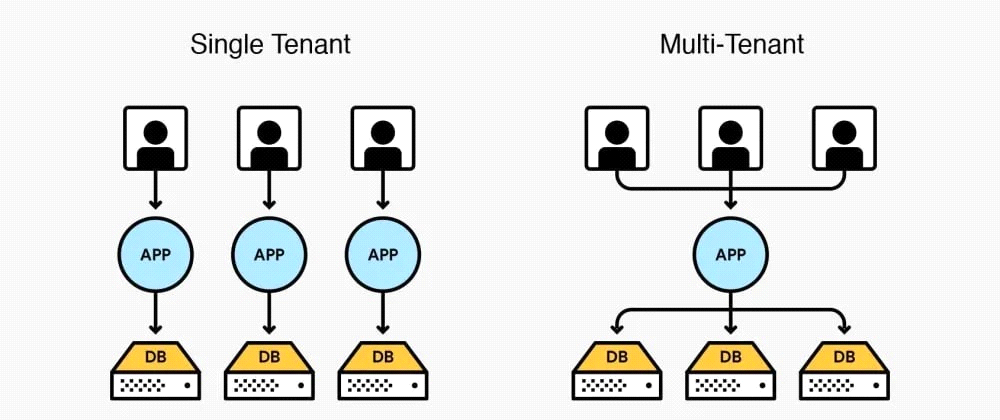
Multi-tenant architectures represent a fundamental shift in delivering cloud services, enabling multiple customers to share the same infrastructure, applications, and databases while maintaining complete data isolation [(2025, Rishi Kumar Sharma, p 3)](#Multitenantdef)

In other words, multitenant is an application that is used by many users who use one resource, and each of them has the same copy of the application. The basic idea of ​​multitenancy can be described as something like this: A typical application is a cottage designed to accommodate one family, which uses its infrastructure (walls, roof, water supply, heating, etc.). A multitenancy application is an apartment building. In it, each family uses the same infrastructure, but the infrastructure itself is implemented for the entire house. It is important to note that multitenant also uses Rapid elasticity and Resource pooling, which guarantees greater cost efficiency when using multitenant systems. As an example from real life, Gmail users use the same application, while each of them has their own mailbox.

In contrast to multitenancy, there is a single tenancy in which one instance of the application is used by only one client, which leads to an isolated environment that ensures high data security; however, the main disadvantage in this environment is the high cost

The concept of the single-tenant and multi-tenant is demonstrated in the picture below.

Picture 1: Difference between single-tenant and multi-tenant.



* https://www.gooddata.com/blog/multi-tenant-architecture/

**Database Multi-Tenancy Models**

Separate Database

Each user has their own version of the data base or cluster. This approach is effective in terms of data security but requires significant financial investment.

Isolation: High,

Since each user has their own database, thus, customer data is completely separated from each other, both at the hardware and software level. If 1 client has a problem in the database, this does not affect the data base of other users, and in this option, the problem of a noisy neighbour does not occur

Advantages

* Security and isolation
* Easy management for each tenant
* Simple to identify and fix problems

Disadvantages

* Highest resource costs
* Poor resource utilization
* Difficulty scaling

Shared Database/Separate Collections

Unlike the first option, in this model users use the same data base but have different collections or Schemas (Relational Databases)

Isolation: Medium

Data isolation occurs at the collection level using authorization and authentication with the RBAC method, but the key drawback is the emergence of the noisy neighbor problem when one user can provoke a high load on the data base, which can affect the performance of other users of the same data base

Pros and cons

* Resource utilization
* logical segregation
* Easy to add/remove tenants

Cons

* "Noisy neighbour" risk
* Limited schema/collection customization
* DBMS dependency

Shared Database/ Shared Collection

In this model, users use a common database and collection.

Isolation

Low

Since user data is in the same collection, there is a risk of mixing this data. However, the solution in this case is isolation based on the Tenant\_id identifier. Based on this identifier, data is filtered. In other words, when making a request, the user enters his Tenant\_id and only then gets access to his data.

! A bug in the application can lead to a high probability of data leakage, another problem is a high predisposition to the noisy neighbour problem since all actions occur in the same collection

Pros and cons

* Maximum resource efficiency
* Lowest cost
* Optimal for Sharding

**Cons:**

* Highest data leakage risk
* Severe "noisy neighbour" vulnerability
* Minimal per-tenant customization

***(https://daily.dev/blog/multi-tenant-database-design-patterns-2024#:~:text=Database%20Design%20Basics-,Key%20Concepts,duplicate%20systems%20and%20lowering%20costs.)***

The Core Trade-off: When choosing a model, it is necessary to place one of the three primary areas of concern issues on the front burner of the concerns: isolation/security (Separate DBs), resource efficiency/scalability (Shared Schema/Collection), or a better integrate the two (Separate Schemas/Collections). Such choice has a big, critical impact on security, performance, cost, and complexity of operations.

MongoDB accommodates all the three models even though robust multi-tenancy especially the effective Shared Collection method poses special challenges in its designs as highlighted in the following section.

Challenges of Multi-tenancy in MongoDB

Challenges of multi-tenancy in MongoDB encompass various aspects related to data isolation, performance, scalability, and security within a shared environment. Here are some key challenges:

Data Isolation Complexity in Shared Collections

Core Problem:

Since users share the same collection, they may encounter the fact that their data may be mixed up or, even worse, data may be lost. In the previous chapter, we discussed that the solution to this problem is to introduce an tenant\_id identifier, Reliance on tenant\_id filtering at the application level makes the system fragile and leads to numerous security vulnerabilities. If there is a bug in the code, clients see someone else's data.

// Unsafe query (without tenant\_id)

db.patients.find({diagnosis: "diabetes"})

// Safe query (with filter)

db.patients.find({

tenant\_id: "clinic\_123",

diagnosis: "diabetes"

})

If you send a query without tenant id, the system will return us a list of people from all clinics with a diagnosis of diabetes, not just from a specific clinic to which the user has access. In this case, the user will have access to data from other clinics, which is a serious violation of privacy (HIPPA / GDPR), and there may also be a performance problem since the system will scan the entire collection. And the absence of a filter will lead to the processing of many records

https://borabastab.medium.com/six-shades-of-multi-tenant-mayhem-the-invisible-vulnerabilities-hiding-in-plain-sight-182e9ad538b5

"Noisy Neighbor" Problem

“Noisy neighbour problem”:  This issue arises when one user consumes excessive resources, leading to service throttling that negatively impacts the performance of others and reduces their productivity.

* System Outages: A system outage can impact all tenants simultaneously in a multi-tenant application. Performance Multi-user applications depend on the provider in case the program has a technical failure it may affect the work of users of applications in terms of maintenance and updates. This problem may occur when updating the application, a detailed example will be given below

Example: AWS EC2 Case 2011

* History: At 12:47 AM PDT on April 21st, a network change was performed as part of our normal AWS scaling activities in a single Availability Zone in the US East Region. The configuration change was to upgrade the capacity of the primary network. During the change, one of the standard steps is to shift traffic off of one of the redundant routers in the primary EBS network to allow the upgrade to happen. The traffic shift was executed incorrectly and rather than routing the traffic to the other router on the primary network, the traffic was routed onto the lower capacity redundant EBS network.

<https://aws.amazon.com/message/65648/>

* The solution in MongoDB in such cases is replication

and the Automatic Failover function together they create a duet for converting System Outage. In the case of replication, the database creates copies of the primary database and its data. Now when the main database is unavailable, its backup copies are activated.

A diagram of a diagram

Description automatically generated

https://www.mongodb.com/docs/manual/replication/

* If we consider Automatic Failover, in the presence of problems on the main database, elections are held between the backup servers. The elections decide which server will become the main one to replace the previous one. Thanks to replication, the new main database will remain as they were copied from the previous one.A diagram of a primary

  Description automatically generated
* https://www.mongodb.com/docs/manual/replication/

**Proxy Servers**

Database proxies have become an integral part of modern cloud environments due to fundamental changes in application architecture.

A database proxy is like a secretary in any busy corporate office. Just as a secretary controls visitor access to management by filtering requests, scheduling visits so that the manager is not overloaded, and maintaining visitor logs, a database proxy controls application connections to the database, filters requests according to access controls, maximizes connection utilization, and provides comprehensive audit data.

Proxy servers are mediators within other IT systems. Proxy servers serve as a so-called gatekeeper of the Internet by ranking between a user and the desired websites they visit granting a great expansion of benefits including better security by concealing IP address of users and blocking malicious traffic, better privacy because it ensures anonymity, access to inaccessible resources, control over the use of Internet in corporate usage and data caching to allow faster query responds [1]. Being applied to databases, these basic postulates of proxy servers obtain a very specific character and exclusive importance in the light of connection management that guarantee the safety of data and the most reasonable performance in multi-tenant clouds.

**A screen shot of a computer

AI-generated content may be incorrect.**

**Key Roles and Functions of Proxy Servers**

**Intermediation**

Basic Operation: The client does not directly address the target server, but instead to the proxy. The proxy connects on its own with the target server, forwards the request of the client, takes the response and sends back the response to the client.

Consequence: The target server will view the request as sent by the proxy server, rather than by the true client, which grants source anonymity (but is not always full).

Access Control & Filtering

The proxy is able to validate the users (verifying logins/ passwords, certificates).

The proxy permits or inhibits access to certain resources (websites, content types, IP addresses, ports) based on specified policies (rules).

Examples: The use of corporate proxies to block social media; parental controls; limiting access to internal company resources to the outside world.

Caching

A proxy has the capability of storing local copies (cache) of very popular resources (web pages, pictures, files).

Advantages:

Quick Access: The client[s] are offered faster access to the data in the proxy cache (which is nearer and more rapid) rather than waiting on the remote server.

Reduced Load: This helps to reduce the amount of traffic that is directed to the target server and to the internet connection.

Bandwidth Savings.

Load Balancing

A proxy (particularly a Reverse Proxy) may spread out the requests received by a number of different clients across a cluster (or farm) of servers (so-called backend servers).

Advantages:

Increased Performance: Assures even distribution of work.

Scalability: It allows the addition and removal of servers easily in the cluster.

Fault Tolerance: In case a single server goes down, the proxy will send its requests to the servers that are operating.

Single Entry Point: Clients deal with one proxy address and they have no knowledge of the internal layout of server cluster.

Security

Backend Protection: A Reverse Proxy will not allow external access to the internal network structure and servers, rather a proxy that provides a buffer or facade.

Attack Filtering: It is able to prevent known attacks (e.g., DDoS), or suspicious traffic at the proxy level before reaching vulnerable servers.

Encryption/Decryption: It is capable of doing SSL/TLS (finishing HTTPS connections on its own), thus removing this load on the backend servers. It is also capable of establishing encrypted connections with the backend.

Traffic Inspection: Content inspection of request/ response to identify threats (virus, vulnerability) or security policy violations.

Logging & Auditing

A proxy has the ability to keep a detailed log of every request and response that is going through it (sources, destinations, time, data volume, actions).

This is essential in tracking down activities, troubleshooting, traffic inspection and compliance with regulatory standards.

General Operating Principles

Transparency: The client can or can not be aware of the existence of the proxy. Proxies in corporate networks tend to be dynamic, and a user is not aware of their existence.

Configuration: The configuration of the proxy is defined by its configuration files or options (access, caching, load balancing, security).

Proxy Types:

Forward Proxy: It is a client-side (or local network) proxy. Intended to allow clients to access the public internet (and have control, caching, anonymity).

Reverse Proxy: It is in front of the servers (the backend). It serves as the face of the service to external clients, and deals with balancing and security and also with caching of the service at rest. The client normally does not have a clue that they are communicating to a proxy and not the backend itself.

Protocols: Proxies may support a number of network protocols, the most widespread of which are HTTP/HTTPS, though SOCKS (any traffic) and FTP etc.

Summary:

The proxy server is a highly flexible and potent tool of IT infrastructure that is used in a wide range of essential functions: not only to intermediation and acceleration via caching, but to provide security, scalability due to load balancing, and powerful control over network activity. It has been positioned as a very important part of corporate network and big internet services because of its flexibility.

**Proxy server Architecture**

A Proxy has an architecture that incorporates important elements that guarantee traffic routing, security, and optimization. We shall look at the usual elements and how they interact.

Key Components

Listener

Role: Accepts inbound connections on a certain port (e.g. 80 on port 80 in case of HTTP).

Mechanism: It is a network daemon, which waits to be connected to. Proxies a variety of protocols (HTTP/S, SOCKS).

Example: Nginx/Apache on port 8080.

Handler

Role: Interprets the request, and enforces rules (filtering, authentication, caching).

Actions:

Verifies URL/Headers with policies.

Makes the decision of either forwarding the request or to honor the request out of the cache.

Takes care of the termination of the SSL connection (when the proxy is used in HTTPS).

Backend Connection Pool

Role: This role is in charge of managing connections to the target servers to reduce latency.

Optimization:

Recycles relationships rather than creating new relationships.

Both to defend the backend and to limit the number of simultaneous connections.

Load balances the servers.

Logging

Role: Documents Actions to audit and diagnostics.

Data:

IP, URL of client, status of response (e.g. 200 OK), processing time.

Errors (e.g., 502 Bad Gateway).

Tools: Syslog, Elastic, log files, which are rotated.

Request/Response Flow: Step-by-step Diagram.

Step 1: Client - Listener

The client requests the proxy (e.g. http:// proxy:8080).

The Listener takes up the connection and forwards it to the Handler.

Step 2: Handler - Processing

The request is inspected by the Handler:

In case the resource is already in the cache it will send the response directly to the client.

Otherwise, it retrieves a connection out of Backend Connection Pool (or else creates a new connection).

It implements regulations (prohibiting prohibited websites, anonymization of IP).

Step 3: Proxy - Backend

That request is passed through the connection pool to the target server (e.g., https://example.com).

The Pool makes the best of the connection: it utilizes an existing connection or opens one with timeouts (e.g. 30 sec).

Step 4: Backend - Proxy

Response of the server comes to the connection pool.

The response is processed by the Handler: the data are compressed, cookies are added (e.g. X-Proxy-Server: MyProxy) and the content is saved.

Step 5: Proxy - Client

The honed reaction is returned to the client.

Logging records the details:

2025-07-26 12:30:15 client 192.168.1.10 GET index.html Backend 93.184.216.34 200 45 ms latency

Summary of Benefits

Flexibility: The components enable the proxy to be customized to meet a certain task (caching, security).

Performance: The connection pool lessens the load on the backend and the cache accelerates the responses.

Reliability: Monitoring and logs can find failures (e.g. backend unavailability) easily.

The basis of the modern solutions is the Proxy Architecture (API Gateways, CDNs, and security systems, e.g., Cloudflare, etc.).

Health information System

Health information system – is a set of components that significantly facilitate the work of medical institutions and medical personnel. As noted in the literature, “Health information systems support health professionals working in medical institutions, as well as healthy or sick people in their various life situations” (Winter et al., 2010).

Currently, most countries use this system in medicine, as it provides access to data access functions, which significantly improves coordination between different institutions. The main goal of these systems is the integrity of information, security and privacy of patient data, such as electronic medical records (EMRs) and electronic health records (EHRs), as well as simplifying the work of medical personnel.

Как справедливо отмечает автор(Francesc Mateu Amengual) “ Interoperability in healthcare isn’t just a buzzword; it’s a fundamental necessity. It refers to the ability of IT systems to enable the timely and secure access, integration, and use of electronic health data.” https://www.mongodb.com/blog/post/dual-journey-healthcare-interoperability-modernization

Considering that HIS system is multi-tenant, it is necessary to have reliable protection, privacy and proper data isolation.

The Healthcare sector is an ideal example to demonstrate the use of proxy servers with limited access to data in multi-tenant cloud environments. According to the study taken from “The Healthcare Information Systems Market Size is Projected to Reach $261985 Million by 2030 at a CAGR of 7.60%” during the forecast period 2024-2030.IndustryARC. (2024).

Healthcare Information Systems Market Forecast (2024–2030). Retrieved from [https://www.industryarc.com/Report/4339/Healthcare-Information-Systems-Market-Research-Report.html].

<https://www.ibm.com/docs/hu/stea/9.0?topic=itxa-tenants-multitenancy>

10)<https://www.selecthub.com/medical-software/practice-management/7-categories-healthcare-information-technology/>)

**Functional Requirements for Health Information Systems**

It is critical for healthcare that information systems (HIS) meet stringent functional requirements such as security, privacy and data accessibility also in the opinion of (Francesc Mateu Amengual) “One of the biggest IT challenges faced by healthcare organizations is sharing data effectively and creating seamless data integrations to build patient-centric healthcare solutions.”

This chapter will cover the key HIS sectors where MongoDB can be useful.

**Patient data management**

Storing and processing structured patient information is the basis of any HIS. Since we are in the era of digitalization, most of the data is stored in the database. The database stores data such as the patient's name, surname, year of birth, etc.

According to a review by Itesh Sharma (2021), electronic medical records (EMR) provide:

• Centralized tracking of medical history;

• Automation of routine tasks;

• Notifications about routine examinations and screenings;

• Monitoring key health indicators (blood pressure, vaccination status).

(Source: TatvaSoft, 2021)

https://www.tatvasoft.com/outsourcing/2021/05/health-information-system.html#:~:text=Key%20Components%20of%20Health%20Information,Pharmacy%20Management

**Key functional requirements**

The system MUST provide secure creation, storage, retrieval, updating and archiving

The system MUST guarantee the immutability of critical records (e.g., records of operations, prescription of potent drugs) and maintain a detailed audit of all changes to patient data (who, when, what changed).

The system MUST support standardized data exchange formats (HL7 FHIR, CDA) for integration with laboratories, pharmacies , etc.

**User and access management**

Since this system is multi-tenant, it is important to understand that this system will be used not only by doctors but also by patients, nurses , system administrators and finally by the other clinics. Data security requires strict delineation of access rights in a multi-tenant environment

**Key Functional Requirements**

• FR2.1 (Strict Tenant Data Isolation):

• Our system MUST ensure that 1 user's data is not accessible to another user. This is 1 of the important aspects of the system

• FR2.2 (Role-Based Access Control - RBAC): The system MUST have a clear implementation of roles and access to information based on the role.

**Example**:

Doctor: Full access to their patients' data within their healthcare facility, the right to prescribe treatment.

Nurse: Access to appointments and procedure schedules for their patients/department in their healthcare facility, the right to mark completion.

Patient: Access only to their EMR via the portal.

System Administrator: Global infrastructure management (no access to patients' medical data!).

• FR2.3 (Strong Authentication): The system MUST require strong authentication for all users before granting access to data.

• FR2.4 (Audit Access): The system MUST record in secure logs all attempts to access data (successful and unsuccessful) indicating the user, time, type of operation, and data object. The proxy MUST contribute to these logs.

**Data Security and Regulatory Compliance**

• FR3.1 (Encryption): The system MUST provide encryption of data both at rest and in transit.

• FR3.2 (Data Integrity): The system MUST ensure that critical medical records cannot be altered or deleted without authorization.

• FR3.3 (Compliance): The system MUST comply with international and national standards for the protection of personal data and medical confidentiality (HIPAA, GDPR, ISO

**Effectiveness of MongoDB in Various Areas of HIS**

When developing a healthcare system, it is necessary to realize that the introduction of reporting for medical institutions requires flexibility, scalability speed and finally storing huge amount of data, which, unfortunately, traditional relational DBMS cannot boast of. MongoDB, as a document-oriented NoSQL database, offers a fundamentally different approach. Its flexible data model (JSON/BSON), horizontal scaling and a powerful indexing mechanism make it an ideal solution for key HIS subsystems.

As Karolina Ruiz Rogelj rightly notes, "The document model also supports nested and hierarchical data structures, which simplifies the presentation of complex clinical data with different levels of detail and granularity" [1]. (. https://www.mongodb.com/blog/post/four-ways-mongodb-solves-healthcares-interoperability-puzzle)

This fundamental feature of MongoDB finds direct application in 3 key real cases,

1. Electronic health records (EHR),

1. 2. Medical imaging systems (PACS/VNA),
2. 3. Clinical data repositories (CDR)

Electronic medical records

• Storage of information that is subject to constant changes (new types of examinations, dynamic forms). Changes are made without expensive ALTER TABLE (SQL) operations and downtime. In MongoDB: You can add new fields to a document without changing the structure of existing records

Example: If a hospital introduces a new type of test (e.g. blood test), in MongoDB you can simply add a blood \_tests field to new patient records without affecting existing documents. The author (Francesc Mateu Amengual) also notes “The constant changes can be as simple as a name change or can be so complex that data has to be rearranged.”

• Millions of write/read operations per day) thanks to the optimized WiredTiger engine and memory management.

• Complex hierarchical structures: patient → visits → diagnoses → medications.

Can be implemented through nested documents and arrays. Supports data exchange standards (HL7 FHIR), often based on JSON.

A screenshot of a computer

AI-generated content may be incorrect.

Own schema

// a

• Such a document naturally reflects the data structure, but searching for {allergies: "Penicillin"} without the tenant\_id filter will return patients from all clinics -> DLP is necessary.

• Millions of operations: Specify the types of operations where speed is important: find() of medical records during emergency admission, update() of patient status in the operating room, insert() of streaming data from sensors in the intensive care unit.////// to do

However, in a multi-tenant environment where data from multiple clinics is stored together, the flexibility of the scheme alone does not guarantee strict isolation of patient records from different tenants. This creates a risk of privacy breaches and requires an additional level of control.

Medical Imaging Systems (PACS/VNA):

**Scalability:** Ideal for managing information thanks to MongoDB's horizontal scalability (sharding), which allows the distribution of load and the storage of petabytes of data(DICOM Data).

DICOM (Digital Imaging and Communications in Medicine) is a generally accepted international standard for storing, processing and transmitting medical images and related data between various medical devices and systems.

Another technology that MongoDB offers is MongoDB GridFS.

DICOM images (CT, MRI) often exceed the 16MB per document limit in MongoDB. GridFS solves this problem by automatically splitting the file into chunks (e.g. 255KB) and storing metadata separately. This enables transparent storage of large objects while maintaining the benefits of MongoDB sharding and replication. As Luís A. Bastião Silva Carlos Costa Louis Beroud and José Luís Oliveira rightly point out “GridFS provides horizontal scalability through replication in several nodes"

• Query performance: Fast search of studies by patient, date, type of study, doctor thanks to complex indexes on DICOM metadata (stored as BSON documents).

Horizontal scaling effectively distributes the load but does not solve the 'noisy neighbour' problem, where a resource-intensive request from one tenant (such as a bulk image upload) can slow down access to critical data for other clinics**.**

Integrated Clinical Data Warehouses (CDR):

• Flexibility: Combines disparate data from multiple sources (labs, registries, departments, external systems) into a single view of the patient, even if the structure of the sources differs or changes.

• Operational Analytics: MongoDB aggregation framework enables on-the-fly preliminary analytics (calculation of department averages, detection of disease trends, quality control) without complex ETL processes.

The ability to quickly aggregate data from different sources is valuable, but in a multi-tenant environment, aggregations must be strictly limited to the current tenant's data to avoid cross-tenant information leaks.

The solution to these problems can be Data-Limiting Proxy (DLP). DLP is a critical component of the architecture, complementing MongoDB.

DLP provides:

* Strict data isolation through automatic filtering of all requests by tenant\_id
* Performance isolation by applying limits (number of documents, execution time) at the tenant level
* Centralized access auditing with a tenant link. Only in combination with DLP can MongoDB fully realize its potential in mission-critical, regulated multi-tenant healthcare systems, meeting HIPAA/GDPR requirements

Data Isolation and Security in Healthcare

When implementing multi-tenant healthcare systems, security is a key consideration, as we work with sensitive data in healthcare.

Unauthorized access to this data can have dire consequences.

For example,

In 2023, over 133 million patient records were exposed in healthcare-related breaches, according to HHS.gov. This number highlights just how heavily targeted PHI has become—especially on the dark web.

https://www.hipaavault.com/resources/dark-web-healthcare-phi/

or in 2024, a ransomware attack on Change Healthcare exposed the protected health information of a staggering 100 million people. Change Healthcare said the hackers, ALPHV (also known as BlackCat), accessed their system through a single user account. How? The account did not have multi-factor authentication enabled.

[**https://www.teamviewer.com/en/insights/healthcare-data-security/?#the**](https://www.teamviewer.com/en/insights/healthcare-data-security/?#the)

Security Mechanisms in Multi-Tenant Systems

Data separation and isolation are key to the security of a multi-user environment. The key parameter in this regard is RBAC.

It works by restricting access to certain persons; for example, we have an administrator role and a user who cannot view the administrator's information. In the case of two people with the same role, the ABAC method is used, which,

Case study

Case study in health care: Here insert code which I will add should look something like this:class EcommerceTenant

depending on the ID, provides access to data. This method works in conjunction with RBAC and provides better data isolation

Why data isolation is important:

Data isolation can create an impenetrable barrier to protect data and prevent it from being accessed, stolen, or corrupted. In a multi-tenant environment, strict data and resource isolation can prevent unauthorized access to the dataset and avoid data leakage.

The another method which can help us isolate data is implementing tenant\_id to each request

Particular attention is also paid to data encryption, which is the main pillar of security in a multi-user environment.

Encryption - consists of a key that can contain numbers, string or text and is controlled by an algorithm. The length of the key is important. Let's take two keys as an example, 256-bit and 128, for 1 it is necessary to try 2 ^ 256 combinations, in the other 2 ^ 128 the result is that in the first case we have to used additional computing resources There are two types of encryptions - symmetric and asymmetric. In symmetric encryption, the same key is used to encrypt and decrypt data. It is usually used when a lot of data is involved (such as in a database) because it is faster. The private key is used to perform encryption, and the public key is used to decrypt the data. For better understanding, a table taken from the literature from NIST will be attached, in the table we can see how many rounds will be spent, that is, according to the formula, it is also important to note that the AES encryption standard is used here

A table with numbers and letters

Description automatically generated

https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197-upd1.pdf

Another thing is authorization and authentication. During the authentication process, the system checks who the user is and provides access to his data.

**Audit**

Governance: Governance involves checking documents and sensitive data (such as an account number, password, Social Security number, or date of birth). This includes both knowing where sensitive data is stored and preventing it from being entered into the system.

What does multitenant mean for the cloud?

Cloud providers offer multi-tenancy as a means of sharing the use of computing resources. However, this shared use of resources should not be confused with virtualization, a closely related concept. In a multitenant environment, multiple clients share the same application, in the same operating environment, on the same hardware, with the same storage system. In virtualization, unlike multitenancy, each application runs on a separate virtual machine with its operating system.

Each resident has authorized access to their apartment, yet all residents share water, electricity, and common areas. Similarly, in a multitenant cloud, the provider sets broad terms and performance expectations, but individual customers have private access to their information.

The multitenant design of a cloud service can dramatically impact the delivery of applications and services. It enables unprecedented reliability, availability, and scalability while enabling cost savings, flexibility, and security for IT organizations.

**Data Schema design**

In this chapter in I would like to discuss the way the data schemas of the system were designed. The principle - to ensure isolation between various tenants (tenants) in a cloud environment, where each of them possesses its own limits of data. MongoDB was used by us and all schemes are designed keeping security, scalability and simplicity. We can split it up into parts, using the code as examples.

The data architecture utilized is a multitenant type of data structure (1.1).

I have chosen to isolate the data at the database level so that the data of different companies does not interfere with each other. Each tenant has a separate database with the prefix tenant. It is easy and sure enough - no ordinary tables, that you can fall all over.

The primary information storing scheme among tenants looks as follows:

1. @Schema()

2. export class Tenant extends Document {

3. @Prop({ required: true })

4. companyName: string;

5.

6. @Prop({ required: true })

7. tenantId: string;

8. }

We store the company name and its unique identifier. This information is used by the system to route requests to the correct database.

**1.2 System of limits and usage control**

Limit management is the main goal in our project since we are working in multitenant environment. For the effectiveness of our system was implemented two schemes: one for setting maximums and one for tracking current usage. With these schemes, we can easily manage the number of documents, space, or requests used by each tenant.

Data Limit scheme - here quotas are set, or, more simply, a limit on memory usage or the number of documents that each tenant can store:

1. @Schema({ versionKey: false })

2. export class DataLimit extends Document {

3. @Prop({ required: true, unique: true })

4. tenantId: string;

5.

6. @Prop({ required: true, default: 1000 })

7. maxDocuments: number;

8.

9. @Prop({ required: true, default: 51200 })

10. maxDataSizeKB: number;

11.

12. @Prop({ required: true, default: 1000 })

13. monthlyQueries: number;

14. }

15.

This scheme defines three types of limits:

• Maximum number of documents (default 1000)

• Maximum data size in kilobytes (default 50 MB)

• Maximum number of requests per month (default 1000)

I also added a schema that tracks how much resources were used (for example, memory, data, and how many queries were executed) by each user. The schema stores key factors such as tenantId, which speeds up searching for tenant statistics.

1. @Schema({ versionKey: false, timestamps: true })

2. export class DataUsage extends Document {

3. @Prop({ required: true, index: true })

4. tenantId: string;

5.

6. @Prop({ required: true, default: 0 })

7. documentsCount: number;

8.

9. @Prop({ required: true, default: 0 })

10. dataSizeKB: number;

11.

12. @Prop({ required: true, default: 0 })

13. queriesCount: number;

14. }

15.

**1.3 Audit and Monitoring Framework**

To ensure no action is missed, I've added an audit log for all actions. Imagine that it is a kind of event log: we save the time, level of errors, individual who made a mistake and restricted information. It can be used in compliance and debugging.

Key features include:

* Event time, level (info, warn, error), request ID, user, and tenant.
* Request method (GET, POST), address (URL), response code, and execution time.
* Limit information: event type (LIMIT\_EXCEEDED, LIMIT\_WARNING, LIMIT\_UPDATED, USAGE\_SPIKE), limit type (documents, data, requests), and details (current value, limit, usage percentage).

For convenience, I've added timestamps: true which creates the event time as a string to help track limit exceedances.

1. @Schema({ timestamps: true })

2. export class AuditEvent extends Document {

3. @Prop({ required: true })

4. timestamp: string;

5.

6. @Prop({ required: true, enum: ['info', 'warn', 'error'] })

7. level: string;

8.

9. @Prop()

10. requestId?: string;

11.

12. @Prop()

13. userId?: string;

14.

15. @Prop()

16. tenantId?: string;

17.

18. @Prop({ required: true })

19. method: string;

20.

21. @Prop({ required: true })

22. path: string;

23.

24. @Prop({ required: true })

25. statusCode: number;

26.

27. @Prop({ required: true })

28. durationMs: number;

29.

30

31. @Prop({ enum: ['LIMIT\_EXCEEDED', 'LIMIT\_WARNING', 'LIMIT\_UPDATED', 'USAGE\_SPIKE'] })

32. eventType?: string;

33.

34. @Prop({ enum: ['DOCUMENTS', 'DATA\_SIZE', 'QUERIES'] })

35. limitType?: string;

36.

37. @Prop({ type: Object })

38. limitData?: {

39. currentValue: number;

40. limitValue: number;

41. attemptedValue?: number;

42. percentage?: number;

43. };

44. }

45.

**1.4 Business Data Schemas**

Now let's talk about real data. We work with patients and users, so the schemas include validation to avoid input errors. We use DTOs for the API.

Patient Schema (CreatePatientDto)

Stores data about each patient:

* name: first name (string, required, not empty).
* surname: last name (string, required, not empty).
* age: age (integer, required, not less than 0).

1. export class CreatePatientDto {

2. @ApiProperty({

3. description: 'Patient first name',

4. example: 'Jan',

5. })

6. @IsString()

7. @IsNotEmpty()

8. name: string;

9.

10. @ApiProperty({

11. description: 'Patient last name',

12. example: 'Novotny',

13. })

14. @IsString()

15. @IsNotEmpty()

16. surname: string;

17.

18. @ApiProperty({

19. description: 'Patient age',

20. example: 30,

21. minimum: 0,

22. })

23. @Type(() => Number)

24. @IsInt()

25. @Min(0)

26. age: number;

27. }

28.

User schema (UserDto):

• name: full name (string, 2 to 50 characters, required).

• email: email (required, valid format, e.g., john.doe@example.com).

• password: password (number, required, from 100000 to 999999999).

1. export default class UserDto {

2. @ApiProperty({

3. description: 'User full name',

4. example: 'Jack Sparrow',

5. minLength: 2,

6. maxLength: 50

7. })

8. @IsNotEmpty({ message: 'Username is required' })

9. @IsString({ message: 'Username must be string' })

10. @MinLength(2, { message: 'The name must contain at least 2 characters.' })

11. @MaxLength(50, { message: 'The name must not exceed 50 characters.' })

12. name: string;

13.

14. @ApiProperty({

15. description: 'User email address',

16. example: 'john.doe@example.com'

17. })

18. @IsNotEmpty({ message: 'Email is required' })

19. @IsEmail({}, { message: 'Incorrect email format' })

20. email: string;

21.

22. @ApiProperty({

23. description: 'User password',

24. example: 123456,

25. minimum: 100000,

26. maximum: 999999999

27. })

28. @Type(() => Number)

29. @IsNotEmpty({ message: 'Password is required' })

30. @IsInt({ message: 'Password must be a number' })

31. @Min(100000, { message: 'The password must be at least 100000.' })

32. @Max(999999999, { message: 'The password must not exceed 999999999.' })

33. password: number;

34.

**}**

Login schema (LoginCredentialsDto):

* email: email (required, valid format).
* password: password (number, required, from 100000 to 999999999).

1. export class LoginCredentialsDto {

2. @ApiProperty({

3. description: 'User email address',

4. example: 'admin@hospital1.ru',

5. })

6. @IsNotEmpty({ message: 'Email required' })

7. @IsEmail({}, { message: 'invalid email' })

8. email: string;

9.

10. @ApiProperty({

11. description: 'User password',

12. example: 123456,

13. minimum: 100000,

14. maximum: 999999999,

15. })

16. @Type(() => Number)

17. @IsNotEmpty({ message: 'password required' })

18. @IsInt({ message: 'Password must be number' })

19. @Min(100000, { message: 'Password must be at least 100000' })

20. @Max(999999999, { message: 'Password must not exceed 999999999' })

21. password: number;

22. }

23.

**1.5 Database Configuration**

The configuration is simple and flexible—everything is stored in .env files. This allows you to easily change settings without recompiling.

Basic settings:

* port: Server port (default 3000).
* connectionString: MongoDB connection address (from .env).
* encryptionSecretKey: Data encryption key (from .env).

1. export default () => ({

2. server: {

3. port: process.env.PORT || 3000,

4. },

5. database: {

6. connectionString: process.env.DB\_CONNECTION\_STRING,

7. },

8. security: {

9. encryptionSecretKey: process.env.ENCRYPTION\_KEY,

10. },

11. });

12.

And the connection to MongoDB is asynchronous:

1. MongooseModule.forRootAsync({

2. imports: [ConfigModule],

3. useFactory: async (config: ConfigService) => {

4. const uri = config.get<string>('database.connectionString');

5. return {

6. uri: uri || 'mongodb://localhost:27017/defaultdb',

7. };

8. },

9. inject: [ConfigService],

10. }),

11.

The asynchronous connection was designed to handle high system loads, as a synchronous connection would block each thread until completion. Another key feature is minimizing latency and maintaining system operation even under peak loads.

Finally, this structure provides isolation, limit control, and validation—everything needed for a robust system.

**2. Configuring the Data-Limiting Proxy**

In this chapter, I'd like to discuss how to set up a proxy server. The proxy, or gateway, in our application checks limits before performing any actions.

I integrated it into NestJS, with middleware, interceptors, and services.

**2.1 Proxy System Architecture**

The entire system is assembled in the main AppModule. This is where we connect the configuration, JWT authentication, MongoDB, and all other modules. This makes the application modular and easy to maintain.

Key elements:

Configuration: Settings are stored in the .env file (port, MongoDB connection string, encryption keys), making them easy to change.

Authentication: JWT is used to verify users, available globally.

Connection to MongoDB: Asynchronous, via MongooseModule, with the address from .env or the default (mongodb://localhost:27017/defaultdb).

Modules: Modules for tenants, patients, users, authentication, audit, proxy, and limits are enabled.

1. @Module({

2. imports: [

3. ConfigModule.forRoot({

4. isGlobal: true,

5. cache: true,

6. load: [configuration],

7. }),

8. JwtModule.register({

9. global: true

10. }),

11. MongooseModule.forRootAsync({

12. imports: [ConfigModule],

13. useFactory: async (config: ConfigService) => {

14. const uri = config.get<string>('database.connectionString');

15. return {

16. uri: uri || 'mongodb://localhost:27017/defaultdb',

17. };

18. },

19. inject: [ConfigService],

20. }),

21. TenantsModule,

22. PatientsModule,

23. UsersModule,

24. AuthModule,

25. AuditModule,

26. ProxyModule,

27. LimitsModule

28. ],

29. controllers: [AppController],

30. providers: [

31. AppService,

32. { provide: APP\_INTERCEPTOR, useClass: AuditInterceptor },

33. ],

34. })

35. export class AppModule implements NestModule {

36. configure(consumer: MiddlewareConsumer) {

37. consumer

38. .apply(TenantsMiddleware)

39. .forRoutes(LimitsController);

**2.2 Middleware for Multi-Tenancy**

To ensure that requests are coming from the correct tenant, we use a middleware.

The purpose of TenantsMiddleware is to verify that requests are coming from the correct tenant.

To better understand how it works, I'll explain the middleware step by step.

1. It takes the x-tenant-id header in the request that defines the tenant ID.
2. When the header is missing, then it raises an exception of BadRequestException ("X-TENANT-ID is not provided).
3. Then we asynchronously query the service on whether the tenant exists, and in case it is not, we get the error "tenant does not exist" back.
4. In case everything has worked out, the tenantId is added to the req object to be processed further via next() function

1. @Injectable()

2. export class TenantsMiddleware implements NestMiddleware {

3. constructor(private tenantsService: TenantsService) { }

4.

5. async use(req: Request, res: Response, next: NextFunction) {

6. const tenantId = req.headers['x-tenant-id']?.toString();

7. if (!tenantId) {

8. throw new BadRequestException('X-TENANT-ID is not provided');

9. }

10.

11. const tenantExists = await this.tenantsService.getTenantById(tenantId)

12. if (!tenantExists) {

13. throw new NotFoundException('tenant does not exist')

14. }

15.

16. req['tenantId'] = tenantId;

17. next();

18. }

19. }

20.

**2.3 Interceptor for Limit Context**

The interceptor is an intermediate layer designed to collect and structure information about requests.

When processing each incoming HTTP request, the interceptor sequentially collects the following parameters:

• Request identifier - extracted from the x-request-id header or generated based on a timestamp

• User identifier - determined from the req.user authentication object

• Client environment - the user-agent header is analyzed

• Network address - the IP address of the request source is recorded

• Target resource - the endpoint and HTTP method of the request are remembered

1. 1. @Injectable()

2. 2. export class LimitsContextInterceptor implements NestInterceptor {

3. 3. intercept(context: ExecutionContext, next: CallHandler): Observable<any> {

4. 4. const http = context.switchToHttp();

5. 5. const req = http.getRequest<Request>();

6. 6.

7. 7. const limitsContext: RequestContext = {

8. 8. requestId: req.headers['x-request-id'] as string || `req-${Date.now()}`,

9. 9. userId: (req as any).user?.id,

10. 10. userAgent: req.headers['user-agent'],

11. 11. ipAddress: req.ip,

12. 12. endpoint: req.originalUrl || req.url,

13. 13. method: req.method,

14. 14. };

15. 15.

16. 16. (req as any).limitsContext = limitsContext;

17. 17. return next.handle();

18. 18. }

19. 19.

20. }

The LimitsService manages document limits for tenants:

Initialization:

* Injects a connection to MongoDB and an audit service.
* Creates the limitsModel and usageModel models to manage limits and document usage.

The checkDocumentsLimit method:

* Checks the document limit for a tenantId via limitsModel.
* Gets or creates a document usage record in usageModel.
* If no limit is set, terminates the check.
* Calculates the new number of documents and the usage percentage.
* If 90% of the limit is exceeded, sends a warning via emitLimitWarning.
* If the limit is exceeded, calls emitLimitViolation and throws a ForbiddenException with error details.

**2.4 Limits Management Service**

The LimitsService is the core service. It checks for exceeded limits and generates audit events. Here's an example of a document check:

When a request is received, the service:

* Finds the client's established limit
* Checks the current number of documents
* If there is no limit, it allows the request
* If there is a limit, it calculates the new number of documents and the usage percentage.
* When 90% of the limit is reached, the service sends a warning. If the tenant reaches the storage limit, the action is blocked and an error is returned.
* This way, we limit system overload.

1. @Injectable()

2. export class LimitsService {

3. private readonly limitsModel;

4. private readonly usageModel;

5.

6. constructor(

7. @InjectConnection() private connection: Connection,

8. private readonly auditService: AuditService

9. ) {

10. this.limitsModel = this.connection.model(DataLimit.name, DataLimitSchema);

11. this.usageModel = this.connection.model(DataUsage.name, DataUsageSchema);

12. }

13.

14. async checkDocumentsLimit(

15. tenantId: string,

16. incomingDocsCount: number = 1,

17. context?: RequestContext

18. ): Promise<void> {

19. const limit = await this.limitsModel.findOne({ tenantId }).exec();

20. const usage = await this.usageModel.findOne({ tenantId }).exec() ||

21. await this.usageModel.create({ tenantId });

22.

23. if (!limit) {

24. return;

25. }

26.

27. const newTotal = usage.documentsCount + incomingDocsCount;

28. const percentage = Math.round((newTotal / limit.maxDocuments) \* 100);

29.

30. if (newTotal >= limit.maxDocuments \* 0.9 && usage.documentsCount < limit.maxDocuments \* 0.9) {

31. await this.emitLimitWarning(tenantId, 'DOCUMENTS', {

32. currentValue: usage.documentsCount,

33. limitValue: limit.maxDocuments,

34. percentage: Math.round((usage.documentsCount / limit.maxDocuments) \* 100)

35. }, context);

36. }

37.

38. if (newTotal > limit.maxDocuments) {

39. await this.emitLimitViolation(tenantId, 'DOCUMENTS', {

40. currentValue: usage.documentsCount,

41. limitValue: limit.maxDocuments,

42. attemptedValue: incomingDocsCount,

43. percentage: percentage

44. }, context);

45.

46. throw new ForbiddenException({

47. message: `Document limit exceeded. Current: ${usage.documentsCount}, Limit: ${limit.maxDocuments}, Attempted to add: ${incomingDocsCount}`,

48. error: 'DOCUMENT\_LIMIT\_EXCEEDED',

49. details: {

50. current: usage.documentsCount,

51. limit: limit.maxDocuments,

52. attempted: incomingDocsCount,

53. percentage: percentage

54. }

55. });

56. }

57. }

58. }

59.

**2.5 Checking Data Size Limits**

Similarly, for data size, we check, warn, and block if necessary.

1. async checkDataSizeLimit(

2. tenantId: string,

3. incomingDataSizeKB: number,

4. context?: RequestContext

5. ): Promise<void> {

6. const limit = await this.limitsModel.findOne({ tenantId }).exec();

7. const usage = await this.usageModel.findOne({ tenantId }).exec() ||

8. await this.usageModel.create({ tenantId });

9.

10. if (!limit) {

11. return;

12. }

13.

14. const newTotal = usage.dataSizeKB + incomingDataSizeKB;

15. const percentage = Math.round((newTotal / limit.maxDataSizeKB) \* 100);

16.

17. if (newTotal >= limit.maxDataSizeKB \* 0.9 && usage.dataSizeKB < limit.maxDataSizeKB \* 0.9) {

18. await this.emitLimitWarning(tenantId, 'DATA\_SIZE', {

19. currentValue: usage.dataSizeKB,

20. limitValue: limit.maxDataSizeKB,

21. percentage: Math.round((usage.dataSizeKB / limit.maxDataSizeKB) \* 100)

22. }, context);

23. }

24.

25. if (newTotal > limit.maxDataSizeKB) {

26. await this.emitLimitViolation(tenantId, 'DATA\_SIZE', {

27. currentValue: usage.dataSizeKB,

28. limitValue: limit.maxDataSizeKB,

29. attemptedValue: incomingDataSizeKB,

30. percentage: percentage

31. }, context);

32.

33. throw new ForbiddenException({

34. message: `Data size limit exceeded. Current: ${usage.dataSizeKB}KB, Limit: ${limit.maxDataSizeKB}KB, Attempted: ${incomingDataSizeKB}KB`,

35. error: 'DATA\_SIZE\_LIMIT\_EXCEEDED',

36. details: {

37. current: usage.dataSizeKB,

38. limit: limit.maxDataSizeKB,

39. attempted: incomingDataSizeKB,

40. percentage: percentage

41. }

42. });

43. }

44. }

45.

**2.6 Limit Management Controller**

To effectively manage tenant limits, API methods have been added to the controller that allow viewing and changing limits. All operations are protected by guards.

The controller provides three main functions:

• Viewing tenant limits

• Changing limits

• Obtaining usage statistics

The security system verifies each request. It is only accessed by authorized users. Everything is passed on to LimitsService. The controller does not do anything but accept requests and give responses. By doing so, the proxy will act as a standalone entity and the administrators can easily change the settings through the API.

1. @Controller('limits')

2. @UseGuards(TenantAuthenticationGuard)

3. export class LimitsController {

4. constructor(private readonly limitsService: LimitsService) { }

5.

6. @Get(':tenantId')

7. async getLimits(@Param('tenantId') tenantId: string) {

8. return this.limitsService.getLimitsForTenant(tenantId);

9. }

10.

11. @Put(':tenantId')

12. async setLimits(@Param('tenantId') tenantId: string, @Body() newLimits: any) {

13. return this.limitsService.setLimitsForTenant(tenantId, newLimits);

14. }

15.

16. @Get('usage/:tenantId')

17. async getUsage(@Param('tenantId') tenantId: string) {

18. return this.limitsService.getUsageForTenant(tenantId);

19. }

20. }

21.

Overall, the proxy operates automatically: it checks limits, logs, and blocks any irregularities. This makes the system secure and predictable.

**3. Examining Data**

In this section I will explain how we analyse data, starting with structure, to monitoring. It is just a matter of isolation and control to have transparency.

When designing this system, I decided it would be best to store each tenant's information in a separate database.

An example to describe this is the case of issuing a safe deposit box to a client.

Step by step:

A request of the patient data is sent to the system:

This checks first whether or not the client has surpassed the request limit.

Thereafter, it links to their own database.

And only then, it delivers the demanded data.

This helps to avoid the mixing up of data amongst tenants.

Access example:

1. async getPatients(tenantId: string) {

2. await this.limitsService.checkQueriesLimit(tenantId);

3. const tenantConnection = this.tenantConnection.useDb(`tenant\_${tenantId}`);

4. const PatientModel = tenantConnection.model(Patient.name, PatientSchema);

5. return PatientModel.find();

6. }

7.

8. async getPatientById(tenantId: string, patientId: string) {

9. await this.limitsService.checkQueriesLimit(tenantId);

10. const tenantConnection = this.tenantConnection.useDb(`tenant\_${tenantId}`);

11. const PatientModel = tenantConnection.model(Patient.name, PatientSchema);

12. return PatientModel.findById(patientId);

13. }

14.

Patient search works in the same way: the system always accesses the database and then the limits.

In the previous section, we mentioned that the system monitors document usage and issues an alert when the tenant sets a limit (90%) or blocks operations if it is exceeded. Now let's look at how this works for the other metrics—data size and number of requests—and how we handle these events.

For data size (dataSizeKB), the mechanism is similar. The proxy doesn't subsequently calculate a new data volume based on the set limit. If usage reaches 90% of the maximum, the system sends a warning via AuditService. If the limit is exceeded, the operation is blocked, and an error is logged.

Here's an example of the data size check we already saw, but now with full context processing:

1. async checkDataSizeLimit(

2. tenantId: string,

3. incomingDataSizeKB: number,

4. context?: RequestContext

5. ): Promise<void> {

6. const limit = await this.limitsModel.findOne({ tenantId }).exec();

7. const usage = await this.usageModel.findOne({ tenantId }).exec() ||

8. await this.usageModel.create({ tenantId });

9.

10. if (!limit) {

11. return;

12. }

13.

14. const newTotal = usage.dataSizeKB + incomingDataSizeKB;

15. const percentage = Math.round((newTotal / limit.maxDataSizeKB) \* 100);

16.

17. if (newTotal >= limit.maxDataSizeKB \* 0.9 && usage.dataSizeKB < limit.maxDataSizeKB \* 0.9) {

18. await this.emitLimitWarning(tenantId, 'DATA\_SIZE', {

19. currentValue: usage.dataSizeKB,

20. limitValue: limit.maxDataSizeKB,

21. percentage: Math.round((usage.dataSizeKB / limit.maxDataSizeKB) \* 100)

22. }, context);

23. }

24.

25. if (newTotal > limit.maxDataSizeKB) {

26. await this.emitLimitViolation(tenantId, 'DATA\_SIZE', {

27. currentValue: usage.dataSizeKB,

28. limitValue: limit.maxDataSizeKB,

29. attemptedValue: incomingDataSizeKB,

30. percentage: percentage

31. }, context);

32.

33. throw new ForbiddenException({

34. message: `Data size limit exceeded. Current: ${usage.dataSizeKB}KB, Limit: ${limit.maxDataSizeKB}KB, Attempted: ${incomingDataSizeKB}KB`,

35. error: 'DATA\_SIZE\_LIMIT\_EXCEEDED',

36. details: {

37. current: usage.dataSizeKB,

38. limit: limit.maxDataSizeKB,

39. attempted: incomingDataSizeKB,

40. percentage: percentage

41. }

42. });

43. }

44.

45. // Обновляем использование

46. await this.usageModel.findOneAndUpdate(

47. { tenantId },

48. { $inc: { dataSizeKB: incomingDataSizeKB } },

49. { upsert: true }

50. ).exec();

51. }

The same is so with queriesCount. Every query is counted and when it attains 90 percent of the limit a warning is emitted. When this limit is reached, queries are blocked until the next cycle (e.g. a month). Here's how it's implemented:

1. async checkQueriesLimit(

2. tenantId: string,

3. context?: RequestContext

4. ): Promise<void> {

5. const limit = await this.limitsModel.findOne({ tenantId }).exec();

6. const usage = await this.usageModel.findOne({ tenantId }).exec() ||

7. await this.usageModel.create({ tenantId });

8.

9. if (!limit) {

10. return;

11. }

12.

13. const newTotal = usage.queriesCount + 1;

14. const percentage = Math.round((newTotal / limit.monthlyQueries) \* 100);

15.

16. if (newTotal >= limit.monthlyQueries \* 0.9 && usage.queriesCount < limit.monthlyQueries \* 0.9) {

17. await this.emitLimitWarning(tenantId, 'QUERIES', {

18. currentValue: usage.queriesCount,

19. limitValue: limit.monthlyQueries,

20. percentage: Math.round((usage.queriesCount / limit.monthlyQueries) \* 100)

21. }, context);

22. }

23.

24. if (newTotal > limit.monthlyQueries) {

25. await this.emitLimitViolation(tenantId, 'QUERIES', {

26. currentValue: usage.queriesCount,

27. limitValue: limit.monthlyQueries,

28. attemptedValue: 1,

29. percentage: percentage

30. }, context);

31.

32. throw new ForbiddenException({

33. message: `Query limit exceeded. Current: ${usage.queriesCount}, Limit: ${limit.monthlyQueries}`,

34. error: 'QUERY\_LIMIT\_EXCEEDED',

35. details: {

36. current: usage.queriesCount,

37. limit: limit.monthlyQueries,

38. attempted: 1,

39. percentage: percentage

40. }

41. });

42. }

43.

44. // Обновляем счетчик запросов

45. await this.usageModel.findOneAndUpdate(

46. { tenantId },

47. { $inc: { queriesCount: 1 } },

48. { upsert: true }

49. ).exec();

50. }

51.

These checks are built into every data operation. For example, when retrieving a list of patients:

1. 1. async getPatients(tenantId: string) {

2. 2. await this.limitsService.checkQueriesLimit(tenantId);

3. 3. const tenantConnection = this.tenantConnection.useDb(`tenant\_${tenantId}`);

4. 4. const PatientModel = tenantConnection.model(Patient.name, PatientSchema);

5. 5. return PatientModel.find();

When limits are exceeded, we receive warnings that we have exceeded the limit, and limit violations are recorded in AuditEvent. This is necessary so that we promptly record excess information. If the tenant wants to store more information, we can offer to increase the quota in advance.

1. private async emitLimitWarning(

2. tenantId: string,

3. limitType: 'DOCUMENTS' | 'DATA\_SIZE' | 'QUERIES',

4. limitData: any,

5. context?: RequestContext

6. ): Promise<void> {

7. const event: LimitWarningEvent = {

8. timestamp: new Date().toISOString(),

9. level: 'warn',

10. requestId: context?.requestId || `warn-${Date.now()}`,

11. userId: context?.userId,

12. tenantId,

13. method: context?.method || 'INTERNAL',

14. path: context?.endpoint || '/limits/check',

15. statusCode: 200,

16. durationMs: 0,

17. ip: context?.ipAddress,

18. userAgent: context?.userAgent,

19. message: `${limitType} usage approaching limit for tenant ${tenantId}`,

20. eventType: 'LIMIT\_WARNING',

21. limitType,

22. limitData,

23. metadata: {

24. service: 'LimitsService',

25. action: 'checkLimit',

26. severity: 'MEDIUM'

27. }

28. };

29.

30. this.auditService.emit(event);

31. }

32.

This makes the system transparent: the administrators are able to see when limits are approaching and the users are given direct feedback when they breach quotas.

**4. Assessing Data Security and Privacy**

When working in a multi-tenant architecture, it's important to understand that we will have many users. This raises the issue of security and privacy, which is one of the key aspects of a multi-tenant system.

It's important to ensure that one tenant's data is not shared with another tenant and that access is restricted to authorized users. In this chapter, I'll discuss how the data was protected and demonstrate how I implemented it.

**4.1 Authentication and Authorization**

When choosing an authentication system for my project, I chose JWT for a number of reasons.

* Self-contained
* Flexibility and scalability
* Security
* Suitable for API

How it works:

Each request must contain:

• JWT token - confirms the user's identity

• x-tenant-id header - indicates which tenant the request refers to

Authentication is done using JWT and the tenant ID is verified using the x-tenant-id header. This provides API level access control. A guard safeguards all the paths pertaining to limits.

1. TenantAuthenticationGuard

2. @Controller('limits')

3. @UseGuards(TenantAuthenticationGuard)

4. export class LimitsController {

5. constructor(private readonly limitsService: LimitsService) { }

6.

7. @Get(':tenantId')

8. async getLimits(@Param('tenantId') tenantId: string) {

9. return this.limitsService.getLimitsForTenant(tenantId);

10. }

11.

12. @Put(':tenantId')

13. async setLimits(@Param('tenantId') tenantId: string, @Body() newLimits: any) {

14. return this.limitsService.setLimitsForTenant(tenantId, newLimits);

15. }

16. }

17.

**4.2 Data isolation**

For complete isolation, it was decided to use separate databases for each tenant with a tenantId prefix, which subsequently eliminates the possibility of accidental data intersection.

1. async createPatient(tenantId: string, createPatientDto: CreatePatientDto) {

2. await this.limitsService.checkDocumentsLimit(tenantId, 1); //

3. const tenantDb = this.tenantConnection.useDb(`tenant\_${tenantId}`);

4. const PatientModel = tenantDb.model(Patient.name, PatientSchema);

5. const newPatient = new PatientModel({ ...createPatientDto });

6. return newPatient.save();

7. }

**4.3 Input Data Validation**

To protect against invalid data or attacks, we use DTOs with validation via a class-validator. For example, to create a patient:

1. export class CreatePatientDto {

2. @ApiProperty({

3. description: 'Patient first name',

4. example: 'John',

5. })

6. @IsString()

7. @IsNotEmpty()

8. name: string;

9.

10. @ApiProperty({

11. description: 'Patient last name',

12. example: 'Doe',

13. })

14. @IsString()

15. @IsNotEmpty()

16. surname: string;

17.

18. @ApiProperty({

19. description: 'Patient age',

20. example: 30,

21. minimum: 0,

22. })

23. @Type(() => Number)

24. @IsInt()

25. @Min(0)

26. age: number;

27. }

28.

This way, the system ensures that the data undergoes rigorous verification before being stored.

**4.4 Configuration Encryption**

Important settings such as encryption keys, passwords, and other sensitive data are stored in environment variables:

1. export default () => ({

2. server: {

3. port: process.env.PORT || 3000,

4. },

5. database: {

6. connectionString: process.env.DB\_CONNECTION\_STRING,

7. },

8. security: {

9. encryptionSecretKey: process.env.ENCRYPTION\_KEY,

10. },

11. });

12.

The bcrypt library was used to encrypt passwords.

**4.5 Full audit of operations**

All operations are recorded using Auditservice to monitor the activity of the users and the system. This assists in meeting such standards as GDPR.

1. @Schema({ timestamps: true })

2. export class AuditEvent extends Document {

3. @Prop({ required: true })

4. timestamp: string;

5.

6. @Prop({ required: true, enum: ['info', 'warn', 'error'] })

7. level: string;

8.

9. @Prop()

10. requestId?: string;

11.

12. @Prop()

13. tenantId?: string;

14.

15. @Prop({ required: true })

16. method: string;

17.

18. @Prop({ required: true })

19. path: string;

20.

21. @Prop({ required: true })

22. statusCode: number;

23.

24. @Prop({ enum: ['LIMIT\_EXCEEDED', 'LIMIT\_WARNING', 'LIMIT\_UPDATED', 'USAGE\_SPIKE'] })

25. eventType?: string;

26.

27. @Prop({ enum: ['DOCUMENTS', 'DATA\_SIZE', 'QUERIES'] })

28. limitType?: string;

29. }

30.

Example of limit exceeding log:

1. private async emitLimitViolation(

2. tenantId: string,

3. limitType: 'DOCUMENTS' | 'DATA\_SIZE' | 'QUERIES',

4. limitData: any,

5. context?: RequestContext

6. ): Promise<void> {

7. const event: LimitViolationEvent = {

8. timestamp: new Date().toISOString(),

9. level: 'error',

10. requestId: context?.requestId || `limit-${Date.now()}`,

11. tenantId,

12. method: context?.method || 'INTERNAL',

13. path: context?.endpoint || '/limits/check',

14. statusCode: 403,

15. durationMs: 0,

16. message: `${limitType} limit exceeded for tenant ${tenantId}`,

17. eventType: 'LIMIT\_EXCEEDED',

18. limitType,

19. limitData,

20. };

21. this.auditService.emit(event);

22. }

23.

This way we have achieved security and transparency. Data is protected and actions are tracked.

**5. Evaluating Performance and Scalability**

In the future, the system is expected to scale even further, so it must be fast, even with thousands of tenants; it must adapt to increasing load. I'll show you how we optimized queries and architecture.

**5.1 Modular Structure**

NestJS makes the system modular. This allows components, from limits to auditing, to be separated into modules, making it easier to add new features.

1. @Module({

2. imports: [

3. ConfigModule.forRoot({

4. isGlobal: true,

5. cache: true,

6. load: [configuration],

7. }),

8. MongooseModule.forRootAsync({

9. imports: [ConfigModule],

10. useFactory: async (config: ConfigService) => ({

11. uri: config.get<string>('database.connectionString') || 'mongodb://localhost:27017/defaultdb',

12. }),

13. inject: [ConfigService],

14. }),

15. TenantsModule,

16. PatientsModule,

17. UsersModule,

18. AuthModule,

19. AuditModule,

20. ProxyModule,

21. LimitsModule,

22. ],

23. controllers: [AppController],

24. providers: [

25. AppService,

26. { provide: APP\_INTERCEPTOR, useClass: AuditInterceptor },

27. ],

28. })

29. export class AppModule implements NestModule {

30. configure(consumer: MiddlewareConsumer) {

31. consumer.apply(TenantsMiddleware).forRoutes(LimitsController);

32. }

33. }

34.

**5.2 Database Optimization**

Indexes have been added to speed up queries. For example, in DataUsage::

1. @Schema({ versionKey: false, timestamps: true })

2. export class DataUsage extends Document {

3. @Prop({ required: true, index: true })

4. tenantId: string;

5.

6. @Prop({ required: true, default: 0 })

7. documentsCount: number;

8.

9. @Prop({ required: true, default: 0 })

10. dataSizeKB: number;

11.

12. @Prop({ required: true, default: 0 })

13. queriesCount: number;

14. }

15.

A unique index in DataLimit prevents duplicates:

1. @Schema({ versionKey: false })

2. export class DataLimit extends Document {

3. @Prop({ required: true, unique: true })

4. tenantId: string;

5.

6. @Prop({ required: true, default: 1000 })

7. maxDocuments: number;

8. }

9.

**How does this work:**

unique: true creates a unique index

What does this mean?

There can only be one tenant in the DataLimit collection. So, for example, if we have a tenant named tenant\_ABC, creating a new tenant with the same name is impossible.

Attempts to create a second record with the same tenantId, and MongoDB will return an error.

This ensures that each tenant has only one set of limits.

**5.3 Efficient Operations**

**To update metrics, use findOneAndUpdate with upsert:**

1. async updateUsage(tenantId: string, docsCount: number, dataSizeKB: number): Promise<void> {

2. await this.usageModel.findOneAndUpdate(

3. { tenantId },

4. {

5. $inc: {

6. documentsCount: docsCount,

7. dataSizeKB: dataSizeKB,

8. queriesCount: 1

9. }

10. },

11. { upsert: true }

12. ).exec();

13. }

14.

These methods do two things in one operation:

1. Search for a record by tenantId

2. If found, updates the counters

3. If not found, creates a new record

Why this is efficient:

• One operation instead of three - no need to check for existence and create separately

• Atomicity - MongoDB guarantees that counters will increment correctly even with concurrent requests

• Performance - fewer database calls = higher speed

When creating multiple patients through a single request, the Bulk Insert Patients feature has been optimized:

Step by step

* Checking limits - the system checks if there is enough space for all new patients
* Connecting to the client's database - using their personal database
* Bulk creation - adding all patients with one command
* Updating statistics - increasing usage counters

1. async createBulkPatients(tenantId: string, patientsDto: CreatePatientDto[]) {

2. await this.limitsService.checkDocumentsLimit(tenantId, patientsDto.length);

3. const tenantDb = this.tenantConnection.useDb(`tenant\_${tenantId}`);

4. const PatientModel = tenantDb.model(Patient.name, PatientSchema);

5. const createdPatients = await PatientModel.insertMany(patientsDto);

6. await this.limitsService.updateUsage(tenantId, patientsDto.length, patientsDto.length);

7. return createdPatients;

**5.4 Caching and Asynchrony**

**Configuration is cached via ConfigModule:**

1. ConfigModule.forRoot({

2. isGlobal: true,

3. cache: true,

4. load: [configuration],

5. }),

Audit runs asynchronously via RabbitMQ to avoid blocking the main thread:

1. @Injectable()

2. export class AuditService {

3. constructor(

4. @Inject('AUDIT\_SERVICE') private readonly client: ClientProxy,

5. @InjectModel(AuditEventDocument.name) private auditModel: Model<AuditEventDocument>

6. ) { }

7.

8. async emit(event: AuditEvent) {

9. const auditRecord = new this.auditModel(event);

10. await auditRecord.save();

11. this.client.emit('audit-log', event);

12. }

13. }

14.

**5.5 Monitoring**

The times required to execute requests are traced to examine bottlenecks:

1. async checkQueriesLimit(tenantId: string, context?: RequestContext): Promise<void> {

2. console.log('🔍 [DEBUG] checkQueriesLimit - START');

3. // ... логика

4. console.log('🔍 [DEBUG] ✅ Query limit check passed');

5. }

The system is also scalable through isolation, indexing and asynchrony, so that it can run fast even with a large load.

**Conclusion**

An effective data management system in a multi-tenant environment has been developed. Now we will sum up the results and consider the further steps.

* 6.1 What has been achieved
* Isolation: The data of tenants is segregated into databases (tenant\_...).
* Limits: Document, size, query control and warn.
* Security JWT, validation and operation auditing.
* Performance: Query optimization and scale.

**6.2 Limitations and Improvements.**

The present system estimates data conditionally (1 KB per document). A more accurate calculation may be incorporated through an MongoDB aggregation. Prometheus monitoring and Redis caching integration is also something to take into consideration.

**6.3 Applicability**

This would be best in SaaS, medical platforms or other cloud-based solutions where isolation and limits are of concern. It assists in cost-control and adherence to security standards

**6.4 Conclusion.**

Our system is scalable, and we have created an elastic and secure system. This gives it a strong basis to build on in the future, e.g., analytics or automatic scale of limits.

Reference

Cloud computing

1. 1https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-145.pdf
2. Resource pooling
3. (<https://staragile.com/blog/resource-pooling-in-cloud-computing#:~:text=Resource%20pooling%20is%20the%20heart,or%20country%2C%20for%20resource%20access>.,
4. https://www.3ritechnologies.com/resource-pooling-in-cloud-computing/
5. <https://www.synopsys.com/blogs/chip-design/essential-cloud-computing-characteristics.html>
6. broad network access [**Vikram Bhatia**](https://www.synopsys.com/blogs/chip-design/authors/vikram-bhatia.html)
7. history of cloud computing
8. <https://www.dataversity.net/brief-history-cloud-computing/>)
9. https://learn.microsoft.com/en-us/azure/architecture/antipatterns/noisy-neighbor/noisy-neighbor
10. <https://staragile.com/blog/resource-pooling-in-cloud-computing#:~:text=Resource%20pooling%20is%20the%20heart,or%20country%2C%20for%20resource%20access>.
11. <https://spot.io/resources/cloud-optimization/cloud-infrastructure-4-key-components-and-deployment-models/>
12. <https://www.coursera.org/in/articles/components-of-cloud-computing>
13. <https://industri.fatek.unpatti.ac.id/wp-content/uploads/2019/03/210-Cloud-Computing-Sandeep-Bhowmik-Edisi-1-2017.pdf>
14. <https://cloudsecurityalliance.org/blog/2023/05/11/8-things-healthcare-organizations-can-do-to-ensure-hipaa-compliance-in-the-cloud>
15. https://www.metomic.io/resource-centre/ensuring-hipaa-compliance-in-cloud-based-healthcare-systems#:~:text=How%20does%20HIPAA%20compliance%20affect,identify%20and%20address%20potential%20vulnerabilities.
16. https://industri.fatek.unpatti.ac.id/wp-content/uploads/2019/03/210-Cloud-Computing-Sandeep-Bhowmik-Edisi-1-2017.pdf

**Introduction to Mongodb**

1. <https://www.mongodb.com/resources/products/capabilities/features>
2. Chodorow
3. <https://ftp.mclarkdev.com/uploads/library/Programming/The%20Definitive%20Guide/mongodbthedefinitiveguide.pdf>
4. <https://www.techtarget.com/searchdatamanagement/definition/MongoDB>

Sass

<https://cloud.google.com/learn/paas-vs-iaas-vs-saas?hl=en>

<https://www.ibm.com/think/topics/iaas-paas-saas#:~:text=IaaS%2C%20PaaS%20and%20SaaS%20are,%2C%20cloud%2Dhosted%20application%20software>.

https://db-engines.com/en/

https://www.ippathways.com/understanding-cloud-service-models-iaas-paas-saas/#:~:text=Each%20cloud%20computing%20service%20model%20offers%20a,to%20understand%20their%20differences%20and%20benefits%20better.

1. **Multiitenant**
2. https://www.researchgate.net/publication/387867858\_Multi-Tenant\_Architectures\_in\_Modern\_Cloud\_Computing\_A\_Technical\_Deep\_Dive

HIS

https://www.mongodb.com/company/blog/innovation/four-ways-mongodb-solves-healthcares-interoperability-puzzle

<https://www.researchgate.net/publication/269291153_Medical_imaging_archiving_A_comparison_between_several_NoSQL_solutions>

<https://www.hipaavault.com/resources/dark-web-healthcare-phi/>

<https://techcrunch.com/2025/01/27/how-the-ransomware-attack-at-change-healthcare-went-down-a-timeline/>

<https://www.industryarc.com/Report/4339/Healthcare-Information-Systems-Market-Research-Report.html>

<https://www.mongodb.com/company/blog/innovation/dual-journey-healthcare-interoperability-modernization>

https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197-upd1.pdf

<https://www.researchgate.net/publication/387867858_Multi-Tenant_Architectures_in_Modern_Cloud_Computing_A_Technical_Deep_Dive>

[https://docs.nestjs.com](https://docs.nestjs.com/pipes)