incedo lighthouse™

incedo

Whitepaper

Multi-Tenancy Simplified for SaaS: Data Isolation with Per Tenant Database Schema

incedo

Table of Content

Introduction

Purpose of the Paper

The goal of the paper is to address the complexities of data isolation introduced by engineering systems managing several tenants, which may result in data privacy infringements and regulatory noncompliance. This investigation seeks to analyze several ways for ensuring data isolation among tenants in the design of multi-tenant systems and provide insights gained from our practical experience in implementing multi-tenancy in our own software SaaS platform to enable readers to make informed decisions on the selection of a data isolation approach that aligns with their application requirements.

Introduction to multi-tenancy

Multi-tenancy pertains to the capability of a single application instance to accommodate several businesses or business units inside the same firm, referred to as tenants, utilizing a shared infrastructure comprising computing, storage, networking, and application delivery resources. In this context, the term "single tenant" refers to all users associated with a single corporate body or business unit inside a company, so differentiating them from individual end-users. It has become prevalent in the era of cloud computing and software-as-a-service (SaaS) applications.

The objective of multi-tenancy is to optimize resource efficiency while upholding stringent boundaries for each tenant's data, hence assuring data isolation.

Prominent instances of multi-tenant apps encompass Salesforce, Microsoft 365, Google Workspace, and WordPress.com.

Structure of the Paper

The document commences with an overview of multi-tenancy, emphasizing its fundamental concepts and the advantages it offers in software engineering. Subsequently, we present a thorough examination of how applications might employ data isolation to effectively accommodate multiple clients utilizing a single instance. As a case study, we will discuss how we implemented multi-tenancy in one of our platforms, moving from dedicated hosting to hosting numerous tenants on shared infrastructure.

Audience

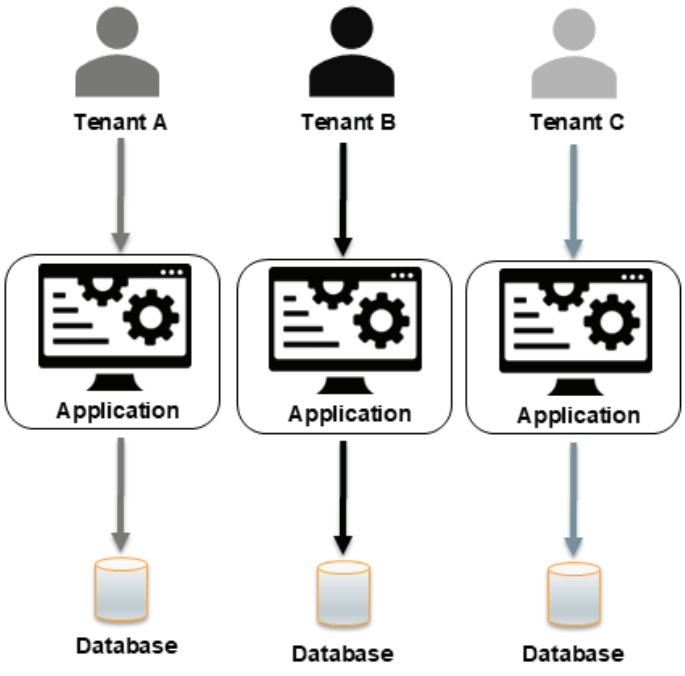
This paper is intended for anyone engaged in the development of multi-tenant solutions, specifically Independent Software Vendors (ISVs) that provide software for both enterprise clients and consumer internet applications. This includes Chief Technology Officers, software architects, software engineers, and product managers.

Transitioning to Multi-Tenancy: Challenges and Benefits

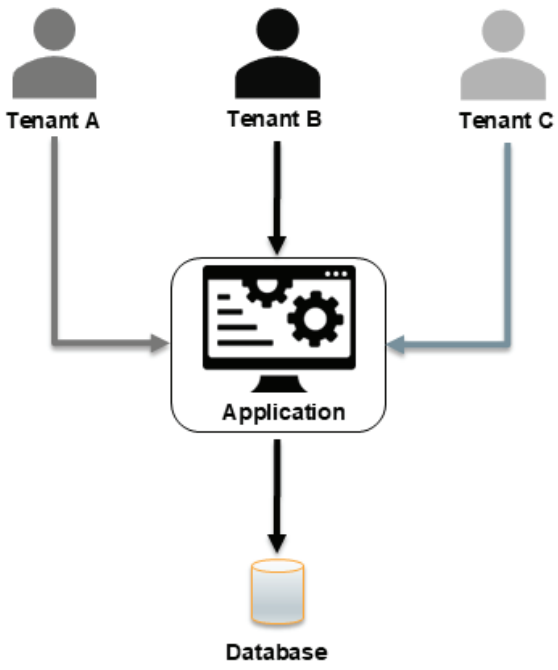
Transitioning to multi-tenancy presents distinct problems, and the application's complexity may first seem to escalate. Nonetheless, it has substantial advantages. When performed correctly, it enables numerous tenants to utilize a single application and shared infrastructure while preserving data segregation, resulting in enhanced operational efficiency and reduced costs.

The diagram illustrates the transition from a traditional single-tenant architecture to a multi-tenant architecture, highlighting the fact that less infrastructure is required in multi-tenancy paradigm.

Before Multi-Tenancy



After Multi-Tenancy



The table below illustrates the problems associated with multi-tenancy and the advantages of its adoption.

Aspect	Challenges Before Multi-tenancy	Benefits of Multi-tenancy
Infrastructure Requirement	**Dedicated Infrastructure for Distinct Clients:** Necessitated individual setups resulting in excessive expenses and operational overheads.	**Utilization of Shared Infrastructure for Various Clients:** The allocation of shared infrastructure resources among multiple tenants reduces costs and operational expenditures.
Change Management	**Significant Overhead:** Deployments required coordination across several environments, resulting in delays and operational burdens.	**Optimized Change Management:** Centralized oversight of updates, patches, and upgrades enhances administration efficiency and minimizes overhead.
Support & Maintenance	**Extensive Support and Maintenance:** The management of many instances of the same application resulted in intricate support, maintenance, and update responsibilities, hence elevating administrative overhead.	**Reduced Support and Maintenance:** Fewer instances result in diminished efforts for support, maintenance, and update activities.
Scalability	**Scalability Challenges:** Scaling necessitated the independent provisioning of additional resources for each tenant, resulting in inefficiencies.	**Improved Scalability:** Capability to support expanding users or demand via effective resource distribution and scaling.
Architecture Complexity	**Fragmented Architecture:** The IT architecture was characterized by fragmentation resulting from resource redundancy and integration difficulties among clients.	**Streamlined IT Architecture:** A cohesive multi-tenant environment enhances IT infrastructure integration and minimizes administrative overhead.
Customizations	**At-will, ad hoc, and poorly conceived fragmented customizations:** Bespoke business logic gain per client gains traction with poorly managed frontline commitments over meticulously planned setups and well-defined customization workflows, such as those for enterprise onboarding and branding.	**Promotes superior design thinking and engineering:** Product-oriented approach for meticulously considered configurations and customizations readily available to guarantee the platform's functionality across numerous tenants.

SaaS Application Design Considerations

In developing an enterprise grade application, several key factors must be considered to ensure its success and effectiveness. The design considerations for our application involve evaluating various aspects that impact its performance, scalability, security, and usability. This includes addressing how to effectively manage data isolation, and support customization while balancing resource utilization and system efficiency. By carefully considering these elements, we can create a robust architecture that meets the needs of diverse users and ensures enterprise grade characteristics and security. Below are the application design considerations that must be kept in mind:

Data Isolation: In a multi-tenant architecture, ensuring robust data isolation is critical to maintaining tenant privacy and security. Data isolation can be achieved through various strategies, such as using separate databases, schemas, or tables for each tenant. The choice of isolation level depends on the specific needs of the application, with higher levels of isolation providing better security but potentially increasing complexity and resource usage. Effective data isolation prevents unauthorized access and ensures that a tenant's data remains private and unaffected by other tenants' operations. Choosing the appropriate data isolation level (e.g., database, schema, table) is also influenced by many factors such as existing infrastructure constraints, performance, billing, and customization.

Dynamic Tenant Onboarding and Offboarding: Implementing streamlined processes for adding new tenants to the system and removing existing tenants is required to be done out-of-the-box without downtime. For enterprise users onboarding, there may be a need to integrate with tenant's authentication and authorization systems if pre-existing connectors with the application are not yet available. For offboarding there may be a need to build a software feature to export the data prior to deletion of tenant data.

Resource Allocation and Quotas: For enforcement of fair usage policies or upgrades to higher billing tiers or for usage based pricing models, defining resource allocation policies and quotas for each tenant to prevent resource contention and enforcement is a key aspect for configuring the resources and application design.

Site Reliability Engineering and Observability: Implementing comprehensive monitoring and management capabilities to track system performance, tracking changes, and understanding how users interact with the system through logs, traces, and metrics, detecting anomalies, data inconsistencies across all tenants are key for continuous system improvements and improving customer experience.

Compliance and Regulations: Ensuring that the multi-tenant solution complies with industry regulations and standards related to data privacy, security, and data residency requirements for each tenant is a key requirement.

Database Performance: The solution should have sufficient strategies and techniques in place to scale the database and mitigate database performance issues as platform scales. Leveraging a scalable approach for database architecture ensures that the SaaS provider can onboard and maintain many tenants without the risk of significant performance degradation. There are a variety of strategies used to scale databases in a SaaS system. Examples include read replication to distribute read traffic, multi-source setups for high availability, sharding to horizontally partition the database, and using multiple clusters to manage different workloads. Additionally, techniques like database caching, for CQRS (Command Query Responsibility Segregation), and serverless databases enhance performance and scalability, ensuring the system can efficiently manage growing and varied tenant demands.

Billing: Billing in a multi-tenant application requires precise tracking of resource usage by each tenant. This can include CPU, memory, storage, and API calls, among other metrics. Implementing usage-based billing allows for a fair and transparent charging model, where tenants pay for exactly what they use. The billing system should be tightly integrated with the application's resource monitoring tools to accurately capture usage data. In addition, the billing system must be flexible enough to accommodate different pricing models, such as pay-as-you-go, tiered pricing, or subscription-based plans, to meet the diverse needs of tenants.

Application Delivery Network and other shared Hosting Considerations: For application delivery network (ADN) elements like shared infrastructure, web servers, DNS (Domain Name Server), and WAF (Web Application Firewall), it is important to note that these components often rely on shared resources across tenants. For example:

- **Web Servers:** Typically, web servers are shared among tenants, which can lead to resource contention. However, with proper configuration and scaling, performance can be maintained.
- **DNS:** DNS settings are shared and managed centrally, with tenants potentially sharing the same domain but using subdomains for isolation.
- **WAF L4 (Layer 4) vs. L7 (Layer 7):** The choice between Layer 4 (L4) and Layer 7 (L7) WAF depends on the level of inspection required. L4 focuses on network traffic, while L7 provides more granular protection by analysing application-level data, which might be crucial for multi-tenant applications where tenants share the same application stack.
- **Load Balancer:** Load balancer distributes incoming traffic across multiple servers for performance and reliability, especially when tenants share the same infrastructure.

SaaS Application Security Considerations

Ensuring robust security is crucial in multi-tenancy applications, as it safeguards tenant data, enforces isolation, and protects against potential vulnerabilities at every layer of the system. Below points must be considered in ensuring the security of the application:

Comprehensive Security Across All Architectural Layers: Ensuring robust tenant isolation requires comprehensive security measures integrated into every layer of the architecture. At the network layer, firewalls, intrusion detection systems, and secure network configurations are essential to prevent unauthorized access. The application layer must implement strong authentication and authorization protocols, enforcing tenant-specific access controls to protect against unauthorized data exposure. At the data layer, encryption both at rest and in transit is crucial to safeguarding sensitive tenant information. Regular security audits, compliance checks, and continuous monitoring across all layers further strengthen the overall security posture, ensuring that tenant data remains isolated and protected from potential threats.

Authentication: For multi-tenancy applications, authentication is crucial for ensuring secure access control tailored to each tenant's needs. Each tenant may employ different authentication and authorization mechanisms based on their requirements. This flexibility allows integration with various protocols such as SAML (Security Assertion Markup Language), OAuth (Open Authorization) and integration with multiple Identity and Access Management (IAM) Providers like OneLogin, AWS Cognito, Azure AD (Active Directory), and PingId. By supporting diverse authentication methods, the application can accommodate varying security standards and compliance requirements across different tenants, enhancing overall security while maintaining adaptability and user management efficiency.

Encryption of Data: For multi-tenancy applications, encryption is essential for protecting sensitive data and maintaining tenant privacy. Implementing encryption ensures that data is securely stored and transmitted, with the added benefit of enabling tenant-specific encryption keys. Each tenant can have its own set of encryption keys, enhancing data isolation and security. These keys can be managed using client-preferred vault solutions, such as HashiCorp Vault, AWS Secrets Manager, or Azure Key Vault. This approach allows for flexible and secure key management practices, ensuring that data remains protected according to the specific needs and policies of each tenant.

Entitlement & RBAC (Role Based Access Control): The solution should offer the capability to scope access to features or product functionality. Leveraging role-based access control (RBAC) in SaaS environments enables developers to manage and restrict access to various products, features, and functions within and across applications. This model is typically supported through a centralized mechanism that enforces access policies consistently across the platform. Advanced and flexible RBAC implementations should provide a richer set of options, allowing administrators to associate multiple roles with individual users, thereby tailoring access permissions to specific needs of various tenants.

Cross-Tenant Prevention: In a schema-based multi-tenancy model, it is imperative to ensure strict measures are in place to prevent unauthorized access to data across tenants. Each tenant's data should be segregated within distinct schemas, enforcing isolation at the schema level. Robust authentication, authorization, and role-based access control mechanisms must be implemented to restrict data access to only authorized users within their respective tenant's schema. Additionally, employing encryption, network security protocols, and regular audits further fortify the system against potential breaches, safeguarding the integrity and confidentiality of tenant data. Also, different keys should be used for different tenants for encryption as mentioned above.

Cross-Tenant Event Tracking: To ensure regulatory and compliance standards are met, SaaS solutions must have the capability to log, audit, monitor, and alert on potential cross-tenant events. This involves implementing robust logging mechanisms to capture all relevant activities, performing regular audits to ensure adherence to policies, and employing real-time monitoring to detect any suspicious or unauthorized actions. Alerts are configured to notify administrators immediately if cross-tenant violations are detected, thereby ensuring swift response, and maintaining compliance with regulatory standards.

Trade-offs in Database Design for Design Isolation

Database design strategies determine how tenant data is managed and isolated within a shared infrastructure. This section explores the three primary approaches — dedicated database, schema, and table-based with each approach providing trade-offs for data isolation, scalability, and resource utilization.

Database based multi-tenancy: In this approach, each tenant's data is stored in a separate database instance. The configuration of database connections is created/maintained dynamically based on the current tenant, enabling isolation and scalability across multiple databases.

Schema-Based multi-tenancy: In schema-based multi-tenancy, each tenant's data is stored in a separate schema within the same database instance allowing dynamic switching of schemas based on the current tenant, ensuring data isolation while utilizing shared database resources.

Table-Based multi-tenancy: Table-based multi-tenancy involves storing data from multiple tenants within the same database table, with an additional column indicating the tenant association for each record. Support is provided for automatically filtering data based on the current tenant, ensuring data segregation at the application level.

Making a choice for database design for data isolation is not quite straight-forward and should consider other decision criterion such as flexibility and controls for quota/policy enforcement, performance degradation based on demand or users, ease of configurations, ease of development, and more and arrive at the trade-offs based on application under consideration.

The table below compares different multi-tenancy strategies - Database-Based, Schema-Based, and Table-Based across several decision criteria to help the reader in selecting the best approach for user's specific project needs.

Values:

- **High:** Optimal for the given criterion (e.g., high isolation, high quota enforcement with 'database based multi-tenancy').
- **Moderate:** Adequate but with some limitations.
- **Low:** Limited or constrained in the context of the given criterion.

Decision Criteria	Database	Schema	Table	Additional Information
Data Isolation - Degree to which data is separated of tenants	High	Moderate	Low	Database-based offers complete isolation, schema provides moderate separation within the same database, and table-based relies on application logic for isolation, thus lower.
Quota/Policy Enforcement (Disk Space Allocation, Limits Efficiency)	High	Moderate	Low	Database-based allows strict quota enforcement per tenant, schema allows for moderate control, table-based has the least granular enforcement of quotas/policies.
Performance - The ability of the system to manage loads and provide fast responses.	High	High	Low	Both database and schema approaches offer high performance due to isolation at the database or schema level. Table-based performance can degrade with tenant and data volume growth.
Configurations - Adjusting settings per tenant without change of code	High	Medium	Low	For each tenant having own database allows high level of configurability. You can configure settings independently for each tenant's database instance (e.g., different connection pools, disk space, or CPU allocation).
Customizations - Making changes or adding features specific to each tenant with code changes for workflows, user interfaces, or even how certain features behave for a specific tenant.	High	Medium	Low	Each tenant having its own database allows for a high degree of customization for changes such as specific application logic
Ease of Development - The simplicity and speed with which developers can create, manage, and maintain the environment.	Moderate	Moderate	High	Middleware like Java/Spring Hibernate simplifies tenant management for schema and table-based approaches, though database-based still requires additional complexity for configuration.
Resource Isolation - The ability to separate and manage resources independently for different tenants.	High	Moderate	Low	Databases provide full resource isolation, crucial for security and performance in multi-tenancy. Schemas offer partial isolation within a shared database, while tables share resources broadly.
Ease of Usage Billing: The ability to track and charge resource usage per tenant, varying from straightforward in database multi-tenancy to complex in table multi-tenancy	High	Moderate	Low	When each tenant has a separate database, tracking usage for billing purposes is straightforward. When all tenants share the same tables, with tenant data distinguished by a tenant identifier it makes things challenging to track and bill usage per tenant, as resources like queries, storage, and compute are shared.

Case Study: Data Isolation with Database Schema Per Tenant

The following case study covers our implementation approach for data isolation in one of our platforms, which transitioned from dedicated hosting to hosting many tenants on shared infrastructure in line with our philosophy to scale when you need to scale.

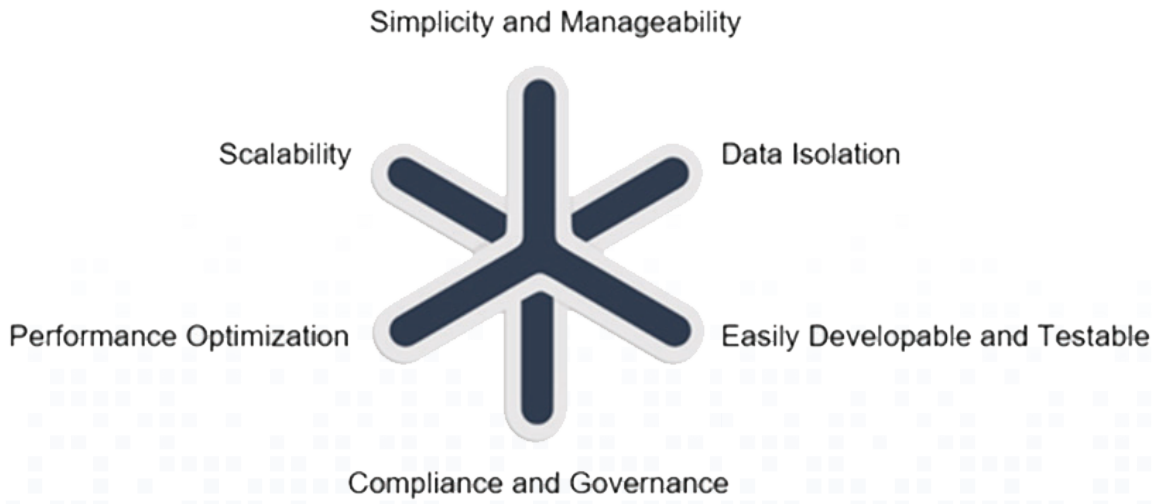
Overview of Application in consideration: AI-driven Decision Automation platform that enables business leaders to improve operational metrics and deliver higher business impact by automating insights from data, actions recommendations and integration into decision workflows.

Business model: The solution is offered as a Software-as-a-Service (SaaS) product.

End Users: CXOs, business and operational leaders, business analysts.

Constraint: Each tenant on the platform would introduce their own schema and data; for instance, a banking client would provide their mortgage application data along with distinct KPIs, which would differ from those of a wealth management client offering portfolio management advisory services, who would have a separate set of KPIs. As the client base expanded, all clients had to be integrated onto the platform without altering the code, necessitating a configuration-heavy and customization-light approach to minimize maintenance expenses, while also delivering insights tailored to each client to enhance their operational KPIs. This constraint effectively eliminated 'table-based design'.

Schema Based multi-tenancy was chosen due to below reasons:



Simplicity and Manageability: For the application in consideration, managing schema-based multi-tenancy is straightforward compared to other multi-tenancy models. With each tenant's data meticulously organized within its own schema, database administration tasks such as backup, restore, and maintenance become simpler and more efficient. Additionally, schema-based multi-tenancy simplifies data migration and tenant onboarding processes, reducing operational overhead.

Data Isolation: For the application in consideration, schema-based multi-tenancy ensures strong data isolation by storing each tenant's data in separate database schemas. This separation prevents data leakage and unauthorized access between tenants, enhancing data privacy and security which can easily happen in table based multi-tenancy.

Ease of code refactoring: For the application in consideration, schema-based multi-tenancy presented a seamless integration path with the current architecture of the application. Leveraging schema-based multi-tenancy allowed for a straightforward adapting of existing components and modules to support multi-tenancy requirements without necessitating extensive architectural overhaul. One of the key advantages of schema-based multi-tenancy is its compatibility with relational database management systems (RDBMS) used in the current infrastructure such as PostgreSQL or Oracle. Since most RDBMS platforms inherently support schema separation, integrating schema-based multi-tenancy into the existing database layer can be achieved with minimal code changing in existing application and hence needing for much less regression testing as well.

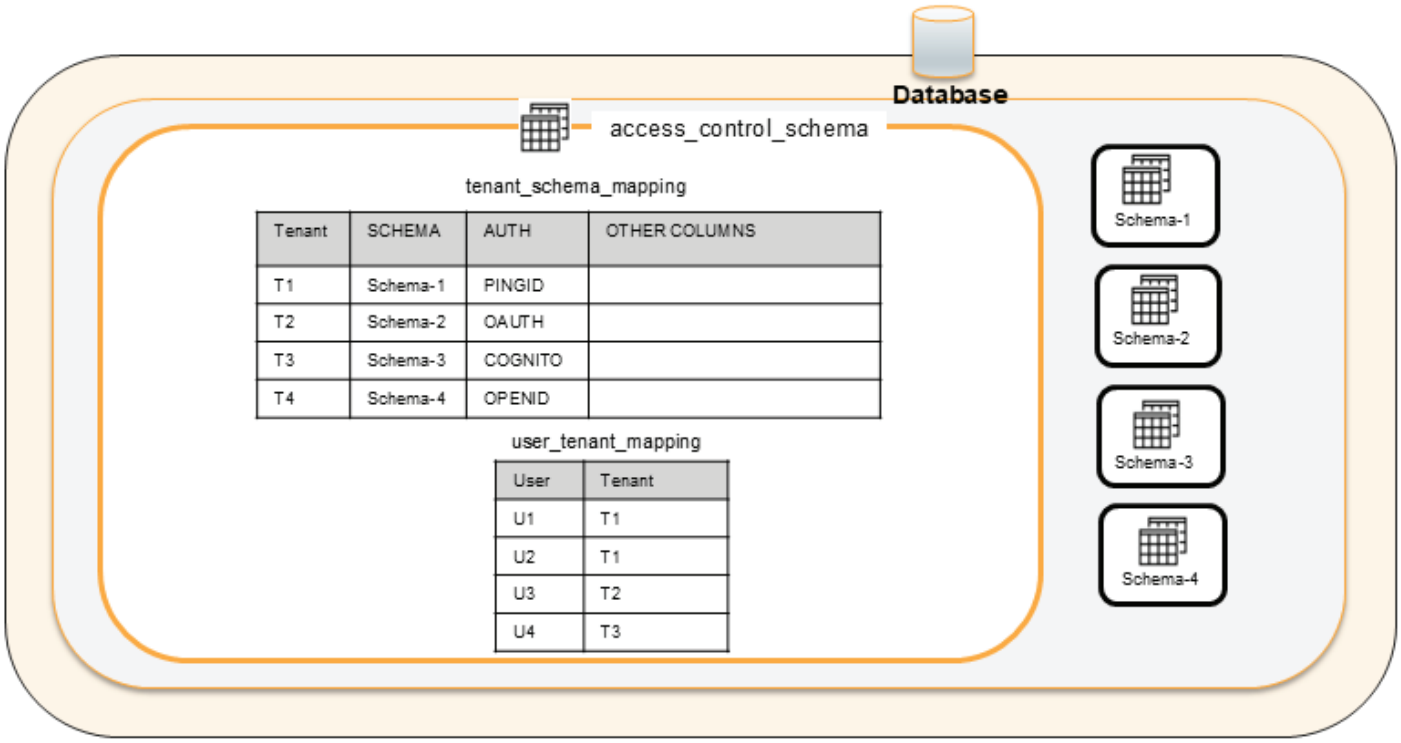
Compliance and Governance: Schema-based multi-tenancy simplifies compliance with industry regulations, as each organization's data remains exclusively within its assigned schema.

Performance Optimization: Segregating tenant data into separate schemas leads to more efficient data retrieval and query execution. This optimization minimizes contention for database resources and improves overall application performance, especially in scenarios with increasing tenants and increasing data volume.

Scalability: For the application in consideration, schema-based multi-tenancy offers scalability advantages by allowing for easy creation of schema either in-advance or dynamically. As the number of tenants grows, additional schemas can be provisioned dynamically, ensuring that the application can scale seamlessly to accommodate increased demand without sacrificing performance or stability, or new schemas can be created in advance to save the provisioning time.

Schema-based Multi-Tenancy Database design

Tenant onboarding is a key workflow in a multi-tenant environment, especially in B2B applications, where each tenant represents an entire organization or a business unit within an organization. In the architecture we have chosen for the application under consideration, each tenant operates within its dedicated schema. This approach ensures strong data isolation while simultaneously enabling the platform to support multiple tenants. The database design for the same is as depicted in the diagram below.



The database is designed with a dedicated schema (for representation purpose called

access_control_schema) for user management, which holds all relevant information about tenants and tenants' association with the users. This schema includes two key tables:

tenant_schema_mapping: This table maintains the mapping between each tenant and its associated schema.

user_tenant_mapping: This table records the association between users and their respective tenants. Additionally, the database contains individual tenant schemas for storing tenant-specific data. As shown in the diagram, Tenant T1 is associated with Schema-1 and has User U1.

This approach ensures that different schemas provisioned by application are for different tenants which ensures data isolation.

One Time Tenant Setup and User Sign-up Workflows

One-time Tenant Onboarding: When a new tenant such as a company or organization is onboarded, schema provisioning is done for the tenant. This involves creating a schema specifically for the tenant, which stores all the tenant's data as per tenant's requirements and inserting a record in access_control_schema's table tenant_schema_mapping to keep the mapping between the tenant and the schema name for the tenant.

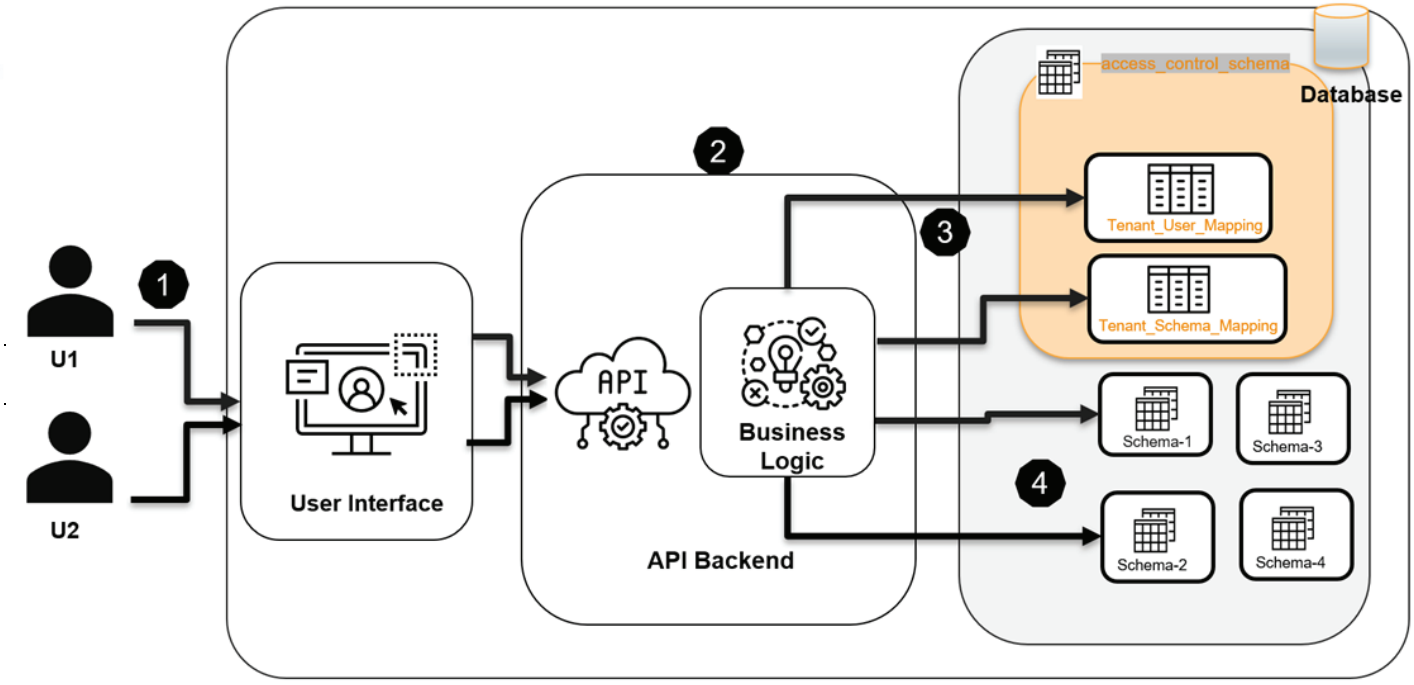
Further, different tenants may have varying security and compliance requirements, so the system allows for be-spoke integrations for authentication by having provision to save the authentication mechanism. For example, a tenant may opt for an in-house Open LDAP solution, while others may prefer other authentication protocols such as Security Assertion Markup Language (SAML), OAuth, or Active Directory Federation Services (ADFS) from Identity Providers like Microsoft, Ping ID, Okta, One Identity, OneLogin.

User Signup: When a new user registers in the application, depending on the tenant to which user belongs, a mapping is created in user_tenant_mapping table depicting the mapping between end-user & tenant. This ensures that each user gets correctly mapped by the application to their tenant, ensuring appropriate data access and isolation.

Multi-Tenancy Request Flow

The multi-tenancy request flow governs how requests from different tenants are securely processed and routed to their own schema preserving security and ensuring operational integrity.

Below over-simplified diagram depicts how the two users U1 & U2 can access the system concurrently retrieving the data from their configured schema based on the tenant to which they belong.



The workflows are:

Step 1: Users (U1 and U2) interact with the User Interface (UI) of the application.

Step 2: Once the UI captures the user input, it sends a request to the API Backend.

Step 3: The API's business logic module, interacts with the access_control_schema in the database to retrieve schema name based on request (explained further below in section Identifying the schema based on http request)

Step 4: Once the schema name is identified (as in step above). Query-rewriting is used to modify the queries based on the current tenant schema to ensure all operations—such as data retrieval, updates, and inserts are executed within the correct schema (explained further in section Query Rewriting)

Identifying the schema based on http request: To identify the schema for the user the username from the request needs to be fetched.

In scenarios, where the request contains an authentication token, the username is extracted from the token:

```
String token = request.getHeader("Authorization");
String username = jwtService.getUsernameFromToken(token);
```

If the token is encrypted, the system must first decrypt the token before extracting the username.

In scenarios not using token-based Authentication, username can be fetched from the Security Context (like Spring Security as referred to at https://docs.spring.io/spring-security/reference/). The method of retrieval is illustrated below:

```
Principal principal = SecurityContextHolder.getContext().getAuthentication();
String user = principal.getName();
```

After identifying the user, the system, needs to resolve the tenant's schema. For achieving this, the username can be used to look up the corresponding tenant in the User_Tenant_Mapping table for the request.

Example Using the figures in above previous section we can see that User U1 belongs to Tenant T1 & Tenant T1 is associated with Schema-1. Then using this schema name, we can re-write the query which is mentioned in the below section.

Query Rewriting: Once the schema name has been determined, the system performs query rewriting. This process involves replacing schema placeholders in the query (which is written in the business logic to fetch the required data as per the request) with the actual schema name specific to the tenant. This ensures that each tenant's data is stored and accessed from the correct schema, maintaining data isolation across the tenants.

Original SQL Query with Placeholder:	SELECT column_list FROM **{{schema}}**.mytable WHERE my_column = 123;
Rewritten Query for U1:	SELECT column_list FROM **Schema-1**.mytable WHERE my_column = 123;
Rewritten Query for U2:	SELECT column_list FROM **Schema-2**.mytable WHERE my_column = 123;

For building multi-tenanted applications in Java, Hibernate's built-in multi-tenancy functionality can be leveraged, and **Hibernate Interceptors** can be used to manipulate or rewrite queries as needed, making query handling more flexible and efficient.

In addition to above method, a tenant can be allocated a separate **subdomain** (like **tenant1**.myproduct.com) or a **slug URL** (like myproduct.com/tenant1) to further personalize and isolate the user experience. Both subdomain and slug URL approaches offer **branding** for the tenant.
In setups using subdomains (e.g., tenant1.myproduct.com), the tenant can be inferred directly from the subdomain. Similarly, a URL slug (e.g., myproduct.com/tenant1) can also help identify the tenant. Once the tenant is determined, the schema is fetched from the corresponding table.

By leveraging techniques mentioned above, the multi-tenancy architecture effectively ensures that requests are processed accurately, with proper data isolation and security across tenants.

Conclusion

The adoption of schema-based multi-tenancy especially for applications engineering for SaaS can be a key design choice towards achieving data isolation, implementing native schemas per client, scalability, cost efficiency, and flexibility in serving a diverse clientele while achieving seamless integration with shared infrastructure. Schema provisioning ensures clear separation between tenants, improving data isolation, security, and ease of management. This not only enhances data segregation but also simplifies compliance with industry regulations, as each organization's data remains exclusively within its assigned schema.

References

Microsoft Corporation. (2024, July 11). Architect multitenant solutions on Azure. Retrieved from https://learn.microsoft.com: https://learn.microsoft.com/en-us/azure/architecture/guide/multitenant/overview?ocid=AID754288&wt.mc_id=azfr-c9-scottha%2CCFID0719

Schauder, J. (2022, July 31). How to integrate Hibernates Multitenant feature with Spring Data JPA in a Spring Boot application. Retrieved from https://spring.io: https://spring.io/blog/2022/07/31/how-to-integrate-hibernates-multitenant-feature-with-spring-data-jpa-in-a-spring-boot-application

Hibernate.org. (2017, Jan 18). Multitenancy. Retrieved from https://docs.jboss.org: https://docs.jboss.org/hibernate/orm/5.0/userguide/html_single/chapters/multitenancy/MultiTenancy.html

Hibernate.org. (n.d.). Chapter-16 Multi-tenancy. Retrieved from https://docs.jboss.org: https://docs.jboss.org/hibernate/core/4.1/devguide/en-US/html/ch16.html

Incedo Inc. (n.d.). Incedo Lighthouse - AWS Solutions Consulting Offer. Retrieved from https://aws.amazon.com: https://aws.amazon.com/solutions/consulting-offers/incedo-lighthouse-ai-decision-automation-platform/

Introduction to Incedo Lighthouse™ – AI Decision Automation Platform

Incedo Lighthouse™ is an AI-driven Decision Automation platform that enables business executives to improve operational metrics and deliver higher business impact by automating insights and recommended actions and integrating them into decision workflows.

"With Incedo Lighthouse companies automate identification of operational problems that are deeply impacting business performance, leverage best-in-class AI and Data accelerators to reduce time to insights and achieve rapid implementation of recommended actions." ("Incedo Lighthouse - AWS Solutions Consulting Offer") It supports multiple deployment models such as on-prem and public cloud SaaS.

To learn more about Incedo SaaS practice and platforms, please contact the authors.

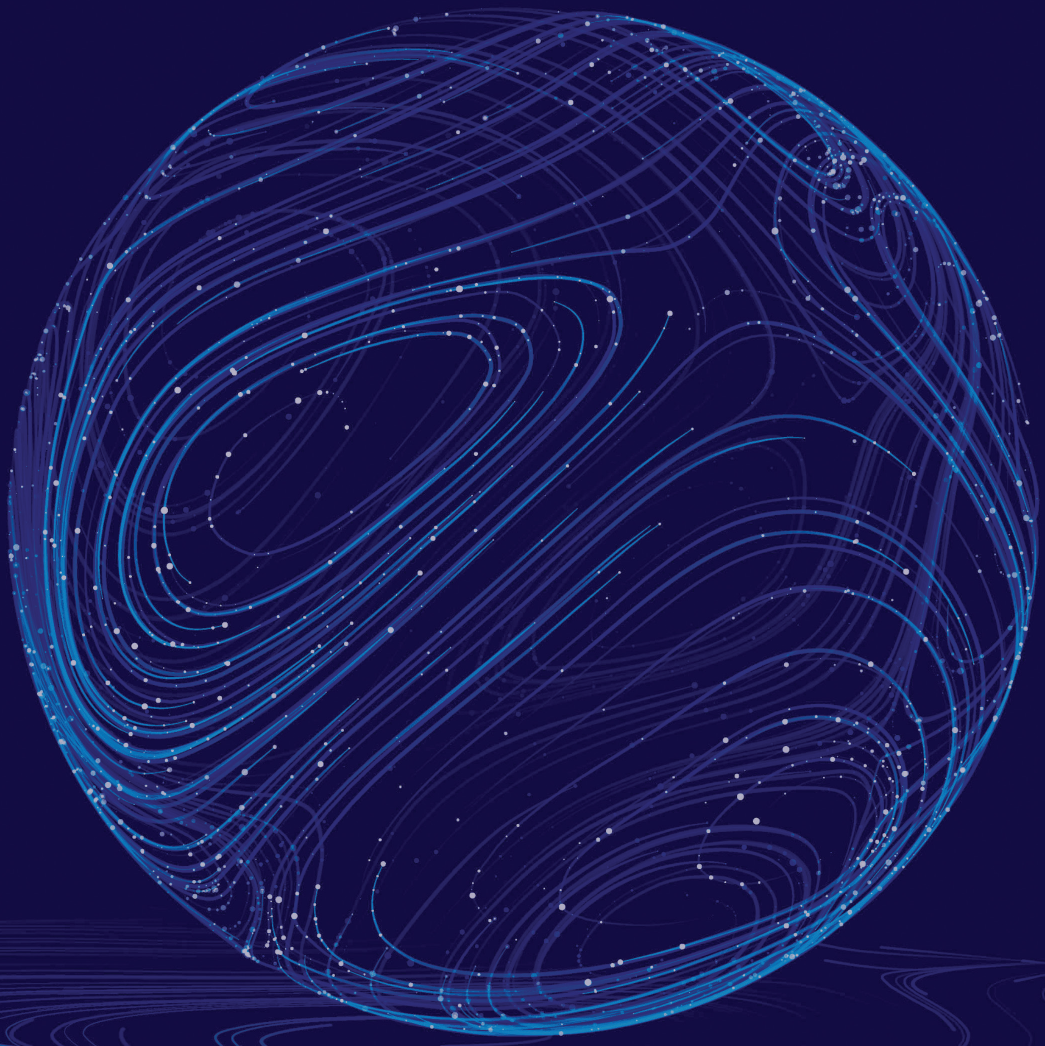
Kulpreet Singh

Hi-Tech Solutions

Avjeet Singh Bhatia

Technical Architect

incedo | Win in the Digital Age



About Incedo

Incedo is a digital transformation expert empowering companies to realize sustainable business impact from their digital investments. Our integrated services and platforms that connect strategy and execution, are built on the foundation of Design, AI, Data, and strong engineering capabilities blended with our deep domain expertise from digital natives.

With over 4,000 professionals in the US, Canada, Latin America, and India and a large, diverse portfolio of long term, Fortune 500 and fast-growing clients worldwide, we work across financial services, telecom, product engineering, and life sciences industries.

9+
Fortune 500 Customers

10+
Global Locations

4k+
Employees Globally

Our Global Presence

India
Gurugram
Chennai
Pune
Bengaluru
Hyderabad

USA
Santa Clara
New Jersey
Dallas
Boston

Canada
Ontario

Mexico
Guadalajara

R/17/03/2025

Follow us on:

www.incedoinc.com