1. [AMBITIOUS]

Multi-tenant architecture with single instance of application layer which will have separate database for each client. Expected the architecture and approach document as output.

**MULTITENANT ARCHITECTURE:**

Software multi-tenancy is the software architecture where one application instance serves multiple clients, which is common feature in SaaS platform. It allows multiple users in different environments to use the same set of applications, and ensures that the data between users is isolated from each other. Each client / customer is called as a tenant.

Software as a Service, also known as SaaS, is a cloud-based service where instead of downloading software at client’s server machines, the application is accessed via an internet browser.   
Most modern Software as a Service solutions are multi-tenant applications.

If a service provider sells an application to 10 customers and if there is a major upgrade to the application, then it needs redeployment / update of the same application in all 10 host machines. Multi-tenant architecture helps in handling such scenarios.

**Features of multi-tenancy:**

* Users and applications share a single, common infrastructure
* Code base is maintained centrally
* Saves deployment time

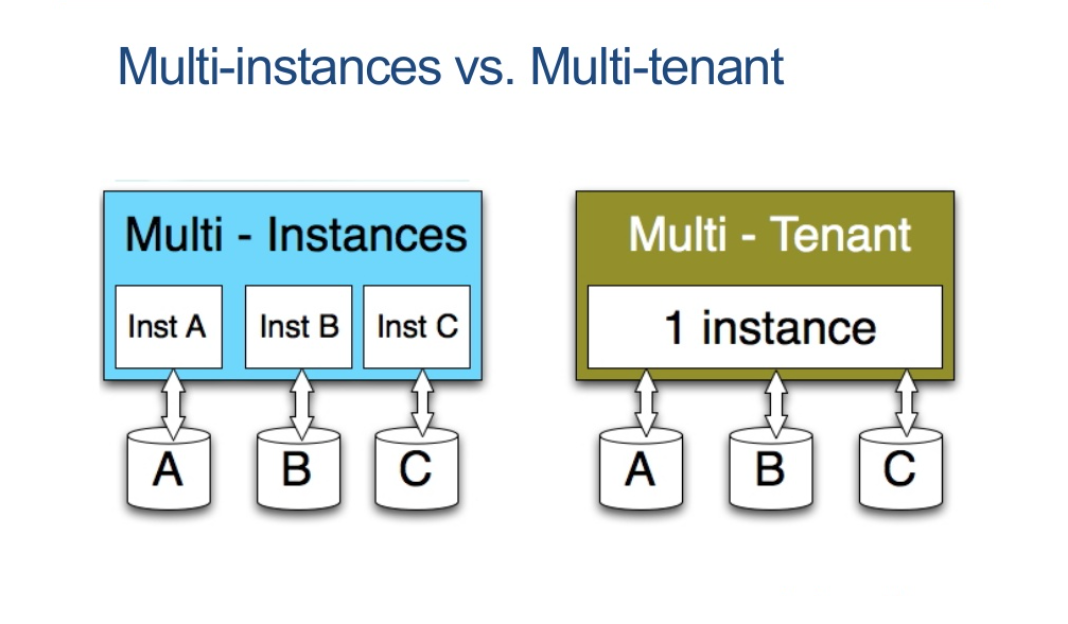
There are 3 approaches w.r.to multi-tenant implementation:

1. **Separate databases**: each tenant has its own database

2. **Separate schemas**: tenants share common database where each tenant has its own set of tables (schema)

3. **Shared schema**: data for all tenants is stored in the same tables and are identified through a tenant discriminator column.

Options 1 and 2 are less intrusive for the application and provide the highest level of isolation. In this sample POC, 2nd method - separate schema implementation is used.



**Identifying the tenant:**

Tenant identification is a main part while implementing multi-tenancy. There are standard ways to identify the tenant from the incoming requests.

Below 3 ways are commonly used:

1. Custom http request headers with tenant identifier key
2. Tenant ID in the URI as path variable or query param
3. Tenant ID in the host name

For the given requirement I have used Option 2 where the tenant id is passed in URI as query parameter. When a request comes from tenant1 then tenant1’s database / schema should be selected; when the request comes from tenant2 then tenant2’s database / schema should be selected and so on.

**GIT REPO:**

**HTTPS:** <https://github.com/tinaash/MultitenantSpringBoot.git>

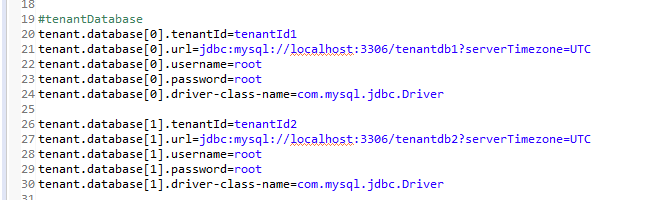
**SSH:** [git@github.com:tinaash/MultitenantSpringBoot.git](mailto:git@github.com:tinaash/MultitenantSpringBoot.git)

**Main classes in this implementation:**

This is a simple multitenant application to get the list of all users from different tenants. mySql DB is used in this POC but in general we can use any database engine, or we can even combine several approaches.

1. **DatabaseProperties.java:**

Reads below values from **application.properties** and maps the values to a static **Database** class:



This class contains **database** member variable which is List<Database> object:

**private** List<Database> database = **new** ArrayList<>();

It holds the custom Database objects mapped with values from application.properties file.

1. **DynamicDataSourceProvider.java**: Creates a list of datasource objects dynamically based on the database member variable from **DatabaseProperties.java**.

HashMap datasourceMap = **new** HashMap();

This datasourceMap contains tenantId as Key and datasource object of type DriverManagerDataSource as value. Each DriverManagerDataSource object contains a database driver, database URL, username and password.

The above two classes provide solution to dynamic creation of datasources. Instead of declaring a new Bean for mapping with each tenant datasource’s properties from configuration file, it simply creates a new dataBase object for each tenant. So while increasing number of tenants, only the properties file need to be updated with the new tenant’s DB details and no code changes are required.

1. **CustomRoutingDataSource.java:**

When any database operation is performed the control automatically comes here. This class extends AbstractRoutingDataSource class and overrides the determineCurrentLookupKey() method.

In this custom method, the tenantId request parameter is read from request URL using ServletRequestAttributes and that tenantId is returned as a String. Based on that string value, the tenant database will be selected by the main SpringBoot application in order to fulfill the requested operations.

1. **MultitenantSpringApplication.java:**

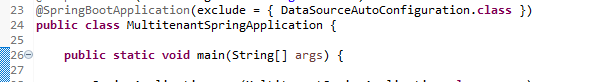
A datasource source bean

@Bean

**public** DataSource dataSource()

is defined in the main StringBoot application to return a CustomRoutingDataSource object

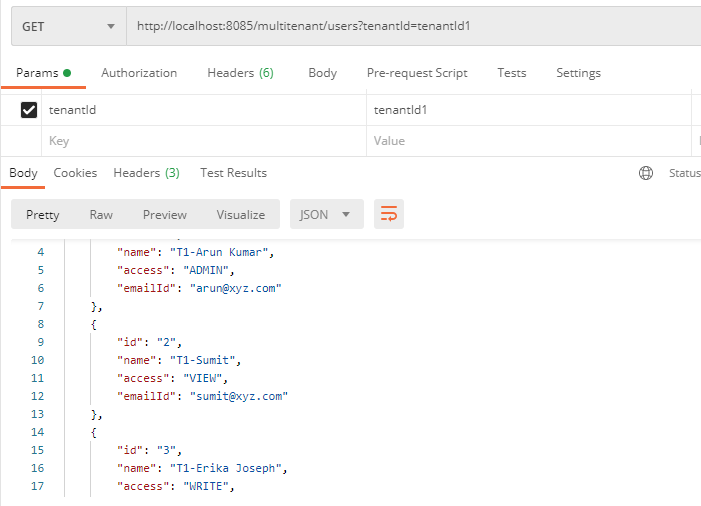
The default spring data source autoconfig is disabled for this SpringBoot app:



**TEST RESULTS:**

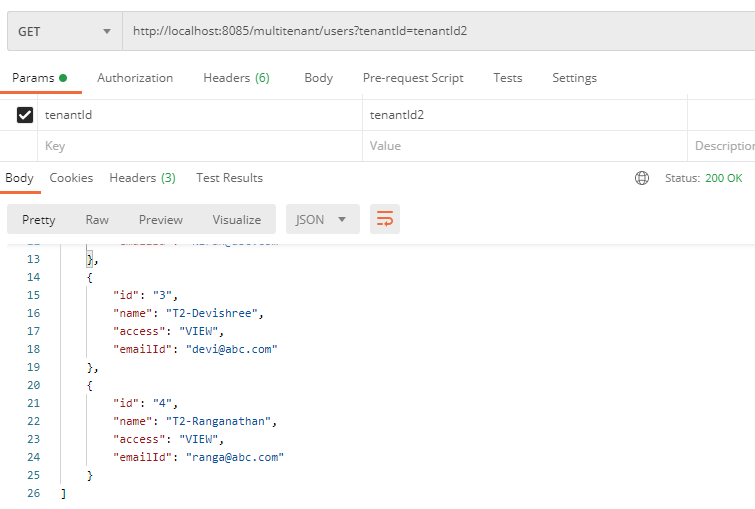
1. Get Users from tenant1

<http://localhost:8085/multitenant/users?tenantId=tenantId1>



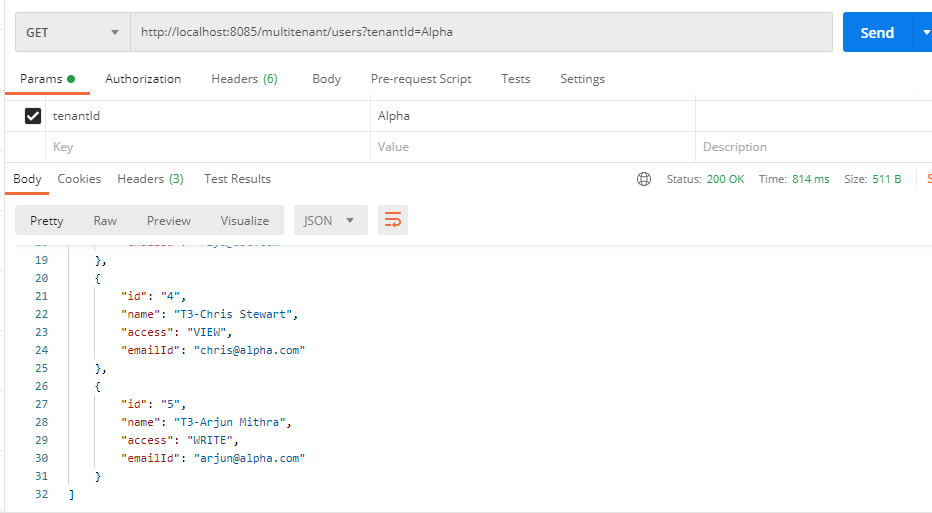
1. Get Users from tenant2

<http://localhost:8085/multitenant/users?tenantId=tenantId2>

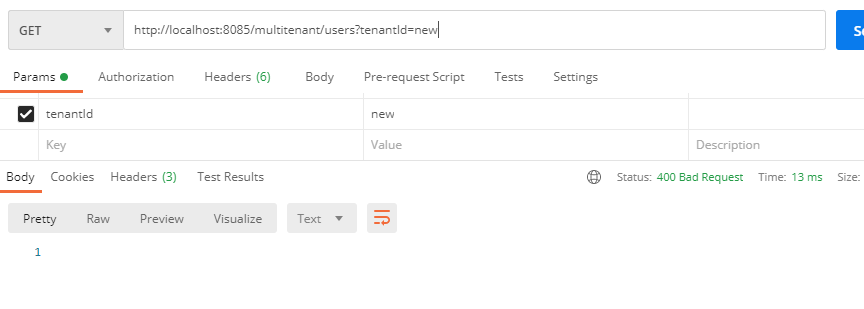


1. Get Users from tenant3, tenant3’s Id is configured as ‘Alpha’

<http://localhost:8085/multitenant/users?tenantId=Alpha>



1. For invalid or incorrect tenantId, 400 Error is returned in response:



This error response 400 Bad request is returned by handling the IllegalStateException in **UserController.java** class.