[New Guide] Download the 2018 Guide to Java: Features, Improvements, & Updates **Download Guide**

Multi-Tenancy Using JPA, Spring, and Hibernate (Part 1)

Learn how you can make your application act like multiple, independent apps by implementing multi-tenancy and keeping your data accessible by the tenants.

by Jose Manuel García Maestre MVB · Dec. 01, 16 · Java Zone · Tutorial

Download Microservices for Java Developers: A hands-on introduction to frameworks and containers. Brought to you in partnership with Red Hat.

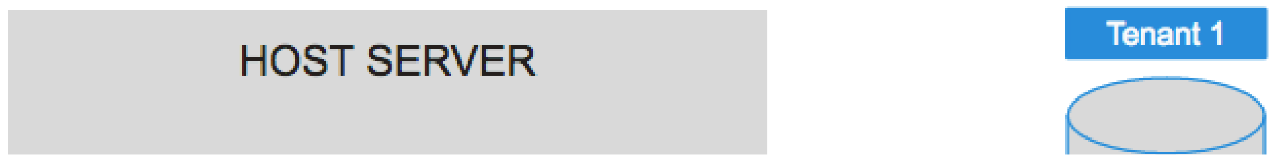
Multi-tenancy allows an application to behave as multiple independent applications hosted for different clients (i.e. organizations). This might not sound impressive, however as the number of clients increase it becomes more evident that it is easier and more cost effective to run a single application hosted for all the clients rather than hosting an independent application for each client.

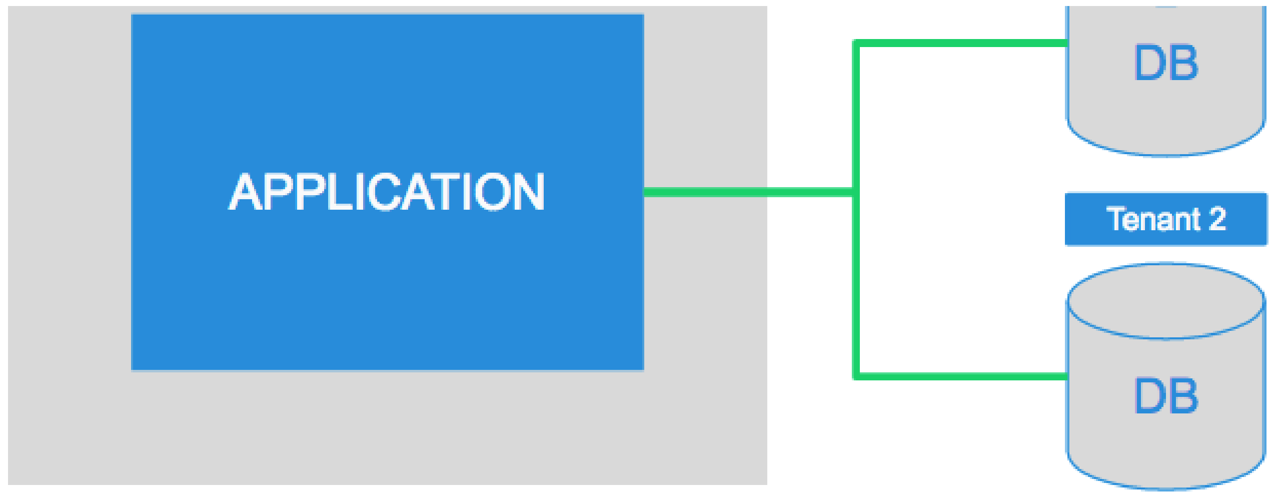
Multi-tenancy has become more popular lately and is very useful for the economy in software companies since it makes your service more affordable using cheaper installations/costs. This is because multi-tenancy can lead your business to a higher level of competitive differentiation.

How Does Multi-Tenancy Work?

Multi-tenancy allows a single application instance to be served for multiple tenants on a single hosting server. This is usually performed by either **separating databases**, **separating schemas**, or **sharing schemas**.

This architecture therefore allows for a single instance to service different companies.





Multi-tenancy works by using the concept of tenants, each tenant has access only to his corresponding data, even if they are in the same or different database. It means that you can have multiple tenants (usually one per client) who will use your application as if it were a single application. Amazing, isn't it?

Multi-Tenancy Implementation: Software Architecture

Let's start with the basic, multi-tenancy has tgree different ways to implement:

- **Separate databases**: Each tenant has its own database.
- **Separate schemas**: Tenants share common database but each tenant has its own set of tables (schema).
- **Shared schema**: Tenants share common schema and are distinguished by a tenant discriminator column.

No choice is better than another, every choice has different advantages and disadvantages, therefore it all boils down to what you want to compromise:

- Quantity of tenants
- Performance
- Time of development
- Reliability
- Disaster recovery
- And so on...

When you want to achieve a feature, it usually brings down another.

For instance: Separate-database is fully isolated, also it provides backup easily per tenant, however its performance isn't so good as shared schema when there are a lot of clients.

For more comprehensive details about Multi-tenant architecture I suggest the following MSDN article which delves more into the topic.

Multi-tenancy implementation

Hibernate, Spring, JPA

In this section we are going to use the Java specification for accessing POJOs to relational databases, that is the Java Persistent API (**JPA**).

As discussed earlier in the blog post, there are different approaches to implement Multi Tenancy, in this example we are going to use a separate database per tenant. In order to access the database we are going to use **Hibernate** which is a JPA Implementation.

We have chosen Hibernate because it is a well known ORM which has provided support for multi-tenancy implementation since version 4. To use Hibernate with your Java application all you need to do is to implement two interfaces: MultitenantConnectionProvider and CurrentTenantIdentifierResolver.

Your custom implementation of these classes will mainly differ based on your chosen multi-tenancy implementation architecture as described in the previous sections. However, it should be noted that Hibernate does not yet support the shared schema architecture (see HHH-6054).

Hibernate Sessions

Through the use of these sessions Hibernate is able to create transactions on entities. A session is opened and closed per transaction, when the session is created the tenantId is specified. Through the use of the tenantId, Hibernate can determine which resources to use, such that a tenant would access its (and only its) database.

In order to be able to create sessions, the SessionFactory class can be used. The SessionFactory needs to be provided with the tenantId so it is able to create a session appropriate to the database/schema that would be used by the tenant of the session being created. Below is an example of how a session is created:

```
Session session = sessionFactory
        .withOptions()
        .tenantIdentifier(yourTenantIdentifier)
        ...
        .openSession();
```

Note: Since we use JPA, we don't have to use the above code because Hibernate sessions are created automatically by JPA.

Note: Also, don't confuse Hibernate sessions with Http sessions, one is for database transactions, the other is for a user across more than one page request.

Implementation Details

MultitenantConnectionProvider and CurrentTenantIdentifierResolver

So now that we know how Hibernate works for multi-tenancy, some more questions can arise.

How does Hibernate know what database or schema to choose in multi-tenancy?

This is done through our custom implementation of MultitenantConnectionProvider. Our implementation provides a different connection by *tenantId*. This connection is then used for the Hibernate *session*.

How does Hibernate know which tenant to use?

For that we must implement CurrentTenantIdentifierResolver, this class has a method called resolveCurrentTenantIdentifier() that resolves the issue of which tenantId must be used by hibernate when a session is created.

Let's implement MultitenantConnectionProvider. We have the interface class MultitenantConnectionProvider which has the next implemented classes by hibernate:

- AbstractDataSourceBasedMultiTenantConnectionProviderImpl
- AbstractMultiTenantConnectionProvider
- DataSourceBasedMultiTenantConnectionProviderImpl

For our purpose, we want to provide connections to different databases through several DataSources. For that we could use a *map<TenantId, DataSource>*, in this way we'd get the *DataSource* by the *tenantId*. Also, we could extend DataSourceBasedMultiTenantConnectionProviderImpl but it uses jndi. As we will not use jndi in this post, we implement AbstractMultiTenantConnectionProvider instead:

```
/**
 * It gets the connection based on different datasources.
 */
public class MultiTenantConnectionProviderImpl extends AbstractDataSourceBasedMultiTenantConnectionProviderImpl
```

```
{

    Log logger = LogFactory.getLog(getClass());

    private static final long serialVersionUID = 14535345L;

    @Autowired
    private DataSource defaultDataSource;
    @Autowired
    private DataSourceLookup dataSourceLookup;

    /**
     * Select datasources in situations where not tenantId is used (e.g. startup processing).
    */
    @Override
    protected DataSource selectAnyDataSource() {
        logger.trace("Select any dataSource: " + defaultDataSource);
        return defaultDataSource;
    }

    /**
     * Obtains a DataSource based on tenantId
    */
    @Override
    protected DataSource selectDataSource(String tenantIdentifier) {
        DataSource ds = dataSourceLookup.getDataSource(tenantIdentifier);
        logger.trace("Select dataSource from "+ tenantIdentifier+ ": " + ds);
        return ds;
    }
}
```

DataSourceLookup bean has a map with different datasources mapped by his tenant id, we will write a new post about possible ways to implement this class and how we did it.

This will answer the question “**How does Hibernate/Spring know what tenants exist?**”

CurrentTenantIdentifierResolver can be implemented in different ways. In our case we decided to take the*tenantId* from an httpSession attribute. And this is used in throughout Hibernate in this interface.

```java
/**
 * It specify what Tenant should be use when the hibernate session is created.
 * @author jm
 */
public class CurrentTenantIdentifierResolverImpl implements CurrentTenantIdentifierResolver {

    Logger logger = Logger.getLogger(getClass());

    @Override
    public String resolveCurrentTenantIdentifier() {

        String tenant = resolveTenantByHttpSession();
        logger.trace("Tenant resolved: " + tenant);
        return tenant;
    }

    /**
     * Get tenantId in the session attribute KEY_TENANTID_SESSION
     * @return TenantId on KEY_TENANTID_SESSION
     */
    public String resolveTenantByHttpSession()
    {
        ServletRequestAttributes attr = (ServletRequestAttributes) RequestContextHolder.getRequestAttributes();
        //If session attribute exists returns tenantId saved on the session
        if(attr != null){
            HttpSession session = attr.getRequest().getSession(false); // true == allow create
            if(session != null){
                String tenant = (String) session.getAttribute(KEY_TENANTID_SESSION);
                if(tenant != null){
                    return tenant;
                }
            }
        }
        //otherwise return default tenant
        logger.trace("Tenant resolved in session is: " + DEFAULT_TENANTID);
        return DEFAULT_TENANTID;
    }

    @Override
    public boolean validateExistingCurrentSessions() {
        return true;
    }
}
```

How Do We Integrate Multi-Tenancy With JPA?

To integrate with JPA you must set the *jpaPropertyMap(Map<String, ?> properties)* in LocalContainerEntityManagerFactoryBean like this:

```
<beanclass="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
id="entityManagerFactory">
<property name="persistenceUnitManager" ref="pum"/>
<property name="jpaVendorAdapter">
<bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
<property name="databasePlatform" value="${database.dialect}" />
</bean>
</property>
<property name="jpaPropertyMap">
     <map>
          <entry key="hibernate.multi_tenant_connection_provider" value-ref="multitenancyConnectionProvider"/>
       <entry key="hibernate.tenant_identifier_resolver" value-ref="tenantResolver"/>
     <entry key="hibernate.multiTenancy" value="DATABASE"/>
     </map>
```

```
  </property>
 </bean>
```

If you are using Spring, this can be set in applicationContext.xml.

In future blog posts we will cover the following topics:

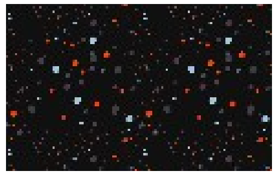
- DataSourceLookup implementation for multi-tenancy
- Hibernate schema export in multi-tenancy
- Different ways to save the tenant id in an http session

Finally, I want to share some useful links with you, they helped me to understand multi-tenancy in Hibernate and JPA.

- Hibernate's user guide to multi-tenancy
- Oracle's persistence package summary

Download Building Reactive Microservices in Java: Asynchronous and Event-Based Application Design. Brought to you in partnership with Red Hat.

Like This Article? Read More From DZone


Multi-Tenancy Using JPA, Spring, and Hibernate (Part 2)

Spring Tips: JPA [Video]

Hibernate Caching With Hazelcast: Basic Configuration

Free DZone Refcard
Getting Started With Kotlin

Topics: MULTITENANCY , SPRING , JPA , HIBERNATE , JAVA

Published at DZone with permission of Jose Manuel García Maestre , DZone MVB. See the original article here.
Opinions expressed by DZone contributors are their own.

Java Partner Resources

Advanced Linux Commands [Cheat Sheet]
Red Hat Developer Program

Microservices for Java Developers: A Hands-On Introduction to Frameworks & Containers
Red Hat Developer Program

Predictive Analytics + Big Data Quality: A Love Story
Melissa Data

Learn more about Kotlin
JetBrains