

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

If you want to support multi-tenancy with Spring in your stack, you should know how Spring actually knows the tenants, then dive right in.

by Jose Manuel García Maestre 🐞 MVB · Feb. 21, 17 · Java Zone · Tutorial

Verify, standardize, and correct the Big 4 + more— name, email, phone and global addresses – try our Data Quality APIs now at Melissa Developer Portal!

To understand how tenants work in Spring, it is important to first understand what multi-tenancy is and how it is implemented in JPA.

As I said in part 1, Hibernate/JPA knows the current tenant through the **CurrentTenantIdentifierResolver**, which is resolved by Spring. So the real question here is, “**How does Spring know what tenants exist?**” The answer to this question has a lot of different ways to be implemented, which is why it’s difficult to find a good reference.

In our case, before implementing it, we will set the basis to know how tenants are defined:

- Each tenant has a property file that describes values for a *dataSource* in the form: *database.<tenantId>.properties*. We will call it a tenant-file in this post.
- Each *<tenantId>* written in a filename is what identifies a tenantId. In this way, we don’t need to duplicate the information. If you need *tenantX*, just create a filename like *database.tenantX.properties*
- There is a *Map<String, DataSource>*, the key will be the tenantId string, this map will be initialized by getting the properties from the tenant-files.
- There is a default datasource called *defaultDatasource*. It contains default properties, so if a property is not specified in the tenant-file, we will get it from the default datasource. In this way, we can set a collection of properties as default and avoid property duplication.

As you can see from above, tenants are acquired from a tenant-file. Now let’s continue with our implementation.

Our Spring Implementation to Get Tenants

DataSourceLookup is a Spring interface that allows us to look up *DataSources* by name, we chose it since we want to find *DataSource* by *TenantId*. I found in the JavaDoc from Spring a class named **MapDataSourceLookup**, which has a **Map<String,DataSource>** and implements **DataSourceLookup**.

I decided to implement my own class so I could get the tenants from the properties file in a folder:

```
1  /**
2   * It lookup the correct datasource to use, we have one per tenant
3   *
4   * The tenant datasource has default properties from database.properties and
5   * also properties in database.{tenantId}.properties whose properties override
6   * the default ones.
7   *
8   * @author jm
9   *
10  */
11  @Component(value = "dataSourceLookup")
12  public class MultiTenantDataSourceLookup extends MapDataSourceLookup {
13
14      Log logger = LogFactory.getLog(getClass());
15
16      private String tenantDbConfigs = TENANT_DB_CONFIGS; // For testing
17      private String tenantDbConfigsOverride = TENANT_DB_CONFIGS_OVERRIDE; // For production
18      private String tenantRegex = TENANT_REGEX;
19
20      @Autowired
21      public MultiTenantDataSourceLookup(BoneCPDataSource defaultDataSource) {
22          super();
23
24          try {
25              initializeDataSources(defaultDataSource);
26          } catch (IOException e) {
27              e.printStackTrace();
28          }
```

```
29     }
30 }
31
32 /**
33  * It initialize all the datasources. If multitenancy is activated it also
34  * add datasources for different tenants on tenantDbConfigs or
35  * tenantDbConfigsOverride
36  *
37  * @param tenantResolver
38  * @throws IOException
39  */
40 private void initializeDataSources(BoneCPDataSource defaultDataSource) throws IOException {
41     //Get the path where server is stored, we will save configurations there,
42     //so if we redeploy it will not be deleted
43     String catalinaBase = System.getProperties().getProperty("catalina.base");
44
45     logger.info("MultiTenancy configuration: ");
46     logger.info("-----");
47
48     // Add the default tenant and datasource
49     addDataSource(DEFAULT_TENANTID, defaultDataSource);
50     logger.info("Configuring default tenant: DefaultTenant - Properties: " + defaultDataSource.toString());
51
52     // Add the other tenants
53     logger.info("-- CLASSPATH TENANTS --");
54     addTenantDataSources(defaultDataSource, tenantDbConfigs);
55     logger.info("-- GLOBAL TENANTS --");
56     addTenantDataSources(defaultDataSource, "file:" + catalinaBase + tenantDbConfigsOverride);
57     logger.info("-----");
58 }
59
60 /**
61  * Add Tenant datasources based on the default properties on
```

```

62  * defaultDataSource and the configurations in dbConfigs.
63  *
64  * @param defaultDataSource
65  * @param dbConfigs
66  */
67  private void addTenantDataSources(BoneCPDataSource defaultDataSource, String dbConfigs) {
68      // Add the custom tenants and datasources
69      Pattern tenantPattern = Pattern.compile(this.tenantRegex);
70      PathMatchingResourcePatternResolver fileResolver = new PathMatchingResourcePatternResolver();
71
72      InputStream dbProperties = null;
73
74      try {
75          Resource[] resources = fileResolver.getResources(dbConfigs);
76          for (Resource resource : resources) {
77              // load properties
78              Properties props = new Properties(defaultDataSource.getClientInfo());
79              dbProperties = resource.getInputStream();
80              props.load(dbProperties);
81
82              // Get tenantId using the filename and pattern
83              String tenantId = getTenantId(tenantPattern, resource.getFilename());
84
85              // Add new datasource with own configuration per tenant
86              -BoneCPDataSource customDataSource = createTenantDataSource(props, defaultDataSource);
87              addDataSource(tenantId, customDataSource); // It replace if tenantId was already there.
88
89              logger.info("Configured tenant: " + tenantId + " - Properties: " + customDataSource.toString());
90
91          }
92      } catch (FileNotFoundException fnfe) {
93          logger.warn("Not tenant configurations or path not found: " + fnfe.getMessage());
94      } catch (IOException ioe) {

```

```

94     , catch (IOException ioe) {
95         logger.error("Error getting the tenants: " + ioe.getMessage());
96     } finally {
97         if (dbProperties != null) {
98             try {
99                 dbProperties.close();
100             } catch (IOException e) {
101                 logger.error("Error closing a property tenant: " + dbProperties.toString());
102                 e.printStackTrace();
103             }
104         }
105     }
106 }
107
108 /**
109  * Create a datasource with tenant properties, if a property is not found in Properties
110  * it takes the property from the defaultDataSource
111  *
112  * @param defaultDataSource a default datasource
113  * @return a BoneCPDataSource based on tenant and default properties
114  */
115 private BoneCPDataSource createTenantDataSource(Properties tenantProps, BoneCPDataSource defaultDataSource)tenantProps)
116 {
117

```

```

7      BoneCPDataSource customDataSource = new BoneCPDataSource();
11
12
11      //url, username and password must be unique per tenant so there is not default value
9
12      customDataSource.setJdbcUrl(tenantProps.getProperty("database.url"));
0
12      customDataSource.setUsername(tenantProps.getProperty("database.username"));
1
12      customDataSource.setPassword(tenantProps.getProperty("database.password"));
2
12      //These has default values in defaultDataSource
3
12      customDataSource.setDriverClass(tenantProps.getProperty("database.driverClassName", defaultDataSource.getDriverClass()));
4
12      customDataSource.setIdleConnectionTestPeriodInMinutes(Long.valueOf(tenantProps.getProperty(
5
12          "database.idleConnectionTestPeriod",String.valueOf(defaultDataSource.getIdleConnectionTestPeriodInMinutes()))));
6
12      customDataSource.setIdleMaxAgeInMinutes(Long.valueOf(tenantProps.getProperty(
7
12          "database.idleMaxAge", String.valueOf(defaultDataSource.getIdleMaxAgeInMinutes()))));
8
12      customDataSource.setMaxConnectionsPerPartition(Integer.valueOf(tenantProps.getProperty(
9
13          "database.maxConnectionsPerPartition", String.valueOf(defaultDataSource.getMaxConnectionsPerPartition()))));
0
13      customDataSource.setMinConnectionsPerPartition(Integer.valueOf(tenantProps.getProperty(
1
13          "database.minConnectionsPerPartition", String.valueOf(defaultDataSource.getMinConnectionsPerPartition()))));
2
13      customDataSource.setPartitionCount(Integer.valueOf(tenantProps.getProperty(
3
13          "database.partitionCount", String.valueOf(defaultDataSource.getPartitionCount()))));
4
13      customDataSource.setAcquireIncrement(Integer.valueOf(tenantProps.getProperty(
5
13          "database.acquireIncrement", String.valueOf(defaultDataSource.getAcquireIncrement()))));
6
13      customDataSource.setStatementsCacheSize(Integer.valueOf(tenantProps.getProperty(
7

```

```
13         "database.statementsCacheSize",String.valueOf(defaultDataSource.getStatementCacheSize()))));
8
13
9     customDataSource.setReleaseHelperThreads(Integer.valueOf(tenantProps.getProperty(
14         "database.releaseHelperThreads", String.valueOf(defaultDataSource.getReleaseHelperThreads()))));customDataSource.setDriverClass(tenantProps.g
0
14
1
14
2     return customDataSource;
14
3 }
14
4
14
5 /**
14
6     * Get the tenantId from filename using the pattern
14
7     *
14
8     * @param tenantPattern
14
9     * @param filename
15
0     * @return tenantId
15
1     * @throws IOException
15
2     */
15
3 private String getTenantId(Pattern tenantPattern, String filename) throws IOException {
15
4     Matcher matcher = tenantPattern.matcher(filename);
15
5     boolean findMatch = matcher.matches();
15
6     if (!findMatch) {
15
7         throw new IOException("Error reading tenant name in the filename");
15
8     }
```

```
15         return matcher.group(1);
9
16     }
0
16
1
16
2 }
```

As there are comments in the code, I will focus just on the important parts.

Implementation Details

Firstly, we have some constants:

- *tenantDbConfigs*: The directory used in development where tenant-files are located inside a folder of the project.
- *tenantDbConfigsOverride*: The directory in production where tenant-files are located outside the project. In this way, if the application is redeployed it's not deleted..
- *tenantRegex*: Regular expression that we use to iterate about different tenant-files and get the tenantId.

It is in the constructor where we initialize the *datasources*, the method **initializeDataSources(DataSource defaultDataSource)** will put the *defaultDataSource* for the default tenant in the map firstly and then the others.

All the magic to get the tenants happens in **addTenantDataSources(DataSource defaultDatasource, String folder)**. Basically, what we do is to use a regular expression to get the properties from each tenant in a folder. We also get the *tenantId* from the filename and override the *DefaultDataSource* properties with the ones in the property file of each *tenantId*. Finally, we add the entry to the map as *<tenantId, DataSource>*.

In this class we have two different processes:

- Add the datasources for each tenant. These datasources will be used for Hibernate.
- Get the properties for each tenant

The ideal should be to separate these processes: “Add datasources” and “get tenant properties.” This can be done by creating a new singleton class, *PropertyManager*, and using it in our *MultiTenantDataSourceLookup*. In this way, you could have every access to any property centralized in the *PropertyManager*. But for the sake of shortening and understanding this topic, I decided to post it in this way.

In future posts, we will talk about:

- Hibernate schema export in multi-tenancy.
- Different ways to save the tenant id in an HTTP session.

I hope that you find the article informative. If you have any questions please don't hesitate to ask.

Developers! Quickly and easily gain access to the tools and information you need! Explore, test and combine our data quality APIs at **Melissa Developer Portal** – home to tools that save time and boost revenue. Our APIs verify, standardize, and correct the Big 4 + more – name, email, phone and global addresses – to ensure accurate delivery, prevent blacklisting and identify risks in real-time.

Like This Article? Read More From DZone



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



Spring Boot + JPA + Hibernate + Oracle




Spring Tips: JPA [Video]



**Free DZone Refcard
Getting Started With Kotlin**

Topics: HIBERNATE , JPA , MULTITENANCY , SPRING , TUTORIAL , 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

