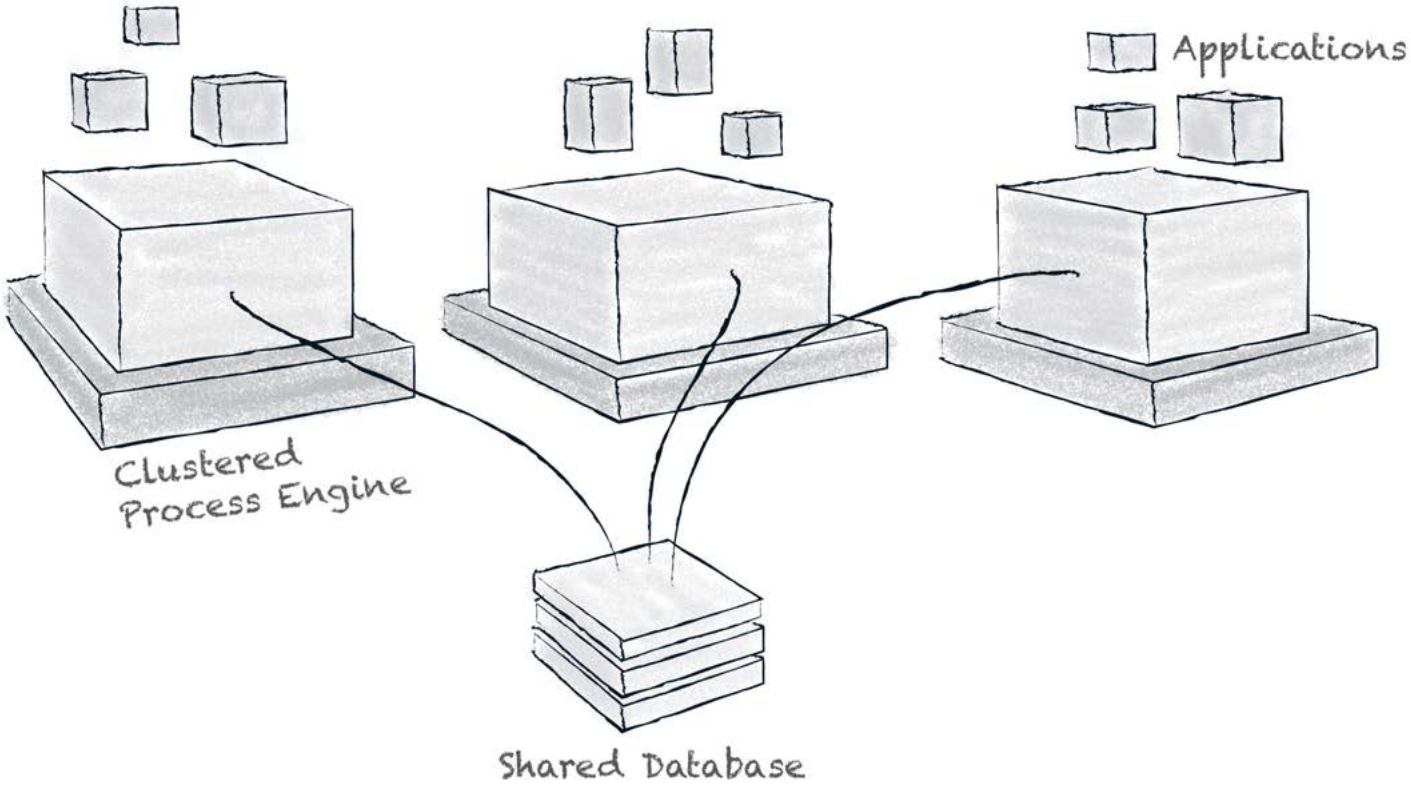
camunda
the business process company

Multi-Tenancy



Contents

What is Multi-Tenancy? 2

Multi-Tenancy Use Case 2

Data Isolation vs. Resource Consumption 2

Tenant Identifiers 2

Server per Tenant 4

Schema per Tenant 5

Tenant Management in Cockpit 7

Conclusion 9

Multi-Tenancy

What is Multi-Tenancy?

Multi-Tenancy is where the same application is run as multiple incarnations or tenants on the same server. That server may be catering for differently defined instances of that application.

With regard to Camunda specifically, this document will be discussing how a single Camunda installation serves more than one tenant. The document intends to clarify what options are available for someone who wants to implement multi-tenancy with Camunda as well discussing the pros and cons of each approach.

Multi-Tenancy Use Case

First we should discuss why someone might come to implement multi-tenancy in Camunda. The most common use case is where the Camunda engine is driving business processes for a SaaS application. In this case each tenant is operated on by a completely different client. When a client logs into their tenant they'll of course only be expecting to see their own data and functions. Nothing a client can do in their tenant should ever be able to interfere with any other tenant and so data isolation is paramount.

So why is multi-tenancy so useful in a case like this? Scalability. With multi-tenancy, granting a new client access to a tenant has the lowest cost and quickest implementation time. The alternative is to deploy and maintain a brand new server, database and software installation each time a new client joins.

Data Isolation vs. Resource Consumption

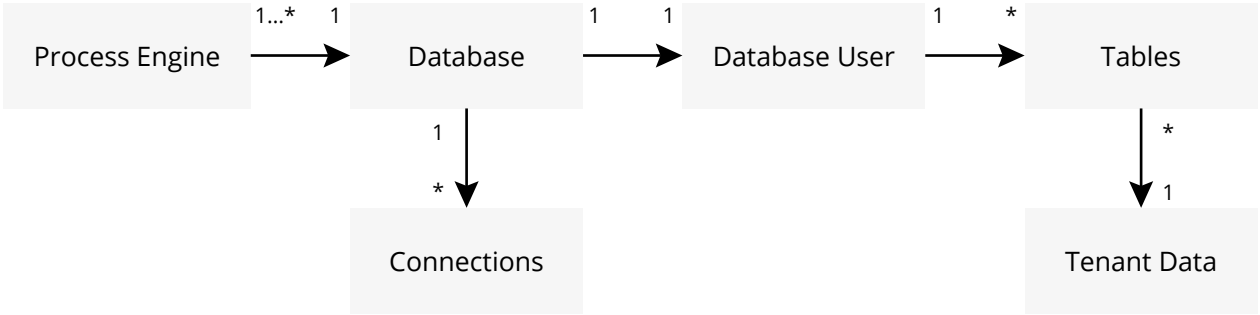
Each type of multi-tenancy implementation walks a line between data isolation and resource consumption. To have the maximum possible data isolation (i.e. a single database per tenant) requires the use of a lot of resources and is generally quite wasteful of those resources – and so this is often avoided. On the other end of the spectrum is of course perfectly optimized resource consumption (i.e. all tenants sharing the same database and tables) but in this case all it takes is a lazily written database query to completely break down the data isolation between tenants.

The ideal scenario is of course to have the most robust data isolation with the fewest possible wasted resources.

Tenant Identifiers

Starting on the scale of least data isolation is Tenant Identifiers. This means that all tenants run separately but write data to exactly the same tables. Each running tenant requires a unique marker in order to gain access to their tenant specific data. This means that all queries and ta-

bles must have this tenant identifier added or data isolation will be lost. Camunda introduced out-of-the-box support for this approach with version 7.5 of the product by adding the tenant ID to all relevant database tables as well as to internal queries.



There are some advantages and disadvantages to this approach. The main advantage is that because the data for all tenants is stored on the same tables, it makes writing queries for data across a number of tenants quite easy. This can be useful if reporting and analytics on your tenants is a strong focus. It is also the most resource-efficient implementation of multi-tenancy.

To deploy definitions for a single tenant, the tenant identifier has to be set on the deployment. If no tenant identifier is set, then the deployment and its definitions belong to all tenants. In this case, all tenants can access the deployment and the definitions and usually a new process instance is associated with a specific tenant when it is started by one of the tenant's users. That means that working with tenant identifiers enables using shared definitions, which in turn reduces a lot of deployment and operations overhead in such scenarios. One of the new options that come with shared definitions include scenarios where a service provider might want to deploy a shared process definition for all tenants which then calls tenant-specific DMN tables or call activities. Another option could be using a shared definition for the majority of the tenants and still being able to provide specific variants of that definition to single tenants. The easiest way of deploying a definition to a specific tenant is to add the tenant ID to deployment descriptor of your process application (processes.xml):

```
<?xml version="1.0" encoding="UTF-8"?>
<process-application
    xmlns="http://www.camunda.org/schema/1.0/ProcessApplication"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    <process-archive tenantId="tenant-one">
        <process-engine>default</process-engine>
        <properties>
            <property name="isDeleteUponUndeploy">false</property>
            <property name="isScanForProcessDefinitions">true</property>
        </properties>
    </process-archive>

</process-application>
```

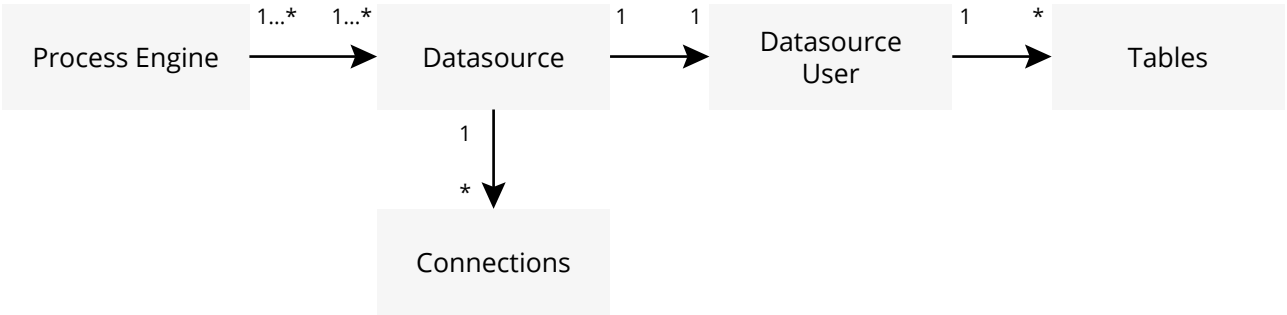
The disadvantages of this approach relate to data isolation and query performance. Because each tenant's data is only distinguished by a marker, that identifier must be included in every query performed by the process. Because it can be cumbersome to pass the tenant ID to every Camunda API call, we use transparent access restrictions for tenants, which allow to omit the tenant ID in queries once an authenticated user has been set. In order to achieve that, a list of tenant IDs needs to be provided when setting the authentication programmatically. Alternatively, the authentication provided by the REST API or by Camunda's web apps can be used.

It's also possible that performance could suffer with the introduction of tenant markers especially when querying process variables. For internal queries, performance has been highly optimized by our developers, but that aspect has to be kept in mind when writing own queries.

Probably the biggest disadvantage with this approach is the risk of disclosing data that belongs to other tenants. Bugs or careless application programming is all it would take for data to be returned to the wrong tenant.

Server per Tenant

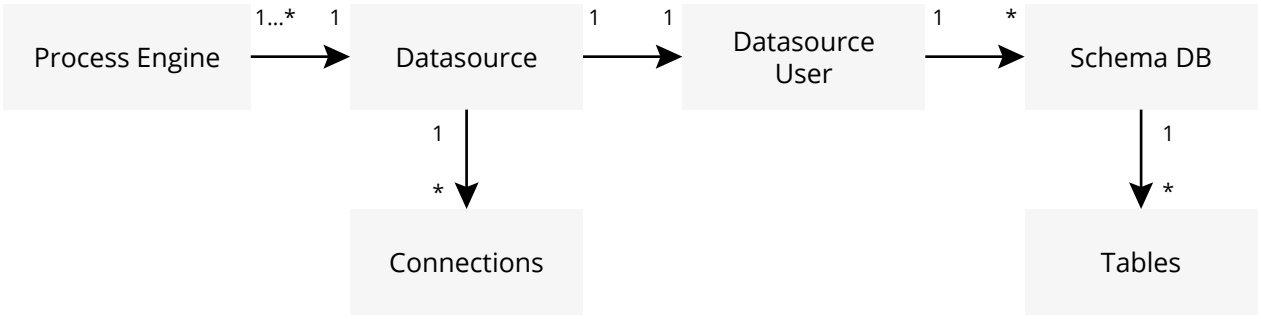
While not often considered as anyone's first choice, setting up a server per tenant is sometimes used for complete data isolation. The tenant data is stored in its own database, without access to or knowledge of the existence of any other tenants. Data isolation by this method can sometimes be considered a little overkill. Often it's existing restrictions by database administrators that might force a business to use this approach.



There are a few disadvantages to server per tenant. One is that querying data for more than one tenant at a time involves a lot of work because the query has to cross multiple databases. More concerning of course is the waste of database resources. This means that in a case where one tenant is a heavy user and another a light user they are given the same number of database connections. Some might go unused while the others could be maxed out. Considering that a company like Oracle charge customers by database connection this type of inefficiency is not trivial.

Schema per Tenant

The Schema per Tenant architecture is the recommended approach when it comes to multi-tenancy implementation in Camunda for scenarios where a higher level of data isolation is required. With this approach, each tenant has its own process engine and each process engine shares the same database, but not the same tables. This is achieved by creating a number of schemas within a single data source. Each tenant connects to its own schema. The result is that data isolation is guaranteed by the process engine and at the same time database resources can be shared between the various tenants.



We can say that data isolation is guaranteed because of how the process engine's internal architecture implements queries. Every query made by the process engine begins with a database prefix. In the code snippet below you can see the variable prefix before the name of the table. If a tenant has been set up for this process engine it would be added to the query automatically. If there is no tenant the prefix variable is empty and will simply query the database tables directly.

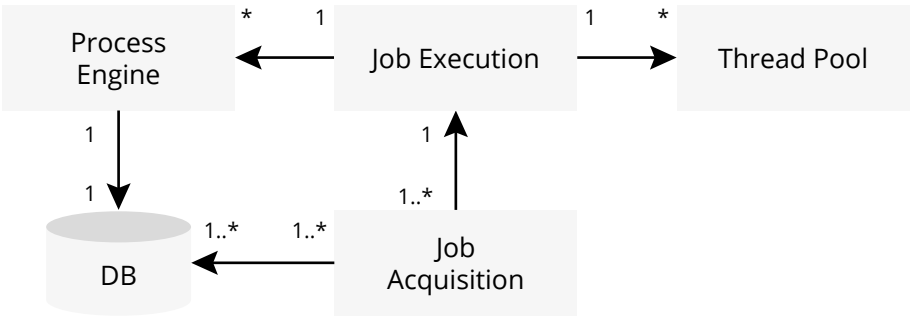
```
select * from ${prefix}ACT_RU_VARIABLE where NAME_= 'customerId'
```

There are a few options for setting up Camunda to run multiple engines and it can actually be quite straight forward. One of the easiest ways is to simply add additional process engine descriptions to the Camunda configuration file (bpm-platform.xml) as demonstrated below.

```
<?xml version="1.0" encoding="UTF-8"?>
<bpm-platform xmlns="http://www.camunda.org/schema/1.0/BpmPlatform" xmlns:xsi="http://www.
w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.camunda.org/schema/1.0/BpmPlatform http://www.camunda.
org/schema/1.0/BpmPlatform">

    <job-executor>
        <job-acquisition name="default" />
    </job-executor>

    <process-engine name="tenant-one">
        <job-acquisition>default</job-acquisition>
        <configuration>org.camunda.bpm.engine.impl.cfg.StandaloneProcessEngineConfiguration
        </configuration>
        <datasource>java:jdbc/ProcessEngine</datasource>

        <properties>
            <property name="databaseTablePrefix">tenant_1.</property>
            <property name="databaseSchema">tenant_1</property>
            <property name="history">full</property>
            <property name="databaseSchemaUpdate">false</property>
            <property name="authorizationEnabled">true</property>
            <property name="jobExecutorDeploymentAware">true</property>
        </properties>
    </process-engine>

    <process-engine name="tenant-two">
        <job-acquisition>default</job-acquisition>
        <configuration>org.camunda.bpm.engine.impl.cfg.StandaloneProcessEngineConfiguration
        </configuration>
        <datasource>java:jdbc/ProcessEngine</datasource>

        <properties>
            <property name="databaseTablePrefix">tenant_2.</property>
            <property name="databaseSchema">tenant_2</property>
            <property name="history">full</property>
            <property name="databaseSchemaUpdate">false</property>
            <property name="authorizationEnabled">true</property>
            <property name="jobExecutorDeploymentAware">true</property>
        </properties>

    </process-engine>

</bpm-platform>
```

In the code snippet above you can see process engines “tenant1” and “tenant2” defined. Each has their own databaseTablePrefix as discussed above and both are pointing to the same data source which will contain their individual schemas. Note that you also need to specify the schema with databaseSchema.

Another important mechanism to understand is the job executor. This is fundamental to how resources are efficiently managed between tenants. In the same code snippet, you’ll notice the job executor defined near the top with a job acquisition element called “default”, which both tenants are linked to through the <job-acquisition> tag. Each tenant can have its own Job Acquisition thread while all tenants share the same Job Executor thread. But what does all this mean?

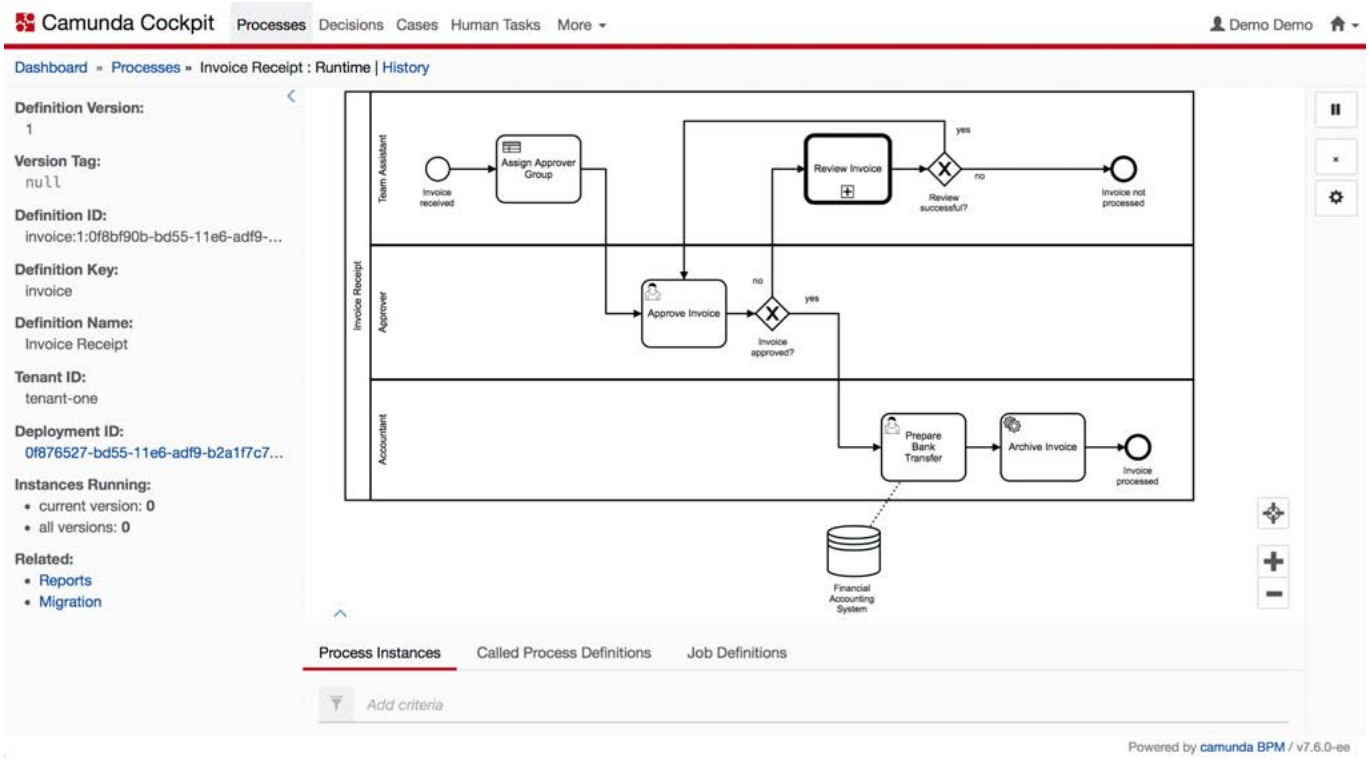


As described in the diagram above, when a process engine requires a job executed it writes it to a table on its database. One or more job acquisition processes can be running and they can pick up jobs from the table. It then passes the job to the executor. There is one job executor, which is in contact with all tenants, it has control over the thread pool and so picks a thread to run the job for the process engine.

What this achieves is that all threads are available equally to all tenants. Tenants that require a lot of threads are not going to be slowed by a per-process engine allocation. While tenants that aren't experiencing such high usage don't have a pool of unused threads sitting idle.

Tenant Management in Cockpit

Once you have a Camunda server running multiple tenants, it might be interesting to see how those tenants are managed from a user point of view. For this we can take a look at Camunda Cockpit and Tasklist.



The Camunda Cockpit - if you don't already know – is a web application deployed with the Camunda server where you can monitor and administrate your running process instances. There are a variety of interesting features available, but for the purposes of this document it's only necessary to describe how it deals with multiple tenants.

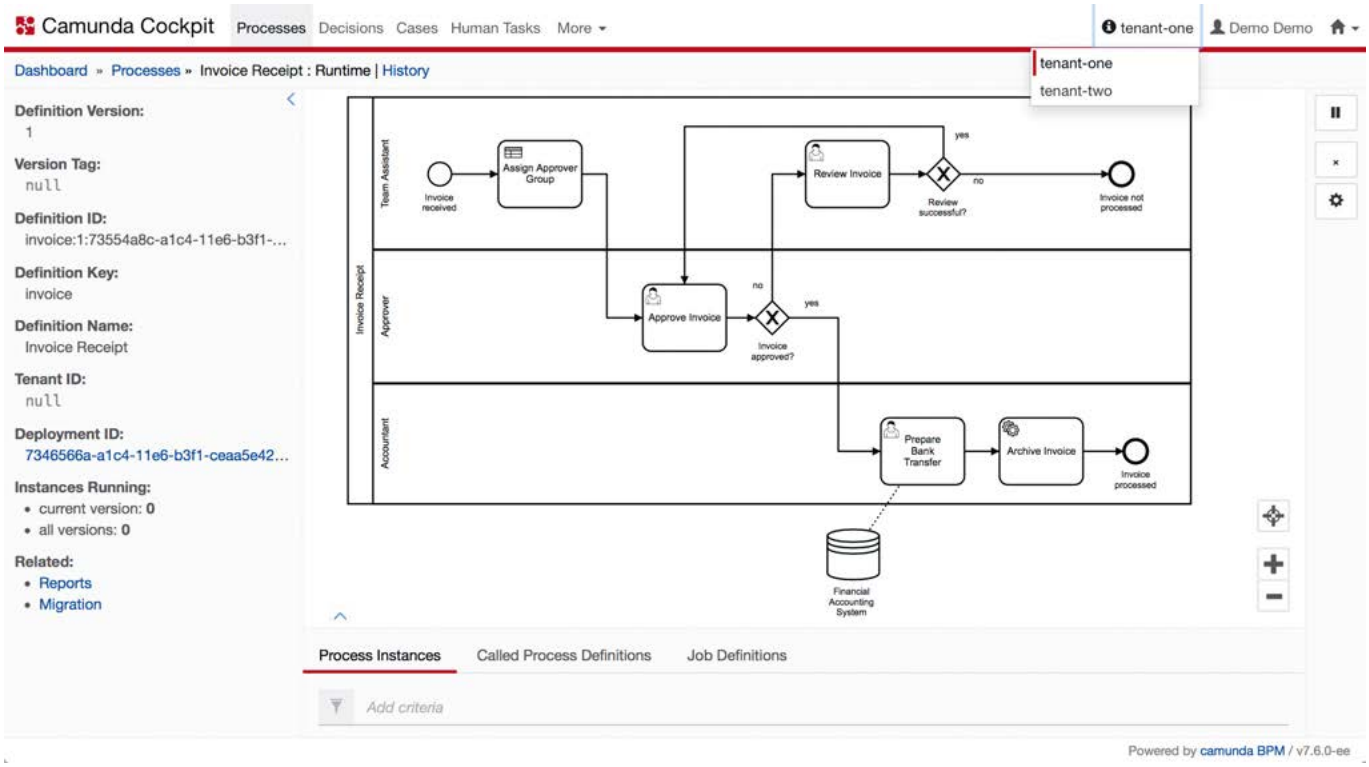
Tenant Identifiers

You can see from the screenshot that a Tenant ID is shown whenever a definition has been deployed or an instance has been started for a specific tenant. This way it is easy to navigate within Cockpit, especially for users that belong to more than one tenant.

For users that administer multiple tenants, there is no need to switch between different views in cockpit since all data is stored in the same database tables. Of course, it is possible to search and filter by tenant when working with definitions and instances.

Reporting can be handled within Cockpit the same way it would be done for a setup without tenants.

Schema per Tenant



You can see from the screenshot that when multiple engines are present, a dropdown is available to a cockpit user allowing them switch between them. So if you decide for the Schema per Tenant approach, you will be able to navigate to the tenant's cockpit this way.

The functionality is also available for users starting up or working on process instances via the Camunda Tasklist. In that case the same dropdown menu will be visible, making it easy for users with access to multiple engines to seamlessly switch between them.

When it comes to reporting, Camunda provides access to its history tables through an API as well as direct database queries. The goal of which is to allow seamless integration with any existing reporting infrastructure.

For multi-tenancy reporting specifically, where no existing software is in place, we would suggest looking into implementing a Business Intelligence (BI) solution. The benefit of which is that the data from all tenants is uploaded independently to a single location and manipulated within the BI itself. The alternative, if setting up a BI solution isn't appealing, is writing queries directly to the history tables that would combine the data from various tenants. It should be noted however that these queries would require a good deal of work to ensure they have optimal performance.

Conclusion

Multi-Tenancy is well supported by Camunda, but obviously some approaches are more beneficial than others. The approach you choose will of course come down to the needs and restrictions of your own infrastructure. The table below should be able to help you understand the choices available at a glance. Of course, they can also be combined.

#	Data Isolation	Resource Consumption	Multi-Tenant Option
1	Very Important	Unimportant	Server per Tenant
2	Unimportant	Very Important	Tenant Markers
3	Important	Important	Schema per Tenant

For more on multi-tenancy as well as examples and additional documentation visit http://docs.camunda.org/.

Imprint

Europe / Asia

Camunda Services GmbH
Zossener Str. 55
10961 Berlin
Germany

Phone: +49 (0) 30 664 04 09 - 00
E-Mail: info@camunda.com
www.camunda.de

America

Camunda Inc.
44 Montgomery St, Suite 400
San Francisco, CA 94104
USA

Phone: +1.415.548.0166
E-Mail: info@camunda.com
www.camunda.com