Performance Optimization and Scalability Tips

Identify Performance Bottlenecks

The saying from renowned computer scientist Donald Knuth goes:

> *"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%."*

Before wasting any effort on optimization, it is important to identify where the specific problems or inefficiencies are first.

1. Identify elements in your userview pages and menu that requires significant amounts of SQL or BeanShell scripts. These are the common causes of performance issues.
2. Use the Performance Analyzer to help identify the slow elements in your app
3. Identify slow SQL queries in your app by leveraging on your database e.g. for MySQL you can use the Slow Query Log
4. Check your custom plugins to ensure that they do not introduce slow processing, database queries, etc.
5. Identify whether there are sufficient resources (physical memory, storage space, Java VM allocation, etc) to handle your app usage, and whether there are any resource leakages.

Once you have identified where the potential problems are, you can then start to look into optimizing each of the areas, described in the following sections.

Ensure Sufficient Java VM Memory Allocation

The platform runs on the Java VM (JVM), and depending on the size and complexity of your apps and usage, you might need to tune the JVM heap memory size to suit your environment. If the setting is too low, the system will run out of memory resulting in OutOfMemory errors. However if the setting is too high compared the amount of physical RAM available, there might be quite a lot of swapping, in addition to overheads in garbage collection.

The default JVM settings are quite low to support installation on a user's individual computer for testing or development. To get an optimum setting might require a bit of trial and error sometimes, depending on the usage environment.

You can use tools like VisualVM to monitor the memory usage as per Monitoring using VisualVM.

Some additional details are available at Deployment Best Practices#JavaVMConfiguration.

Eliminate Resource Leakage (Memory, Database Connections, Files, Network, etc)

At the platform level, Joget Workflow has been optimized and tested to ensure that there are no resource leakages.

However, it is very possible to introduce these problems into your app. For example, you might be creating and using database connections without closing them. This would cause your database connection pool to be exhausted, and eventually cause the system to grind to a halt. When using custom Java or JDBC code in a BeanShell script or custom plugin, database connections must be closed in a try-catch finally loop e.g.

```
Connection con = null;
try {
   // get connection
   con = datasource.getConnection();

   // more processing
} finally {
   // close connection
   try {
       if (con != null) {
           con.close();
       }
   } catch (Exception e) {
   }
}
```

Likewise, any custom code that creates objects, network connections or file IO must release such resources in a try-finally block. An indication of unreleased memory would manifest itself in the Java VM memory usage, which would continually increase without reducing over time.

Using a tool like VisualVM, you will be able to monitor the memory usage.

The database connection usage can be monitored in your database server, or use the v6 Database Connection Monitoring and Leak Detection feature.

Optimize your App and Database Queries

Joget Workflow has been thoroughly profiled and optimized to ensure that there is minimal overhead at the platform level. However, in enterprise apps that rely on dynamic data, each page request would require many database queries. Database calls are slow not just in query execution, but especially in network I/O. In most cases, database calls are the main performance bottlenecks that cause lengthy page response times and limit scalability.

It is therefore significant to minimize the number of database calls that are made in the app. Check through your app to ensure that any database calls (or processing) are really necessary, if not remove them.

This also applies not just to the main userview page, but also in the userview menus. For example, displaying a count of items in the menus incur a potentially slow database query, so you need to make a decision on whether or not to display it. You can test the performance of a page with and without menus by using the Userview Embedded Mode.

In the upcoming v6, there will be a new Userview Caching feature to optionally cache userview pages and menus for a significant performance boost.

Tune the Database and Application Server

You should optimize custom SQL queries that are used in your app, as slow queries are very common bottlenecks. In MySQL for example, you can use the EXPLAIN command to help you determine the execution plan of your queries, and find ways to optimize them.

For apps that require a lot of database queries, you would want to optimize both your database and database server configuration. Depending on the type of database server used, your database administrator would be best placed to perform such tuning based on your usage pattern. For example, MySQL has a section on optimization in the documentation http://dev.mysql.com/doc/refman/5.6/en/optimization.html.

There are some additional information at Deployment Best Practices#DatabaseConfiguration.

If there are a lot of concurrent requests to your app, it might also be prudent to tune your application server as per Deployment Best Practices#WebApplicationServerConfiguration.

Do update your database server to the latest MySQL releases. New stable database engine releases usually have performance improvments.

If you use a lot of form grids with parent and child database tables, do add indexes to your foreign key child table. This will speed up retrieval of the child form records.

Load Test your App and Server Sizing

There are many factors involved in determining the server specifications needed to run Joget Workflow effectively. These are some of the non-conclusive factors:

1. Total number of users
2. Maximum expected concurrent users
3. Number of apps running on the platform
4. Complexity of each of the apps
5. Amount of data generated in each app
6. Network infrastructure

For example, an environment with a small number of users running a heavily-used complex app might require more resources than an environment with large number of users running some simple apps only once a day.

So based only on number of maximum and concurrent users, you wouldn't be able to predict what is needed. Usually installations would start with a single application server first, and scale up or cluster when necessary.

To determine the actual requirement for your app and usage, it is best to perform load testing in your environment. There are many free and commercial load testing available, there is an article using the open source Apache JMeter tool at Joget Workflow Clustering and Performance Testing on Amazon Web Services (AWS)

Introduce Clustering and Load Balancing

When you have optimized your apps, you can increase your server capacity as appropriate to handle increasing load (vertical scaling). You can also start to consider performing horizontal scaling i.e. to cluster or load balance your installation. This can not only handle increased load, but also offer high availability. Read more on this at Deployment Best Practices#ClusteringandLoadBalancing.