Hello, Coherence Community Edition, Part 3: Packaging, deployment, scaling, persistence, and operations with Java

How to push a Coherence CE application out to a Kubernetes cluster, and then scale and manage that cluster

*by Aleks Seović*

January 15, 2021

Oracle Coherence started as a distributed caching product and then evolved into an in-memory data grid. It’s an essential tool for improving the performance and scalability of Java EE applications, and it’s widely used for large-scale projects—think of it as a scalable, concurrent, fault-tolerant implementation that is partitioned across multiple JVMs, machines, and even data centers.

In the summer of 2020, Oracle released Coherence Community Edition (CE), an open source version of the product.

In the first article in this series, I implemented a REST API that allows you to manage a to-do list of tasks stored in Coherence CE. In the second article, I built a React-based web front end and JavaFX-based desktop client.

In this part, I’ll complete the series by covering the packaging, deployment, and operational aspects of this project. That includes converting the existing demo into a production-quality application by adding support for scale out, persistence, monitoring, and end-to-end request tracing. **Figure 1** shows the UI for the application.
Creating a Docker image

To deploy the application to Kubernetes, first create a Docker image for it. Oracle provides prebuilt Docker images for the latest Coherence CE versions, which can be downloaded from the GitHub Docker Registry. However, these images are primarily useful for testing and quick demos that require only the basic functionality and do not use any custom server-side code. For everything else, including the to-do list application I’ve been working on, you need to create a custom Docker image.

It’s important to remember that Coherence CE is just a library and is embedded into your Java SE application or microservice the same way any of the popular lightweight frameworks such as Spring Boot, Micronaut, or Helidon are. In that way, Coherence CE’s data management services are similar to the embedded HTTP and gRPC services these lightweight application frameworks allow you to run. In other words, there is no separate server to run that your application then connects to: Your application is the Coherence CE cluster member.

While there are several ways to build a Docker image for a lightweight Java SE application and some frameworks, such as Spring Boot, even provide their own tooling for that purpose, the easiest way to build an image for a generic Java application I have used so far is the Jib Maven plugin. It gets the job done by packaging all the necessary dependencies based on the contents of your POM file and doesn’t require that you write any Dockerfiles or startup scripts. The plugin also packages various types of dependencies (external and project) into different image layers, reducing image size and build time in the process. It is also a perfect fit for Helidon/Coherence CE applications, such as the one I’ve created, so that’s what I will use.

The first step is to configure the Jib plugin within the POM file. I will do this within a Maven profile that needs to be explicitly activated to build a Docker image only when I want to, not during each Maven build.
The configuration above uses the "distroless" Java 11 image as a base image and creates the image with the same name and version as the Maven build artifact (todo-list-server, in this case). The configuration also exposes ports 1408 (for gRPC) and 7001 (for REST endpoints the application uses).

Now that the Jib plugin is configured, building a Docker image locally is a simple two-step process. First, you need to build the application and install its artifacts into a local Maven repository by running the following command:

$ mvn clean install

Once that's done, you can build the image by running this command:

$ mvn package -Pdocker
Important warning: Be sure you have the Docker daemon running locally; otherwise, the step above will fail.

If everything goes well, you should see output similar to the following in your Maven logs:

```
[INFO] --- jib-maven-plugin:2.6.0:dockerBuild
[INFO] Containerizing application to Docker daemon
[INFO] Using base image with digest: sha256:base
[INFO] Container entrypoint set to [java, -cp
[INFO] Built image to Docker daemon as todo-list-server:latest
```

You can also verify that the image has been created by executing this command:

```
$ docker images | grep todo-list-server
```

The output should look similar to the following:

```
todo-list-server 1.0.0-SNAPSHOT 584a44a8539b 50 years ago 260MB
```

```ntodo-list-server latest 584a44a8539b 50 years ago 260MB
```

By the way, if you plan to run the application in a remote Kubernetes cluster, you can now tag the images above accordingly and push them to a Docker repository that your Kubernetes cluster can access, using standard Docker tooling. If you only want to run the application in a local Kubernetes cluster, you should be all set.

You can test the Docker image by running it locally and accessing `http://localhost:7001/`, just like you did when running within an IDE or from the command line before, for example:

```
$ docker run -p 7001:7001 todo-list-server
```

You should see familiar log output in the terminal window and in the UI that you already know and love, and you should be able to create, edit, and complete some tasks.

**Deploying the application to Kubernetes**

By far the easiest (and recommended) way to deploy Coherence CE applications to Kubernetes is with **Coherence Operator**, which is a Go-based application designed for this purpose.

Coherence Operator defines a custom resource definition (CRD) for Coherence deployments, which makes correct Coherence cluster configuration, scaling, and management in general significantly simpler.
For example, Coherence Operator will automatically configure a headless service that allows Coherence cluster members to discover one another, services for external endpoints that you want to expose, a readiness probe that checks whether a Coherence member is fully initialized and ready to accept requests, and many other things. In other words, if you need to run Coherence in Kubernetes, use Coherence Operator.

The easiest way to install Coherence Operator is with the Helm package monitor for Kubernetes.

The first step is to add the Coherence Helm repo to your local Helm configuration by running the following commands:

```bash
$ helm repo add coherence
https://oracle.github.io/coherence-operator/charts
$ helm repo update
```

The next step, which will install Coherence Operator, depends on whether you are using Helm 2 or Helm 3. If you are using Helm 2, run the following command:

```bash
$ helm install --name coherence-operator coherence/coherence-operator
```

If you are using Helm 3, you can omit the `--name` flag, for example:

```bash
$ helm install coherence-operator coherence/coherence-operator
```

Once Coherence Operator is installed, you should see a message indicating that. You can also verify the installation by running the following command and making sure the output resembles what’s shown below:

```bash
$ kubectl get deployments
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
</tr>
</thead>
<tbody>
<tr>
<td>coherence-operator-controller-manager</td>
<td>1/1</td>
</tr>
</tbody>
</table>

Once Coherence Operator is up and running, you are ready to deploy the to-do list application. The first step is to create a YAML file for the Coherence deployment, as follows:

```yaml
apiVersion: coherence.oracle.com/v1
kind: Coherence
metadata:
  name: todo-list
spec:
  replicas: 1
  image: todo-list-server:latest
  application:
    type: helidon
  jvm:
```
Here’s an explanation for the resource above:

- The values for the `apiVersion` and `kind` attributes reference a custom CRD that was installed by Coherence Operator and trigger the processing of the Coherence deployment resource above by Coherence Operator.

- The `metadata/name` attribute is the only other required piece of information, and it is used as a Kubernetes identifier for this Coherence deployment. It is also used as the name of the Coherence cluster, unless it is overridden via the `spec/cluster` attribute.

The other attributes within the `spec` element should be fairly self-explanatory:

- The `replicas` attribute defines the number of Coherence cluster members to start.
- The `image` attribute tells Kubernetes to use the Docker image created earlier (and should be modified accordingly if you tagged the image and pushed it to a Docker repo).
- The `jvm/memory/heapSize` attribute tells Coherence Operator to set the initial and maximum heap size for each cluster member to 2 GB.
- The `application/type` attribute tells Coherence Operator that this is a Helidon application, which results in the default Helidon main class being used to start each cluster member, instead of the `DefaultCacheServer` class that would normally be used in a standalone Coherence deployment.
- Finally, the resource exposes ports 1408 and 7001 to serve the gRPC and HTTP/REST endpoints, respectively, via Kubernetes services `todo-list-grpc` and `todo-list-http`, based on the default service naming convention of appending the port name to the Coherence deployment name, and the resource tells Prometheus to scrape the metrics from each cluster member over the HTTP port by enabling the service monitor.

With Coherence Operator installed, and the content above available in the `app.yaml` file, you can now install the application by running this command:

```
$ kubectl apply -f java/server/src/main/k8s/app.yaml
```

`coherence.coherence.oracle.com/todo-list` created
Let's check what was actually created by Coherence Operator based on the Coherence deployment resource above.

First, you can see the details about the Coherence deployment itself by running this command:

```
$ kubectl get coherence
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>CLUSTER</th>
<th>ROLE</th>
<th>REPLICA</th>
</tr>
</thead>
<tbody>
<tr>
<td>todo-list</td>
<td>todo-list</td>
<td>todo-list</td>
<td>1</td>
</tr>
</tbody>
</table>

The output shows that the deployment name, the cluster name, and the role have all defaulted to `todo-list`, and you have one ready member in the cluster. Let's see what hides behind the Coherence deployment above by using this command:

```
$ kubectl get all
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>pod/todo-list-0</td>
<td>1/1</td>
<td>Running</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>CLUSTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>service/todo-list-grpc</td>
<td>ClusterIP</td>
<td>10.96.1</td>
</tr>
<tr>
<td>service/todo-list-http</td>
<td>ClusterIP</td>
<td>10.96.8</td>
</tr>
<tr>
<td>service/todo-list-sts</td>
<td>ClusterIP</td>
<td>None</td>
</tr>
<tr>
<td>service/todo-list-wka</td>
<td>ClusterIP</td>
<td>None</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>statefulset.apps/todo-list</td>
<td>1/1</td>
<td>8m52s</td>
</tr>
</tbody>
</table>

As you can see, Coherence deployment is backed by a stateful set, which allows you to configure reliable persistent storage for cluster members, as you'll do in a minute.

There are also four Kubernetes services that were created by Coherence Operator. The first two are what you would expect based on the deployment resource, and they enable access to gRPC and the HTTP endpoints provided by each cluster member. The other two are headless services that are automatically created by Coherence Operator to keep track of the stateful set members and a well-known addresses (WKA) list that allows new members to join the Coherence cluster. You can ignore the fact that they exist, at least for now.

### Scaling the Coherence cluster

At the moment, the stateful set has only one pod within it, but that will change when you scale the cluster with this command:

```
$ kubectl scale coherence todo-list --replicas 10
```

coherence.coherence.oracle.com/todo-list scaled
It may take a minute or two for the additional nine members to start and join the cluster, but once everything is up and running you should be able to run the following commands and see similar output:

```
$ kubectl get coherence
NAME    CLUSTER    ROLE     REPLICAS
todo-list todo-list todo-list 10

$ kubectl get all
NAME                          READY   STATUS
pod/todo-list-0               1/1     Running
pod/todo-list-1               1/1     Running
pod/todo-list-2               1/1     Running
pod/todo-list-3               1/1     Running
pod/todo-list-4               1/1     Running
pod/todo-list-5               1/1     Running
pod/todo-list-6               1/1     Running
pod/todo-list-7               1/1     Running
pod/todo-list-8               1/1     Running
pod/todo-list-9               1/1     Running
NAME                         READY   AGE
statefulset.apps/todo-list   10/10   46m
```

Yes: That's what I mean when I say that Coherence allows you to scale stateful workloads just as easily as stateless workloads. Each one of the 10 pods above not only serves the incoming REST and gRPC requests but also stores approximately one-tenth of the data the application manages!

The best part is that I scaled to only 10 nodes to keep the output somewhat manageable. As long as I have enough capacity in the Kubernetes cluster I’m using, I can easily scale to 100, 500, or even 1,000 members. Keep in mind that Kubernetes limits the number of pods per node to 100, and you shouldn’t really push that limit. So you will need a fairly large Kubernetes cluster to run large Coherence clusters.

I can also scale up by making each member bigger. For example, by changing the member heap size from 2 GB to 20 GB, I can increase cluster storage capacity tenfold without changing the number of members.

You can play with the size of the cluster and the size of each member to balance your data storage and processing needs. The more members you have, the more JVMs your HTTP and gRPC requests will be load balanced across; at the same time, the bigger each member is, the more data it will be able to store.

However, it is important to understand the trade-offs as well. Larger clusters (based on the number of members) consume more resources, create more monitoring data to collect, and tend to be more difficult to manage.
At the same time, in a cluster that is too small (say 2 to 4 members, each holding a lot of data), failure of a single member can have a significant impact on the time it takes to fail over and rebalance the data across the remaining members. It is much more costly to fail over and rebalance one-third of the data set than one-thirtieth of the data set.

Accessing the application

Now that you have the application up and running, how can you access it? The simplest way, which is good enough for a quick test, is by using Kubernetes port forwarding to forward local ports to the HTTP and gRPC services you created, for example:

```bash
cat
$ kubectl port-forward service/todo-list-http 7001:7001
$ kubectl port-forward service/todo-list-grpc 1408:1408
```

Each command above will block, so you need to run them in separate terminal windows or tabs. You can also run them in the background, but I do not recommend that because you won’t see the output. Also, keep in mind that Kubernetes has a habit of dropping forwarded connections quite frequently, so you may need to reconnect if that happens.

With the port forwards above in place, you should be able to access both the web front end and the gRPC endpoints from the JavaFX client via the same local endpoints you used before.

Obviously, port forwarding is not how you should run a production application. Instead, you would make the application available via an ingress controller, typically fronted by a load balancer.

How exactly you do that is outside the scope of this article. That said, assuming you have configured the Nginx ingress controller for your Kubernetes cluster, you should be able to create ingresses for both HTTP and gRPC endpoints using definitions similar to these:

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: http-ingress
spec:
  rules:
  - host: tasks.seovic.com
    http:
      paths:
      - backend:
        serviceName: todo-list-http
        servicePort: 7001

---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
```
To complete the configuration, you would need to use the domain name that you own and create DNS “A” records for the hosts above that point to the load balancer in front of your ingress controller.

### Configuring data persistence across restarts

You now have a cluster of 10 members up and running that is deployed across multiple VMs, machines, and possibly even availability zones or domains in the cloud. This allows the application to tolerate the failure of not only individual members but of a whole availability zone or domain. Why? Coherence will, by default, create backups as far from the primary copy of the data as possible. If the cluster is spread across availability zones A, B, and C, a backup for a primary copy owned by a member in zone A will be stored in zone B or C. This is completely automatic, as long as the Kubernetes zone metadata is configured correctly.

However, the application’s data is still stored only in memory, and it remains available only as long as the cluster is up and running. If you shut the whole cluster down, you will lose all the data, which is rarely, if ever, desirable—especially in a to-do list application such as the example in this article.

To preserve data across cluster restarts, you have two options:

- Write data automatically to some kind of persistent datastore, such as a relational database or key-value store, and load data from it when necessary. This is certainly possible via the Coherence cache store mechanism, but it is less than ideal for this application because it requires setting up an external datastore. That negates some of the benefits and simplicity of the stateful application you created.

- Enable Coherence persistence and configure the application to attach a persistent disk volume to each pod within the stateful set backing your deployment. This way Coherence itself will persist all the data to a disk volume that is managed by the cloud provider and will be reattached to the correct pod upon restart. This is the option that’s best for this project, and that’s what I’ll discuss next.

To enable Coherence persistence and tell the cloud provider to attach some disk volumes to the stateful pods, add the following annotations to your ingress:

```yaml
name: grpc-ingress
annotations:
  nginx.ingress.kubernetes.io/backend-protocol: grpc
spec:
  rules:
  - host: grpc.tasks.seovic.com
    http:
      paths:
      - backend:
          serviceName: todo-list-grpc
          servicePort: 1408
```

To enable Coherence persistence and tell the cloud provider to attach some disk volumes to the stateful pods, add the
This will configure Coherence to use the active persistence mode, instead of the default on-demand mode, which will cause it to write data to disk upon each modification. This also instructs Coherence Operator to provision a 50-GB persistent volume storage for each cluster member using the oci-bv (OCI Block Volume) storage class.

**Note:** I am running the application in Oracle Container Engine for Kubernetes, which is why I specified oci-bv as a storage class. If you are using a different cloud provider, you should modify the storage class name accordingly.

Persistence is one of the few features that impact the whole cluster and cannot be changed at runtime, so to apply the change above you will need to redeploy the application, as follows:

```shell
$ kubectl delete -f java/server/src/main/k8s/coherence.coherence.oracle.com "todo-list" delete
$ kubectl apply -f java/server/src/main/k8s/app.yaml
```

Once everything is back up and running, you can check that a PersistentVolumeClaim (PVC) was created for the (only) cluster member, for example:

```shell
$ kubectl get pvc
```

```shell
NAME                             STATUS   VOL
persistence-volume-todo-list-0   Bound    csi
```

Similarly, when you scale the cluster back to 10 members, each member should have a 50-GB PVC attached, as shown below:
You can verify that the PVC was mounted by the pod by running this command:

```bash
$ kubectl describe pod todo-list-0
```

Then, verify that you see the persistent volume mount within the “Mounts:” section

```
/coherence-operator/persistence from persistence-volume (rw)
```

and the corresponding volume definition within the “Volumes:” section

```
persistence-volume:
  Type:       PersistentVolumeClaim
  ClaimName:  persistence-volume-todo-list-0
  ReadOnly:   false
```

The easiest way to test if the persistence actually works as expected is to create some tasks, complete a few of them, and then scale the cluster down to zero members by using the following command:

```bash
$ k scale coherence todo-list --replicas 0
```

This will effectively shut all the members down and terminate all the pods within the Coherence deployment. Once you have verified that all the pods have been terminated, scale the cluster back to 10 members and refresh the UI (you may need to forward the port again, because the connection will likely be broken by the previous shutdown). All the data should be there, exactly the way you left it before shutting the cluster down.

By making a simple YAML change and enabling persistence, you now have a durable, stateful application that can be scaled to hundreds of JVMs, members, and pods just as easily as any stateless application.
Configuring observability and tracing

It’s great that you have the application up and running and that you can easily scale it out or up to support additional load, but this is still a complex, distributed system. You need to be able to observe what’s happening inside of it to address issues when they inevitably arise.

Coherence enables observability via several mechanisms:

- Monitoring metrics, exposed via Java Management Extensions and an OpenMetrics-compliant HTTP endpoint that can be consumed by tools such as Prometheus
- Built-in Grafana dashboards that can be used to visualize those metrics
- Tracing information via OpenTracing that allows you to better understand the flow of individual requests through the system, which can help you understand and fix performance bottlenecks

I will cover the first two topics in a bit, but for now I’ll focus on the third one: tracing.

My colleague on the Oracle Coherence core development team, Ryan Lubke, who implemented most of the OpenTracing support in Coherence, has written a nice series of articles that describe how OpenTracing works and how it integrates with other technologies. You should read those articles, but I’ll try to summarize the important bits in the context of a Helidon application, which makes a number of things a bit simpler.

Both Helidon and Coherence support OpenTracing out of the box. All you have to do to enable it is to make a few minor changes to your application.

First, specify the name of the service that you want to publish tracing information for. This is easily accomplished by adding the following line to the META-INF/microprofile-config.properties file within the server project:

```
service.name=Tasks
```

You also need to add a dependency on the OpenTracing library you want to use. Helidon supports both Zipkin and Jaeger, and for this application, I’ve chosen the latter. To enable it, add a single dependency to the POM file, as follows:

```
<dependency>
  <groupId>io.helidon.tracing</groupId>
  <artifactId>helidon-tracing-jaeger</artifactId>
</dependency>
```

That’s really all there is to it as far as the application is concerned. The two minor changes above will cause the Helidon
application to trace each REST request, all the way through Coherence, which will simply add its own spans to an existing trace started by Helidon.

With the changes above in place, you can rebuild the application and repackage it into a Docker image, but before you can deploy it, the following are a few additional changes you need to make for the Kubernetes deployment resource:

```yaml
spec:
  # insert before 'application' section
  env:
    - name: TRACING_HOST
      value: "jaeger-collector"
    - name: JAEGER_SAMPLER_TYPE
      value: "const"
    - name: JAEGER_SAMPLER_PARAM
      value: "1"
```

The first environment variable, `TRACING_HOST`, is actually the only one that is required, and it tells the Jaeger client where to publish tracing information.

The other two variables are there to change the default Jaeger client behavior and publish tracing information for every single request, instead of only for a small sample of the requests. It’s not something you would do in production, but it is certainly handy when you want to demonstrate tracing support in an application.

The last thing you need is to make sure you have a Jaeger instance up and running that you can publish tracing information to. Setting up Jaeger is outside the scope of this article, but you can follow these instructions to install Jaeger Operator into your Kubernetes cluster. Once that’s done, create a Jaeger instance for the application, which can be as simple as running this command:

```bash
$ kubectl apply -f java/server/src/main/k8s/jaeger.yaml
```

If you are using an ingress controller, you will also need to configure the ingress for the `jaeger-query` service, similar to the following:

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: jaeger-ingress
spec:
  rules:
    - host: jaeger.seovic.com
      http:
        paths:
          - backend:
              serviceName: jaeger-query
              servicePort: 16686
```
With Jaeger up and running and the latest version of the application deployed, you can create, complete, and delete some tasks and use the Jaeger UI to see what’s happening under the hood.

For example, the trace (see Figure 2) for one of the GET /api/tasks requests clearly shows that the ToDoResource.getTasks Java API for RESTful Web Services (JAX-RS) method was invoked, which then executed a query against all 10 Coherence members in parallel to retrieve a list of tasks.

![Figure 2. Querying for all tasks](image)

In Figure 3, you can see that when a new task is created via the POST /api/tasks request, the ToDoResource.createTask method is called, which performs a Put request against a primary owner for the newly created Task, which then writes it to a backup member and the persistent store (disk) in parallel.

![Figure 3. Creating a task](image)

You can see a similar trace (Figure 4) when updating the task via the PUT /api/tasks/:id request. The
**ToDoResource.updateTask** method is called, which then uses an entry processor to update the task on the primary member or owner. The primary member then updates the backup and the persistent store in parallel, just as in the case of the **Put** request.

![Image of a task update](image)

**Figure 4. Updating a task**

I’m sure you’ll agree that having this level of visibility, from the REST API endpoint all the way to the disk write deep in the guts of Coherence, across potentially many cluster members executing code sequentially or in parallel, can greatly simplify troubleshooting performance issues. It can also help you visualize what’s going on in a complex, distributed system and allow you to verify that what you expect to be happening is actually happening.

I’m personally not familiar with any other product that provides this level of visibility into the inner workings.

**Monitoring with Prometheus and Grafana**

Tracing is great for understanding the flow of the requests through the system and pinpointing where performance bottlenecks are, but it certainly doesn’t provide the full picture. For example, there is no way to see how much data is in various caches or maps, what the overall health of individual cluster members and services is, the rate of requests or data growth, and many other things you might need to determine to truly understand the health of the system as a whole and to be able to make scaling decisions and other decisions to make the system perform well over time.

Doing all that requires you to capture a number of point-in-time metrics periodically and analyze them over time, which is exactly what time-series databases such as Prometheus and monitoring dashboards such as Grafana are designed for. Setting up Prometheus and Grafana is far beyond the scope of this article, but there are plenty of resources on the web that will help you configure both using Prometheus Operator. Make sure you enable the Prometheus **ServiceMonitor**, since that’s what
Coherence uses to configure Prometheus to scrape the `/metrics` endpoint on each Coherence cluster member.

**Important note:** If you don’t have Prometheus up and running before deploying the application, you may need to delete and redeploy the application for Prometheus to start scraping metrics from Coherence members.

You should also follow the instructions in the Coherence Operator documentation for how to import Coherence monitoring dashboards into Grafana.

Once you have everything up and running, you should see something similar to **Figure 5** when you log in to Grafana and open the main Coherence dashboard:

![Grafana's cluster overview dashboard](image)

**Figure 5.** Grafana’s cluster overview dashboard

This dashboard allows you to see, at a glance, the total number of cluster members, heap utilization for the cluster as a whole and per member role, and member load averages, as well as how many members have recently left the cluster and whether there are any endangered services (that is, services without backups).

If you click the **Cluster Members** box, the Member Summary dashboard will open, and you can then look at individual member dashboards by clicking a member’s name. For example, the Member Details dashboard for the todo-list-0 member looks like **Figure 6**:
On the Member Details dashboard, you can see heap utilization, thread utilization and garbage collection information for that member, as well as publisher and receiver success rates, which indicate the quality of the network between this and other members; the closer the number is to 100%, the better.

On the Services Summary dashboard, shown in Figure 7, you can see the status of each Coherence service.

In Figure 7, the application service I care most about, todo-listService, runs on all 10 members and is in the MACHINE-SAFE state, meaning it could tolerate the loss of any of the six virtual machines in the Kubernetes cluster without losing any data. You can also see when the load was the highest by looking at the distribution of task executions over time.

Clicking the todo-listService link brings up the Service Details dashboard for that service, as shown in Figure 8.
You can see average task and request times for the service per member, as well as thread utilization information and the task rate and backlog per member.

The Cache Details dashboard (Figure 9) shows the total number of tasks you are managing, the amount of memory those tasks consume, and the size of indexes (if there are any). You can also see query duration information, how much memory was consumed by this cache over time, and how the entries are distributed and accessed across cluster members.

Finally, on the Persistence Summary dashboard (Figure 10), you can see how much disk space is consumed by the data the application is managing and how much disk space is still available:
These are not all the dashboards provided by Coherence, but they are the most useful ones for this particular application and will likely be useful in almost any application, so they are worth becoming familiar with.

As you can see, these tools provide lots of useful information out of the box, and they can be customized as necessary to provide exactly the information you care about.

**Conclusion**

This article completes the series and, truth be told, it's a bit longer than I expected it to be and covers a lot of ground. The same is true for the whole series, which was originally supposed to be a single article. [Whoops! —Ed.]

I hope you enjoyed seeing how Oracle Coherence Community Edition can help you build stateful services and applications that are resilient and easy to scale. Feel free to use the code for the sample application as a guide when implementing your own applications, and if you have any questions please reach out to me directly, or use one of the official Coherence social media channels, all of which you can find on the Coherence CE website.

I will leave you with **Figure 11** as an idea of where you can go with Coherence CE:
Aleks Seović

Aleks Seović (@aseovic) is an architect at Oracle, where he works on Oracle Coherence, a leading in-memory data grid product, and contributes to the Helidon microservices framework. Most recently, Seović led the design and implementation of the Helidon gRPC framework, as well as CDI and Eclipse MicroProfile support in Coherence. He currently leads the implementation of Coherence native clients, GraphQL support, and Spring integration. Prior to joining Oracle in 2016, Seović led a boutique consultancy practice, where he worked with customers around the world to help them implement mission-critical applications on top of Coherence. Seović is the author of Oracle Coherence 3.5 (Packt Publishing, 2010) and frequently speaks about and evangelizes Coherence at industry conferences, Java and .NET user group events, and Coherence SIGs.