payara®

Unlocking the Speed: Performance Tuning for Jakarta EE Applications With JCache

The Payara® Platform - Production-Ready, Cloud Native and Aggressively Compatible.

User Guide

Contents

Guide Updated: **August 2023**

Application performance has always been an important aspect of software engineering. However, in these times of constant cloud connectivity and availability, where users expect applications to respond instantaneously, application performance is even much more crucial to the full user experience. As organisations increasingly migrate their applications to cloud environments and embrace the principles of cloud native architecture, the need for finely tuned performance becomes even more pronounced.

In this brief guide, we look at how to improve cloud native, Jakarta EE application performance using the JCache, the standard cache specification on the Java Platform. We start by looking at the fundamentals of performance tuning, then take a look at caching in general, and then move on to JCache, what it is and how to use it in a Jakarta EE context. By the end of this guide, you will have a foundation in getting started with caching with JCache on the Jakarta EE Platform.

Fundamentals Of Performance Tuning

Performance tuning is an important part of software development that aims to optimise an application's behaviour and execution to achieve maximum speed, responsiveness, and resource efficiency. Overall, performance tuning aims at improving the user experience of an application. It involves identifying and resolving performance bottlenecks that may hinder an application's ability to deliver a seamless user experience. In the context of cloud native applications, performance tuning takes on added significance due to the distributed, dynamic, and scalable nature of these applications.

Goals of Performance Tuning

The primary goals of performance tuning are to enhance an application's efficiency, reduce response times, minimise resource utilisation, and ensure consistent and reliable performance under varying conditions. By fine-tuning an application's performance, you can mitigate potential issues such as slow load times, unresponsive interfaces, and degraded user experiences. Effective performance tuning translates into improved customer satisfaction, increased user retention, and optimised resource utilisation, contributing to the overall success of your application.

Key Performance Metrics and Indicators

Performance tuning involves measuring and analysing various metrics and indicators that provide insights into an application's performance. These metrics help you gain a comprehensive understanding of how your application is performing and identify areas that require optimization. Some key performance metrics and indicators include:

Response Time

The time it takes for your application to respond to a user request. Shorter response times generally indicate better performance and increased user experience.

Latency

The delay between initiating a user request and receiving a response. Lower latency is crucial for real-time and interactive applications.

Memory Usage

The amount of memory consumed by your application. Optimising your application's memory consumption leads to better performance and less costs.

Throughput

The number of operations your application can perform in a given time period. A higher throughput means better utilisation of compute resources per unit of time.

CPU Utilisation

The percentage of the CPU's processing capacity being used. High CPU utilisation may indicate resource contention or inefficient code or sometimes different combinations of both problems.

Network Latency

The delay in transmitting data over a network. Minimising network latency is important for distributed cloud applications.

Common Performance Bottlenecks in Cloud Native Applications

Cloud native applications, while offering scalability and flexibility, can encounter specific performance bottlenecks. Some common bottlenecks in cloud native applications include:

Network Latency

Communication between microservices and services hosted in different containers (or even domains) can introduce latency, impacting overall application performance.

Resource Contentions

In a shared cloud environment, competition for resources such as CPU, memory, and network bandwidth can lead to performance degradation.

Database Access

Inadequate database optimization can result in slow query response times, affecting application responsiveness. Also, unnecessary round trips to the database for data that can be cached could result in large database bills. This can affect both SQL and NoSQL databases.

Data Transfer

Excessive data transfer between microservices or from external sources can strain network resources and cause delays.

Inefficient Scaling

Poorly orchestrated scaling of microservices can lead to over-provisioning or under-utilisation of resources.

Having a better understanding of these points in mind during development can help you develop much more performant and resilient applications.

Caching With JCache

Caching, according to Amazon Web Services, "is a high-speed data storage layer which stores a subset of data, typically transient in nature, so that future requests for that data are served up faster than is possible by accessing the data's primary storage location." Caching then means bringing data closer to the point of query.

Caching typically entails the use of very fast temporal storage mechanisms to keep data in-memory for a predefined period of time, after which it is either manually or automatically purged and replaced with new data. Caching has many benefits of for applications in general and cloud native applications in particular, such as:

Improved User Experience

Users have come to expect near instantaneous applications responsiveness. One of the ways this can be achieved is through faster reading of application data. The cached data could be anything, from database record instances to files to any form of data the application handles and processes.

Lower Database Costs

Cloud native applications store data in some form of permanent data warehouse, be it SQL or NoSQL databases. By caching data, the need to make calls to the database can be significantly reduced. This can translate to significant database cost savings, especially for cloud hosted database services.

Consistent Performance

The use of caching allows your application to deliver predictable performance within reasonable bounds. This can help you plan resource allocation for your project without falling into over or under provisioning.

JCache

JCache, or Java Caching API, is the standard cache API on the Java Platform. It is developed under Java Specification Request (JSR) 107. JCache, similar to other Java APIs, comprises a set of core interfaces organised under the javax.cache package. These are

- **Cache** - This defines access to the actual cache, which is a map-like data structure that stores key-value pairs
- **Entry** - This defines access to the key-value pairs stored in the cache. You can think of this as a unit of the cached data
- **CacheManager** - This defines how caches are managed.
- **CachingProvider**- This defines how CachingManagers are managed.
- **ExpiryPolicy** - This defines how expiration is handled for entries.

These interfaces give you fine grained control over how to consume the API in your applications. As with other Java APIs, JCache needs an underlying implementation of the specification. Jakarta EE runtimes bundle implementations on your behalf. But you can always change to one that you prefer. The Payara runtime ships with the popular Hazelcast by default.

JCache is light in terms of strict requirements for implementations. It leaves out a lot of things for the implementations to decide. For instance, JCache does define how data should be stored in caches. It leaves out such technical details for the implementations to decide. This leaves room for innovation on the part of the implementations on top of the base spec.

Getting Started With JCache In Jakarta EE

As JCache is a standalone API, you will need to explicitly add it to your project dependency as shown below.

```
<dependency>
        <groupId>javax.cache</groupId>
        <artifactId>cache-api</artifactId>
        <version>1.1.1</version>
    </dependency>
```

This is all you will need when using Payara as your Jakarta EE runtime, as it already ships with Hazelcast.

Using JCache

There are three broad ways of creating caches using the JCache API. Which one you use will depend on the needs of your application. Two of them are easy to use but offer less control. The third option offers more control but is verbose.

Declarative Caching

You can effortlessly create and cache data through JCache annotations. The root entry annotation is through the use of the @CacheDefaults annotation as shown below.

```
@CacheDefaults(cacheName = "geoCache")
@ApplicationScoped
public class GeoController {

    @CacheResult
    public String searchZip(@CacheKey final String cityName) {
        //Make external call to some service
        //Cache result to not need to make the same call for the same city
        return "";
    }
    @CacheRemove
    public void removeFromCache(@CacheKey final String cityName) {
    }
}
```

Class `GeoController` is an application scoped bean that is annotated `@CacheDefaults`. This annotation creates a default cache named `'geoCache.'` The searchZip method is annotated `@CacheResult`, and the method parameter annotated `@CacheKey`. The first annotation will cause JCache to put the returned value of the method into the cache. The key to each entry of the returned value in the cache is the method parameter. A subsequent call to the method with a key that is in the cache will cause the cached value to be returned. With these few annotations, you automatically have caching enabled. For most use cases, this should be enough.

The `removeFromCache` method in `geoController` is annotated `@CacheRemove` and its parameter also annotated `@CacheKey`. Calling this method will cause any cached key/value pair that matches the passed key to be removed. You should normally have this kind of method to manually remove stale entries. For instance, if a given entity is updated somewhere in the application layer, it is a good idea to call this method to remove any cached value to ensure the updated data is returned to the client next time around.

Caching By Injection

The second way you can get a Cache is through injection.

```
@ApplicationScoped
public class TripsAdvisorService {
@Inject
Cache<Integer, PointsOfInterestResponse> cache;
}
```

You can inject a typed cache using `@jakarta.inject.Inject` as shown above. A cache will be instantiated and injected into the field. You can then use the cache to store and retrieve entries as shown below.

```
public PointsOfInterestResponse suggestPointsOfInterest(String city, BigDecimal
budget) {
    int cacheKey = generateKey(city, budget);
    if (cache.containsKey(cacheKey)) {
    return cache.get(cacheKey);
    }
    String request = String.format(Locale.ENGLISH, "I want to visit %s and have
a budget of %,.2f dollars",
    city, budget);
    var poi = sendMessage(request);
    List<PointOfInterest> poiList = generaPointsOfInterest(poi);
    PointsOfInterestResponse response = new PointsOfInterestResponse();
    response.setPointsOfInterest(poiList);
    cache.put(cacheKey, response);
    return response;
    }
```

Method `suggestPointsOfInterest()` first generates a cache key based on the passed method parameters. It then checks if there is an entry in the cache based on the passed key. If not, the method creates a new `PointsOfInterestResponse` object, puts it in the cache and then returns the response. I personally find the use of cache injection much cleaner and readable. Injection of cache is fully supported by the Payara runtime, though other runtimes may equally provide support for it.

Programmatic Caching

Programmatic caching is the most flexible of the three options to get hold of a cache. The flexibility comes at the cost of a little verbosity to glue together a cache. The following class shows an example of programmatic cache creation.

```
@ApplicationScoped
public class CacheController {
@Inject
CacheManager cacheManager;

private Cache<Integer, PointsOfInterestResponse> geoCache;

@PostConstruct
void init() {
CacheManager cacheManager = Caching.getCachingProvider().getCacheManager();

MutableConfiguration<Integer, PointsOfInterestResponse> mutableConfig = new
MutableConfiguration<>();

mutableConfig.setReadThrough(true);
mutableConfig.setCacheLoaderFactory((Factory<CacheLoader<Integer,
PointsOfInterestResponse>>) MyCacheLoader::new);
mutableConfig.setExpiryPolicyFactory(CreatedExpiryPolicy.factoryOf(Duration.
FIVE_MINUTES));
geoCache = cacheManager.createCache("myCache", mutableConfig);
}

public PointsOfInterestResponse getResponse(final Integer key) {
return geoCache.get(key);
}

@PreDestroy
void cleanUp() {
geoCache.close();
}
}
```

`CacheController` is an application scoped CDI bean that initializes a Cache in the `@PostContruct` method. The Payara runtime supports direct injection of the `CacheManager`, which we do to get an instance. The class then creates a `javax.cache.configuration.MutableConfiguration` object for passing configurations to the Cache. The first configuration passed is setting the `readThrough` value to true followed by setting a `CacheLoader` through the `CacheLoaderFactory` field.

The read through configuration tells the cache to read a given entry from a given data source when there is a call to the Cache for an entry that is not in the cache. So if a key is passed for a given value, and the value is not present in the cache, the cache runtime will load the given value from a pre-defined source. The way to define this predefined source for loading of cache values is through an implementation of the `javax.cache.integration.CacheLoader` interface. The `CacheLoader` implementation `MyCacheLoader` is shown below.

```
public class MyCacheLoader implements CacheLoader<Integer,
PointsOfInterestResponse> {
    @Override
    public PointsOfInterestResponse load(Integer key) throws
CacheLoaderException {
        //Load data from some durable data store
        return new PointsOfInterestResponse();
    }

    @Override
    public Map<Integer, PointsOfInterestResponse> loadAll(Iterable<? extends
Integer> keys) throws CacheLoaderException {
        //Load data from some durable data store

        return Map.of(1, new PointsOfInterestResponse());
    }

}
```

This class will be consulted for getting a given entry from whatever durable data store you implement when a request is made to the cache and there is no corresponding value for it. Back to class `CacheController`, the class also sets the `javax.cache.expiry.ExpiryPolicy` to five minutes through the `CreatedExpiryPolicy.factoryOf()` convenience method. Finally, the class calls the createCache method on the CacheManager instance, passing in the `MutableConfiguration`.

The class has a `getResponse` method that takes an Integer and calls the Cache#get method with it. If there is no corresponding value in the cache for the passed key, the cache implementation will call the load method on the passed `CacheLoader` instance to retrieve a possible value (or null if

none) for the given key. Class `CacheController` finally calls Cache#close() in the `@PreDestroy` method. Calling close on the Cache signals to the `CacheManager` to release resources being coordinated on behalf of the Cache among other reasons.

As you have seen, the programmatic way of creating a Cache offers us much flexibility at the price of a little verbosity. Which option you use, as usual, will depend upon your use case. But as a rule of thumb, you should start with the simplest, and only resort to programmatic if your use case really demands it. Your application domain will determine what to use. But start easy.

There's More

JCache has support for advanced use cases such as processing cache entries in an atomic manner using `javax.cache.processor.EntryProcessor` and writing data to a configured durable data store whenever a cache entry is made through `javax.cache.integration.CacheWriter` implementation. `CacheWriter` is analogous to `CacheLoader`. One writes to the data store when data is put into the cache, while the other reads from the data store if the data is not found in the cache.

Using JCache in your Jakarta EE applications can significantly improve their responsiveness and user experience. You can find out more by looking at the JCache specification document here. As an example, you can look at this Jakarta EE ChatGPT application deployed to Payara Cloud. Enter a city and a budget, click go. The first time, a call is made to the OpenAI GPT API. The second time

around if you enter the same city/budget combination, the data is retrieved from a CDI injected JCache Cache instance. You will notice the second call is significantly faster than the first.

Summary

In this guide, we took a quick tour of application performance tuning with JCache. We looked at performance tuning, caching and finally how to use JCache in your Jakarta EE application. There is a lot more you can do with JCache, so check out the specification document. As a standalone

Interested in Payara? Try Before You Buy

PAYARA SERVER FREE TRIAL

PAYARA CLOUD FREE TRIAL

sales@payara.fish

UK: +44 800 538 5490
Intl: +1 888 239 8941

www.payara.fish

Payara Services Ltd 2023 All Rights Reserved. Registered in England and Wales; Registration Number 09998946
Registered Office: Malvern Hills Science Park, Geraldine Road, Malvern, United Kingdom, WR14 3SZ