medium.com

Introducing In-Memory Data Grid — Hazelcast IMDG - The Startup - Medium

Dina Bogdan

15-19 minutes

Win by using data!

Today’s article will be oriented to a very specific concept, which is the In-Memory Data Grid or IMDG, discussing all the ideas introduced by this one.

IMDG is a general or abstract concept, which describes a way to leverage some kind of distributed system for storing data and performing in-memory computations on stored data.

Being an abstract construction, we will dive into a concrete implementation of the concept, looking specifically at Hazelcast In-Memory Data Grid.

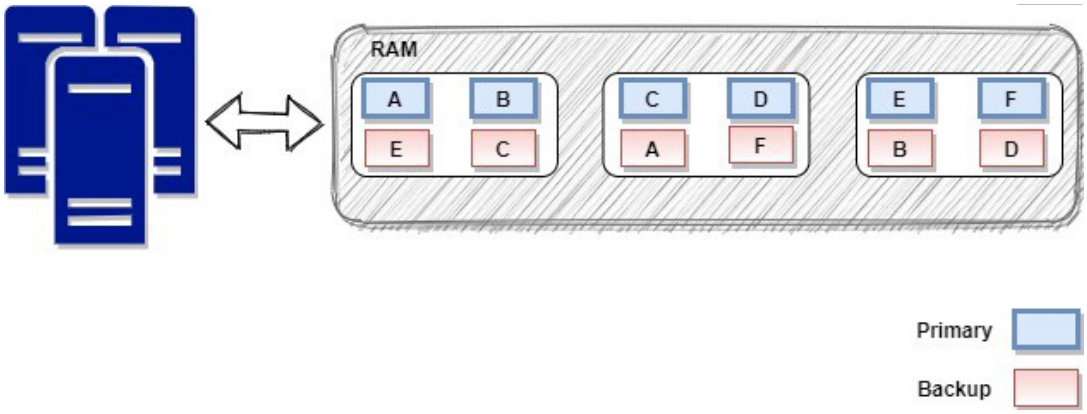
What is an In-Memory Data Grid (IMDG)?

First of all, let's introduce the concept of Data Grid. A Data Grid is a system of multiple servers that work together to manage information and related operations in a distributed environment.

The servers from the grid can be located in the same location or distributed across multiple data centers.

An In-Memory Data Grid is a grid that stores data entirely into Random Access Memory (RAM).

The visual representation of how a Data Grid looks like can be seen below.

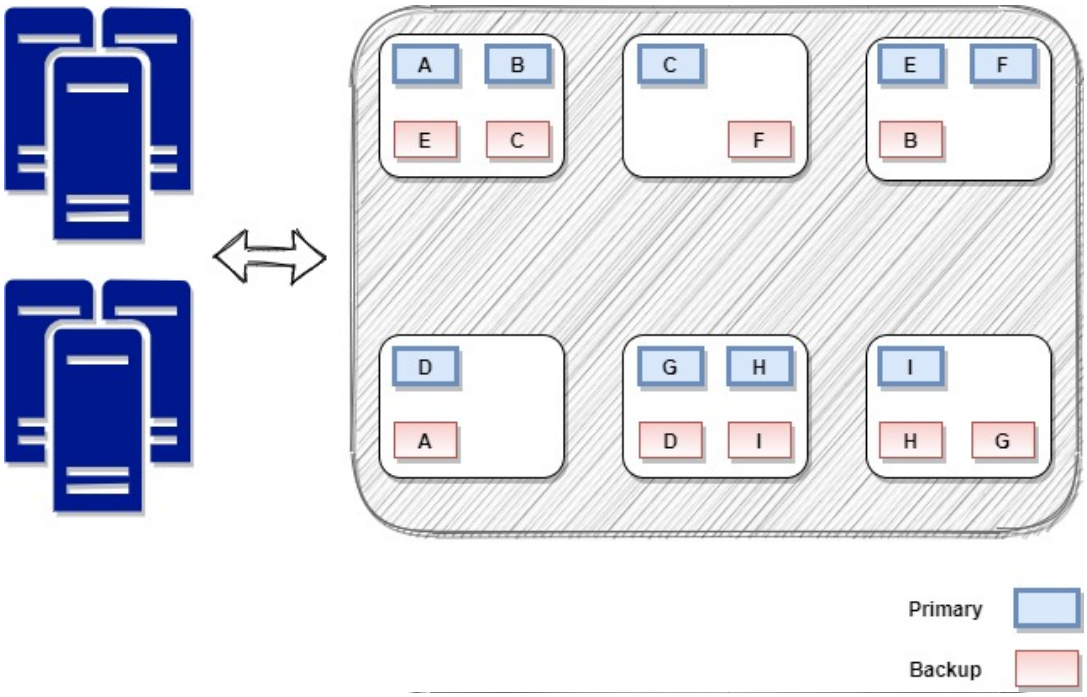


We can see that data is distributed across the entire cluster in such a way that the load and the capacity are balanced across all nodes in the grid. Also, the data is stored in multiple copies on different nodes. This will offer automatic resilience in case of single or multiple server failures.

If we need to scale the grid, which means that we need more capacity, we can just add more servers to the grid. The IMDG will redistribute data to even out the load across all available nodes. The same process works in reverse if one or multiple servers becomes unavailable. In such a case, some backups stored on

running nodes will be promoted as primaries.

The above-described scenario can be viewed in the below diagram. We can see how adding three more servers to our data grid, the load was redistributed to all the members in the cluster.



Why to use an In-Memory Data Grid?

There is more than one reason for using an IMDG. In the following, I will enumerate only a few of them.

- **Performance increase:** we have seen what is the real cost of latency. One of the best advantages of an In-Memory Data Grid is the increase in the entire system performance.

  Moving all the data from external data sources into memory will increase the performance of the entire system. In this way, we will avoid all the database read or write operations and replace them with reading and writing data from memory, which can be up to 1000 times faster.

- **Data structure:** the data structure itself is a simple key-value store, rather than a complex relational model. This simple structure is easy for developers to work with, while still providing all the ACID transactional properties.

- **Operations:** from an operational perspective, we already saw how an IMDG can be scaled up and down and how redundancy is built-in, which will allow us to easily achieve a highly-available system.

When to use an In-Memory Data Grid?

We have seen so far some of the advantages provided by an

IMDG, but let's look at some specific use-cases.

There are two major use-cases for an In-Memory Data Grid: Data Cache and Data Service Fabric. Let's see some characteristics of each of these two use-cases and also some real-world examples of IMDG usage.

1. **Data cache**

   In modern distributed systems, one of the common issues that we are facing when the number of users grows large enough is the contention on some data store or some component of the system which is providing data to the rest of the system's components.

   A solution to this issue is to move the data into some distributed cache and retrieve the data from there. Because the data is all resident in memory, there is no read/write delay to a datastore. So we are eliminating both the data store bottleneck along with the slow network connections.

   By using an IMDG in this way we don't have to worry about the performance of the network between middleware and the database. Also, we can work-around the long-running calculations that keep the application blocked for continuing and move them into memory, basically, this is a step forward into designing a reactive system according to the reactive manifesto.

2. Data Service Fabric

By using the IMDG as a Data Service Fabric we can perform near real-time integration between multiple components of a system.

In this way, we achieve a lot of computing power which can help us in designing a responsive distributed system, which is the core of the entire reactive systems idea.

Leveraging the IMDG can allow us to use it as a messaging platform, basically performing in-memory streaming.

Real-world examples

Some real-world examples from the industry are:

- Analytics, which includes Risk Analysis or Fraud Detection
- Trading Systems, like Foreign Exchange Trading or Stock Exchange
- eCommerce
- Online Gaming

Basic operations of an In-Memory Data Grid

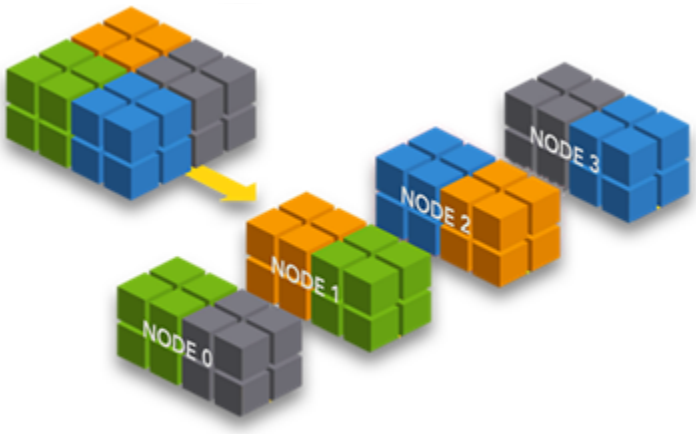
When looking at an IMDG implementation we should analyze some basic operations of it.

- **Cluster**

  The first of these operations is the capacity to run as a cluster. A cluster is a set of members that work together as a single unit to execute applications or perform specific tasks.

  The cluster ensures the even distribution of the data across all the nodes. Also, it provides high scalability and fault tolerance.

  The scalability performance of an IMDG is up to thousands of members or terabytes of data.



- **Discovery**

  Another important aspect that the IMDG must take care of is the

discovery mechanism which allows nodes to form a cluster or to find and join an existing cluster.

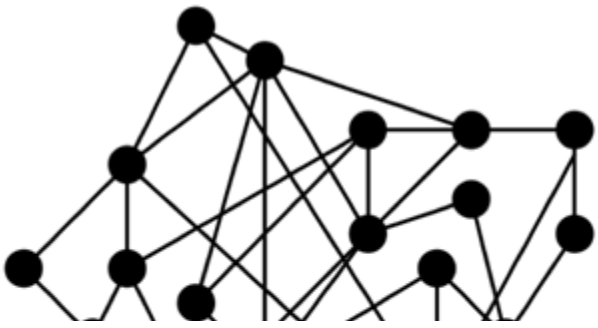
When an IMDG node is started, it looks for finding an existing cluster to join. If no cluster exists, then the node remains running as a stand-alone node. The second node will find the first one and together will form a cluster. Subsequently, all the other nodes will join the existing cluster.

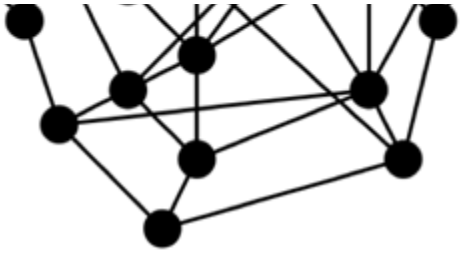
- **Data distribution**

  An IMDG must provide data distribution. The data is distributed across the entire cluster via replication or partitioning which is also known as sharding.

  Replication assumes that all nodes in the cluster will store a specific data structure which is very useful for achieving a high fault-tolerance degree.

  Partitioning means that a data structure is divided across multiple nodes in the cluster into disjoint pieces called partitions. For achieving a high degree of fault-tolerance we can provide partition backups.





Replication and Partitioning

We can have an entire and separate discussion about replication and partitioning, but in the following, we will try to show the consequences of using each of them in our distributed system.

We will avoid putting them in comparison or trying to consider what are the pros and cons of each because there is no one solution better than the other. As Michael Nygard shows in one of his articles, it's about the consequences of choosing one or the other, not about the advantages or disadvantages of it.

When we dealing with replication we will benefit from:

- the highest degree of availability because each node in the cluster can be used for serving data;
- workload scalability because the nodes in the cluster can be used for doing parallel computations;
- instantaneous failure recovery due to the fact that all nodes are storing an instance of our replicated data structure.

Still, there is a cost to be paid when we choose replication instead of partitioning for some data:

- there is a negative performance impact because when a data is changed, then that change must be propagated to all cluster members;
- synchronization of data or keeping the data consistent is challenging and when the data becomes out of sync then we can be not reliable anymore;

- replicated clusters have scalability issues because a new cluster member adds just another instance of a replicated structure, so if we want more memory available then we must scale in a vertical way the memory of the nodes.

  If we will look at replication from the CAP Theorem point of view, we will face an Available and Partition-Tolerant (AP) system.

  On the other hand, when we are using partitioning we will take advantage of the following:

- workload scalability because the data is partitioned across all the nodes and we can use cluster members in parallel for doing computations;
- failure recovery achieved by the usage of partition backups;
- increased memory scalability provided by adding more nodes to our cluster and producing a rebalancing of the partitions;
- compared with replication, there is no need for data synchronization, then the negative performance impact is out of discussion.

  As in the case of replication, partitioning implies doing some trade-offs:

- when we make changes to our cluster like adding or removing nodes, the data must be migrated so that it will be evenly distributed across all cluster members. This can have a negative performance impact in our system;
- there are large memory requirements when doing partitioning because of the backups that are made to partitions.

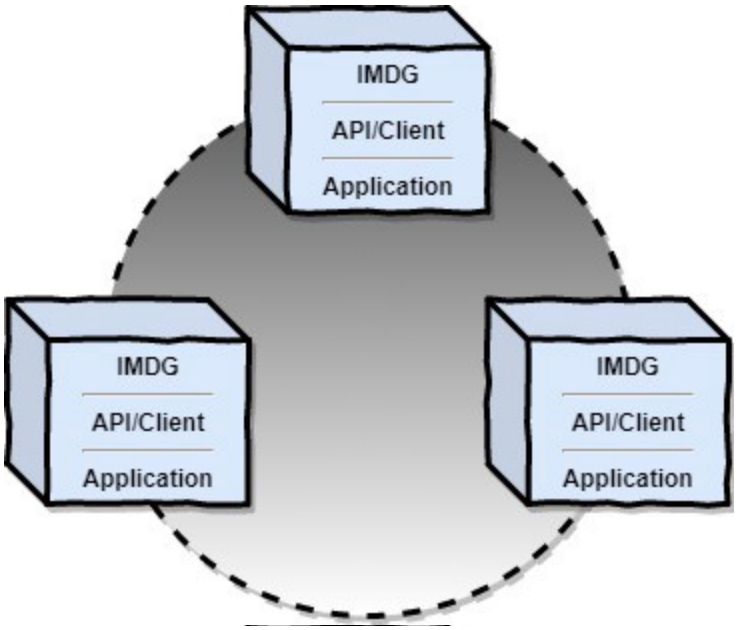
  Looking at a system that uses partitioning, from the CAP Theorem perspective, we will see a Consistent and Partition-Tolerant (CP) kind of system.

  Deployment options

There are two deployment models when using an IMDG: embedded IMDG and classical Client-Server architecture.

- **Embedded IMDG**

  This deployment model means that the server on the cluster member is running in the same process with the business application. Basically in this way, the data and the business logic are co-located in the same process. This is called data locality and is very efficient. This deployment model is used when dealing with the highest performance requirements.

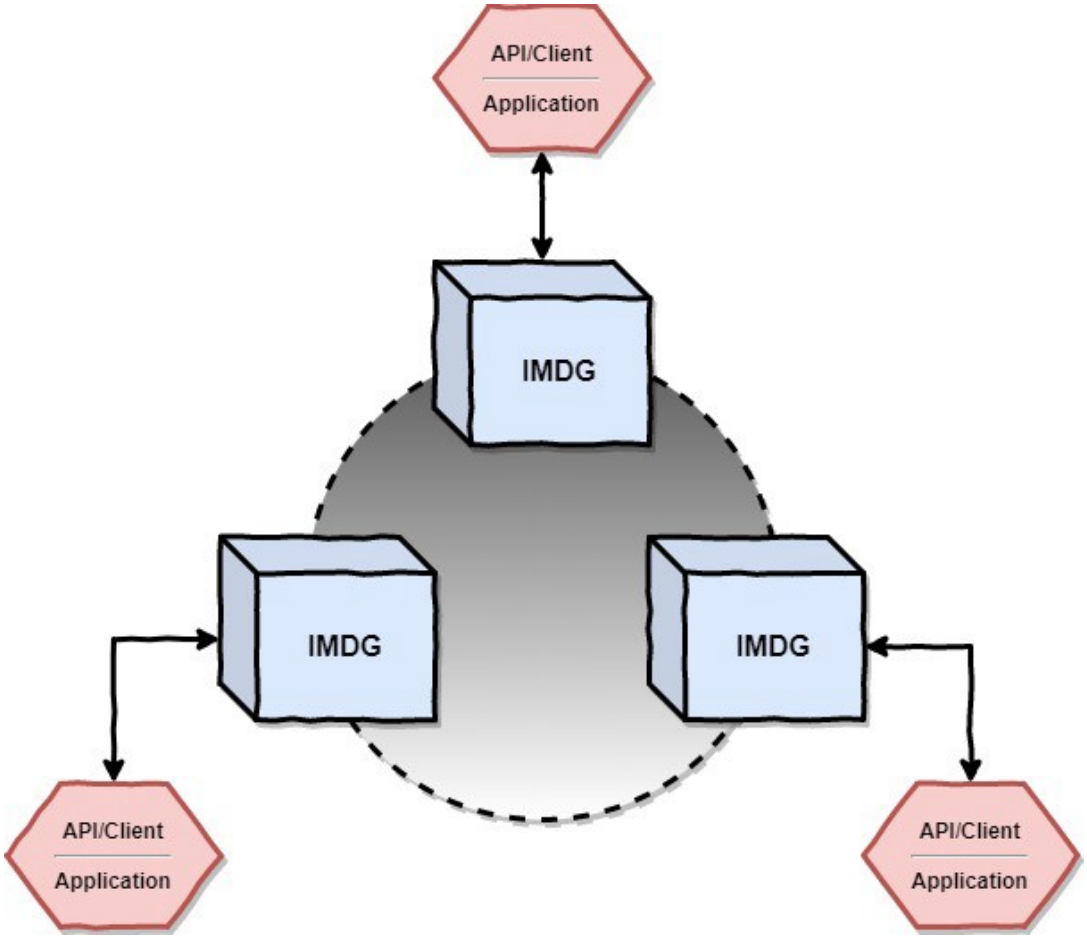


- **Client-Server**

  The Client-Server architecture assumes that IMDG members and business applications are running in two separate processes. The integration between the IMDG cluster and business apps is made via an IMDG client or APIs.

  The performance is not as good as in the case of the embedded IMDG model, but we will benefit from the decoupling between the cluster and the applications which will allow us to scale and evolve each of them independently. Also, this decoupling will allow us to develop business applications in whatever programming language we want.

Another very important aspect of this kind of deployment is that the business apps can be rebooted without losing the data that is stored in some IMDG members, as in the case of the Embedded IMDG model.



Hazelcast IMDG — Characteristics

During the rest of the article, we will look at some specific characteristics of one of the most popular implementations of the IMDG concept: Hazelcast IMDG.

Why we should choose Hazelcast IMDG?

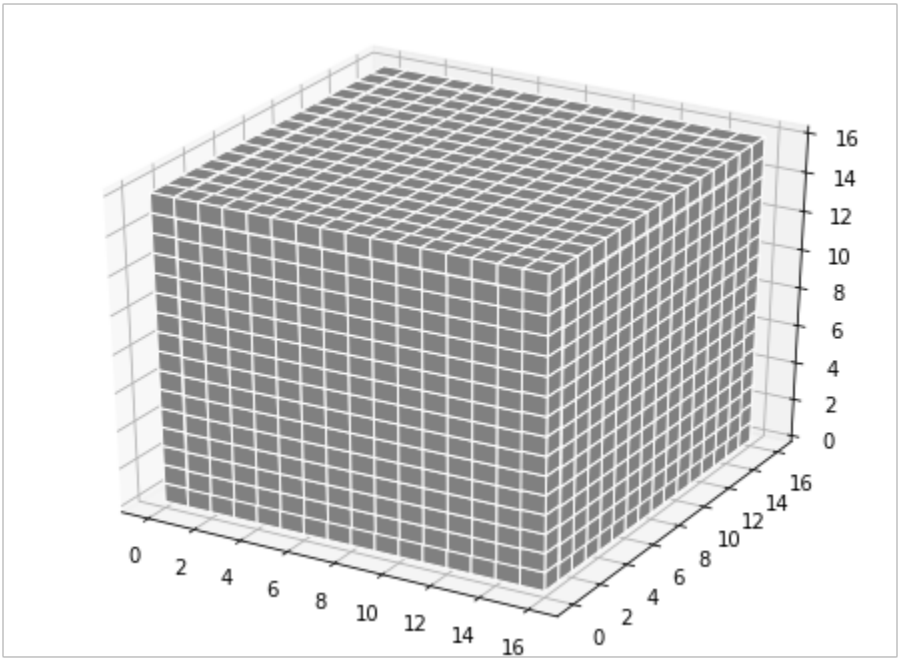
When we think about why should we choose Hazelcast when we need an IMDG we should have in mind the following aspects:

- **Marked leader**: Hazelcast is the marked leader among In-Memory Data Grid solutions.

- **Rich API:** it has a rich API and provides clients in various programming languages such as Java, C#.NET, Python, and so on. It has all the powerful features of an IMDG and a huge open source community.

- **Ease of use:** it has legendary ease of use because it's basically a simple key-value data store. The data structures are standard ones like Map, List, or Queue (in Java there are just other implementations of the standard Java interfaces for java.util.List or java.util.Map). Also, the redundancy and scaling aspects are built-in.

- **Distributed data store & computation system:** Hazelcast it's both a distributed data store and a distributed computation system. We can leverage it in both of these ways.



Hazelcast Cluster Discovery

As we have already seen, when talking about an IMDG, a very important aspect is the Cluster Discovery mechanism. Hazelcast provides multiple methods that can be used:

- **TCP/IP multicast discovery** which is not recommended in a production environment;
- **TCP/IP unicast discovery** which implies managing a static list of addresses of some members of the cluster;
- **Various discovery plugins for cloud**, like Eureka, Zookeeper, Kubernetes, Openshift, Pivotal Cloud Foundry, Google Cloud Platform, Amazon Web Services, Microsoft Azure;
- **Implement a custom discovery mechanism** via Discovery Service Provider Interface (SPI).

User code deployment

Another important and powerful feature of Hazelcast is user code deployment.

This feature allows us to load pieces of code from the client to the cluster members. Basically, we can have code that can be viewed as a task on the client-side and when we want to execute that piece of code, we can serialize it and execute on the cluster members without having it in the classpath of these.

The feature is not enabled by default and requires some specific configurations to be done, both on the client-side and on the cluster members.

On the client-side project, it's necessary to add all the classes that we want to load into the cluster member and also to specify the classload that owns these classes.

On the cluster member side, it's enough to enable the user code deployment feature.

Hazelcast-Spring

Hazelcast-Spring is another module provided by Hazelcast when we want to use it inside Spring Framework projects.

This module can be used by adding *hazelcast-spring* as a dependency to our Gradle or Maven project. It's worth mentioning that *hazelcast-spring*'s version is not the same as Hazelcast's version.

This module, together with the user code deployment feature allows us to use the Dependency Inversion Principle for integrating the client and the server of Hazelcast.

Basically, on the client-side, we can use an abstract class or an interface, which can be injected using the dependency injection provided by Spring, without having an actual implementation of this

abstract class or interface into our client. The class where the dependency is injected it's annotated with the *@SpringAware* and will be serialized and executed into the cluster member.

On the cluster member side, we must provide at least an implementation of this abstract class or interface and create a Spring Bean of this type.

When the serialized class will be executed on the cluster member side, the actually implemented dependency will be injected by the Spring IOC container.

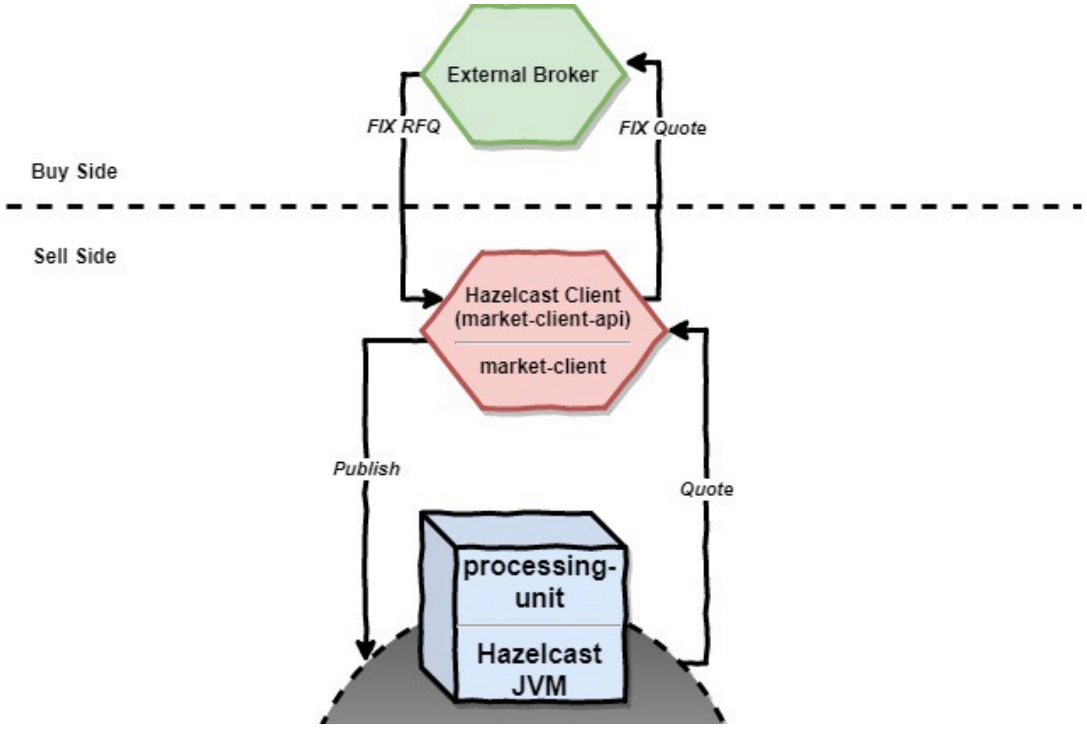
Example project

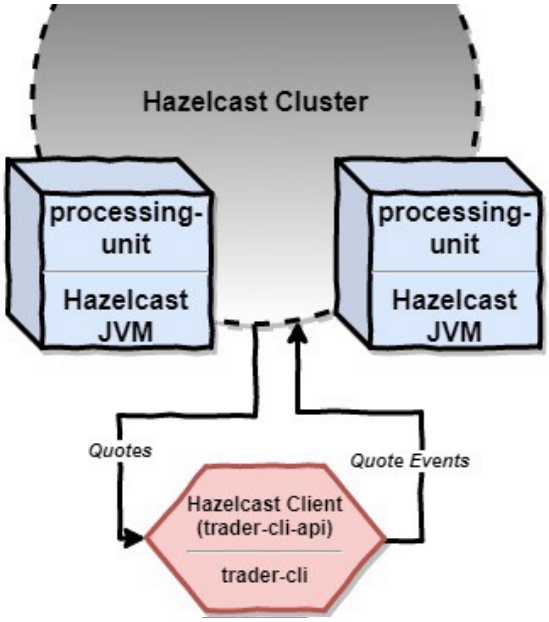
In the following, we will see a real example where Hazelcast is used in combination with Spring Boot, leveraging some features specific to an IMDG along with Hazelcast-Spring and Hazelcast user code deployment.

- **Business scenario**

  We will use Hazelcast IMDG for developing a Foreign Exchange Quotation Management System.

  The High-Level Architecture looks like the following:





- **System's components**

  The system developed consists of:

1. Two Spring Boot microservices: market-client and trader-cli which are basically Hazelcast clients and are communicating with the grid via APIs, specifically market-client-api and trader-cli-api.
2. One Spring Boot microservice called processing-unit which is basically a Hazelcast server member that will join the cluster when started.

   The market-client is used for publishing quotation prices from an external broker into the grid. Also, it should be used for publishing transactions that are made into our system back into the external broker.

   The trader-cli is a client used by trader to buy or sell FX pairs.

- **Hazelcast features used**

  The current example uses the Client-Server deployment model, leveraging the TCP/IP unicast cluster discovery mechanism. Also, there are both replicated and partitioned data structures used.

  Conclusions

- We have seen what an In-Memory Data Grid (IMDG) is and also

what are its characteristics;

- There are some benefits that can be obtained by using an IMDG, like performance increase, usage of simple data structures, or operations simpleness;
- It were presented the two major use-cases for using Hazelcast: as a data cache or as a data service fabric, along with some examples from the industry;
- There was presented both concepts of replication and partitioning along with the consequences of each of them;
- We have seen which are the deployment models for an IMDG;
- There were presented the characteristics of Hazelcast IMDG as a concrete implementation of the IMDG concept and also why should we choose Hazelcast when we need an IMDG;
- We have seen two special features of Hazelcast: user-code deployment and hazelcast-spring module and how we can use both of them into a concrete implementation.

Don’t forget to follow me on Twitter!