




Leveraging In-Memory Data Grid

Hazelcast IMDG

SWAT / Dina Bogdan

July 2020

 @dinabogdan03



do your thing

Agenda

1. What is In-Memory Data Grid (IMDG)?
2. Hazelcast IMDG
3. Cluster Discovery
4. Partitioning & Replication
5. Data Structure Overview
6. User-Code Deployment & Hazelcast-Spring
7. Demo time!

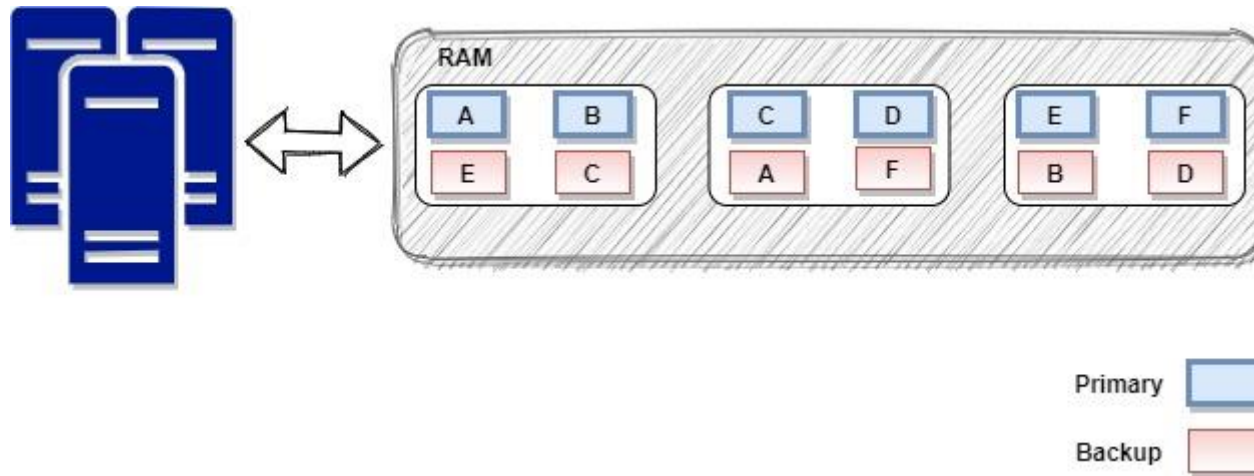
What is an In-Memory Data Grid (IMDG) ?

A **Data Grid** is a system of multiple servers that work together to manage information and related operations in a distributed environment.

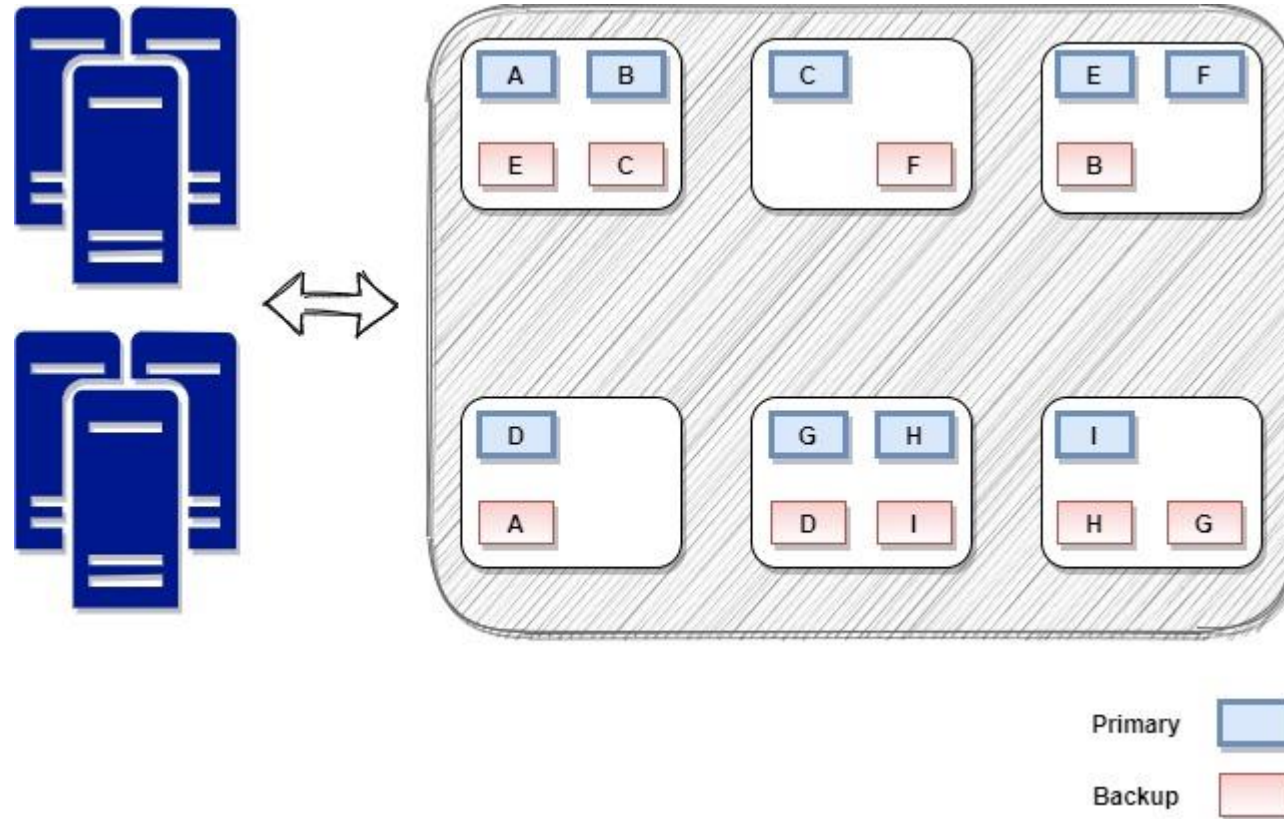
The servers from the **grid** can be **located** in the **same location** *or* **distributed across multiple data centers**.

An **In-Memory Data Grid** is a grid that **stores** data **entirely into RAM**.

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

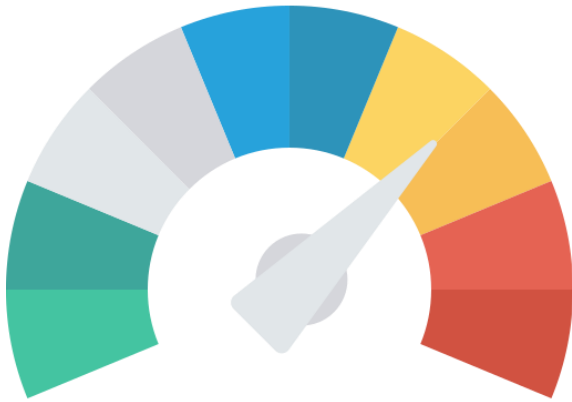


What is an In-Memory Data Grid (IMDG) ?



Why to use an In-Memory Data Grid?

Performance



- Access data 1000x faster than a database
- Low latency for batch and stream processing

Data Structure/Handling



- Non-relational key-value
- ACID compliance

Operations



- Scalability
- Redundancy for HA

When to use an In-Memory Data Grid?

Data Cache

- Eliminates data store bottlenecks
- Eliminates slow network connections
- Long-running blocking calculations

Data Service Fabric

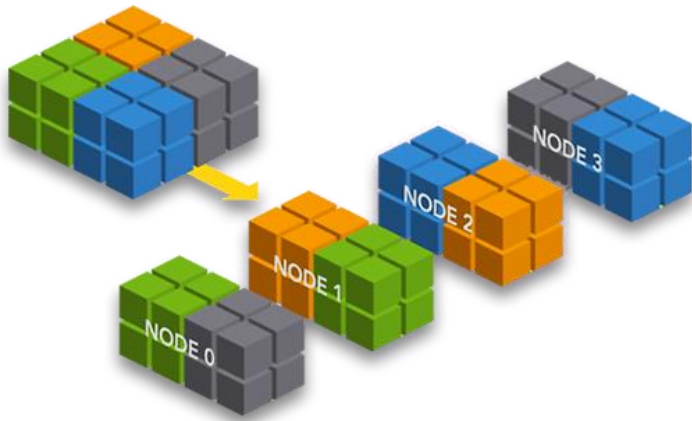
- Real-time integration
- Compute grid
- Message broker

Examples

- Analytics (Risk, Fraud-detection)
- Trading Systems (FX Trading, Stock Exchange)
- eCommerce
- Online Gaming

Basic operations of an In-Memory Data Grid

Cluster



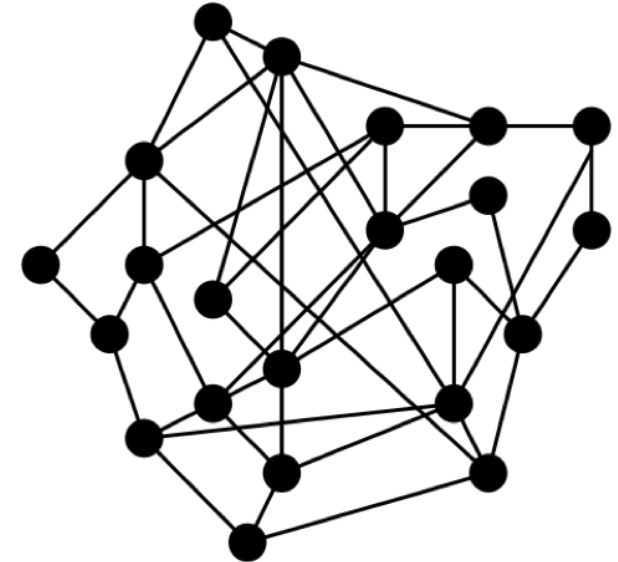
- Distributed data
- Highly scalable
- Fault tolerance

Discovery



- Form
- Find
- Join

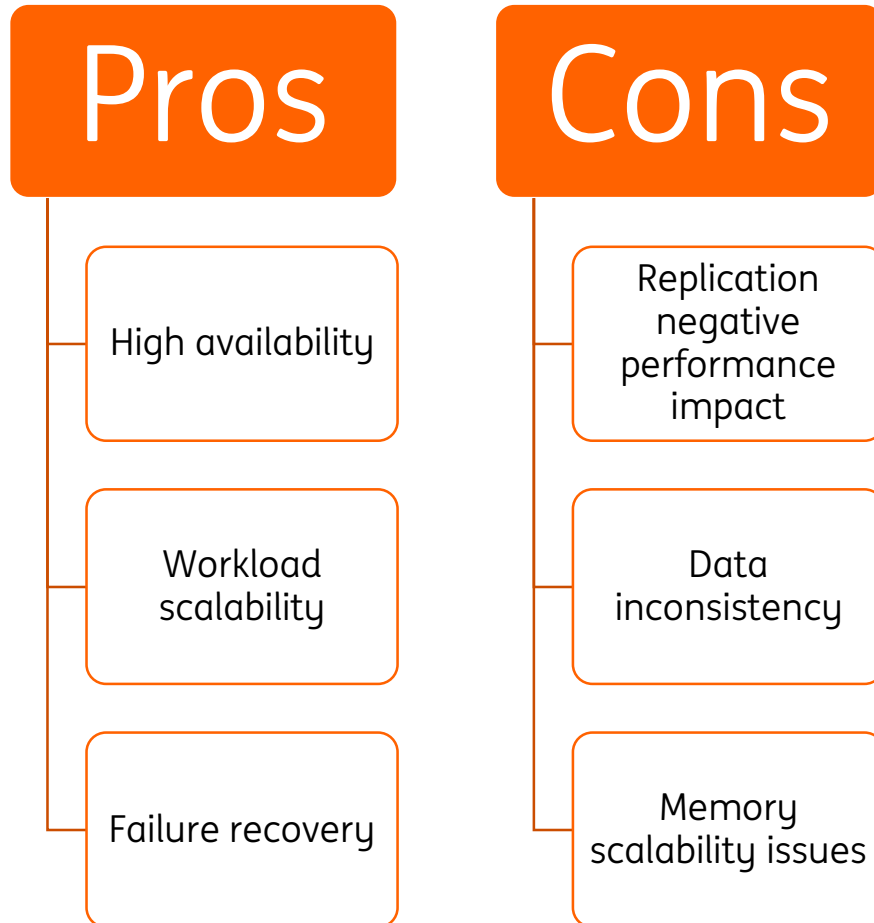
Data distribution



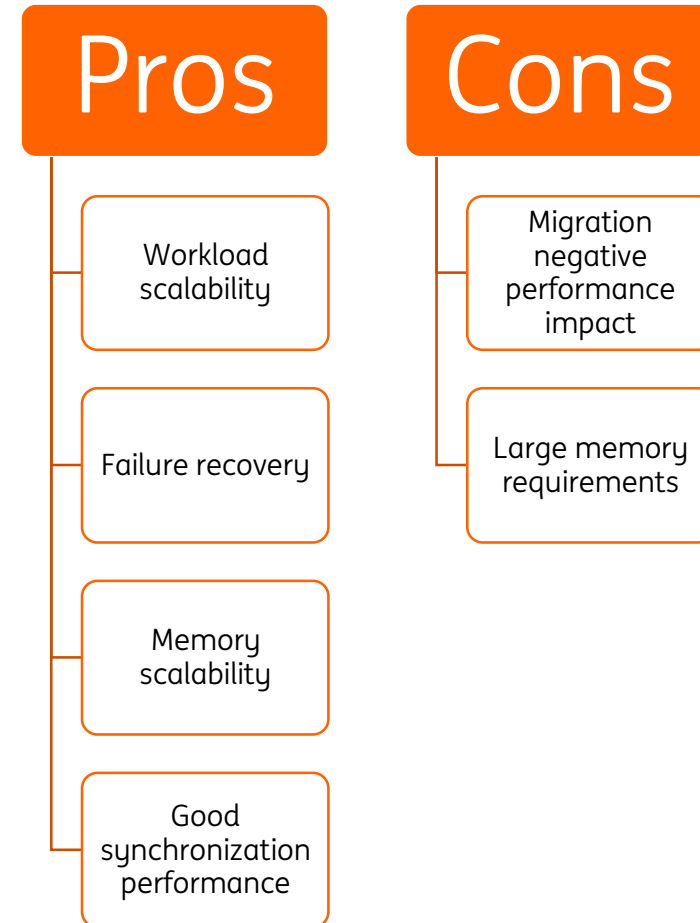
- Replication/Mirroring
- Partitioning/Sharding

Replication vs Partitioning

Replication

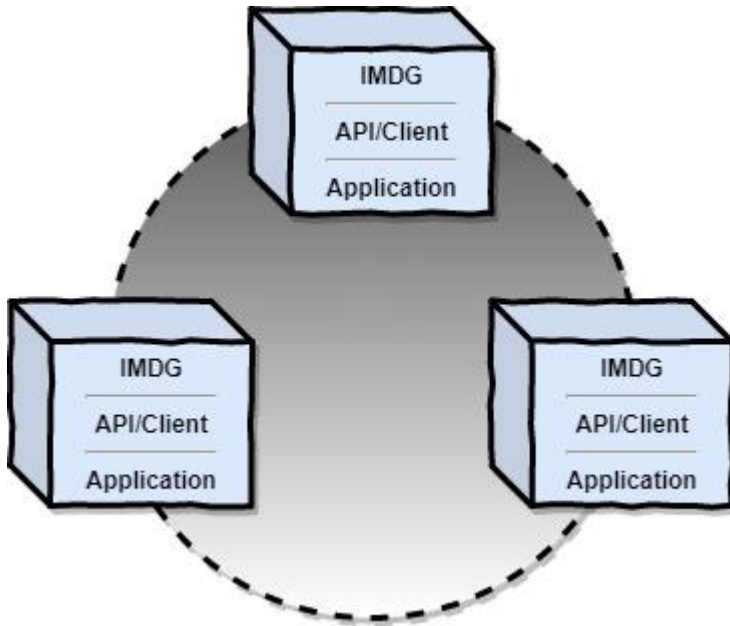


Partitioning

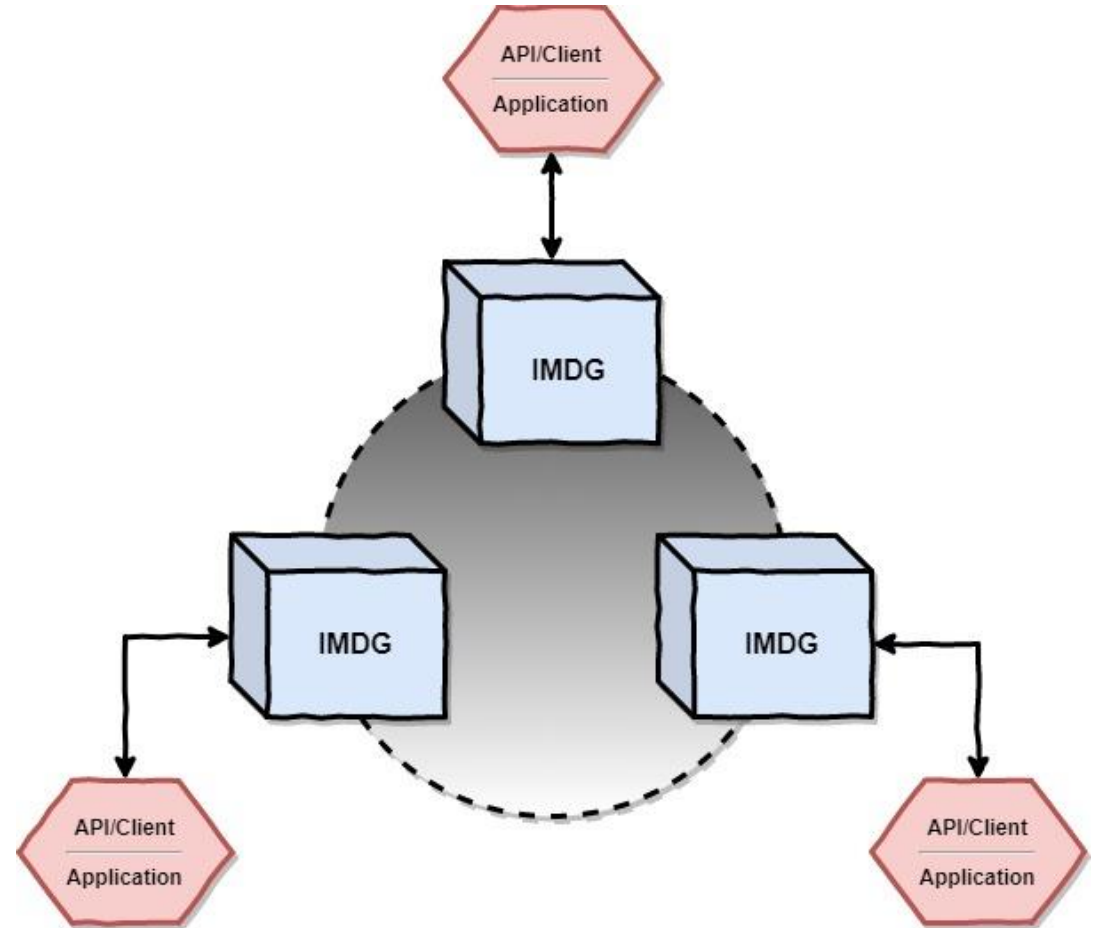


Deployment options

Embedded IMDG



Client-Server



Hazelcast IMDG

Characteristics

Why to choose Hazelcast IMDG?

Market Leader



- Hazelcast IMDG is market leader among In-Memory Data Grid solutions

Rich API



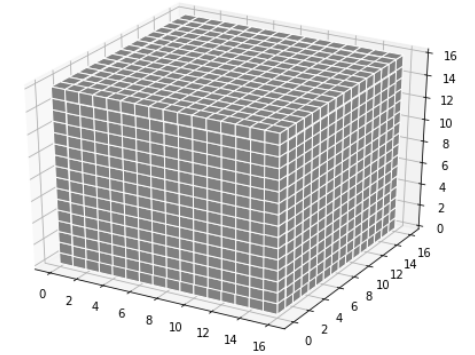
- APIs in various programming languages: Java, C#.NET, Python, etc.
- Powerful features
- Huge user base – open source project

Ease of use



- Simple to use key-value data store
- Standard data structures: Map, List, Queue, etc.
- Clients for many programming languages
- Redundancy/fail-over/scaling built in

Distributed data store & computation system



- Distributed data store
- Distributed computation near stored data

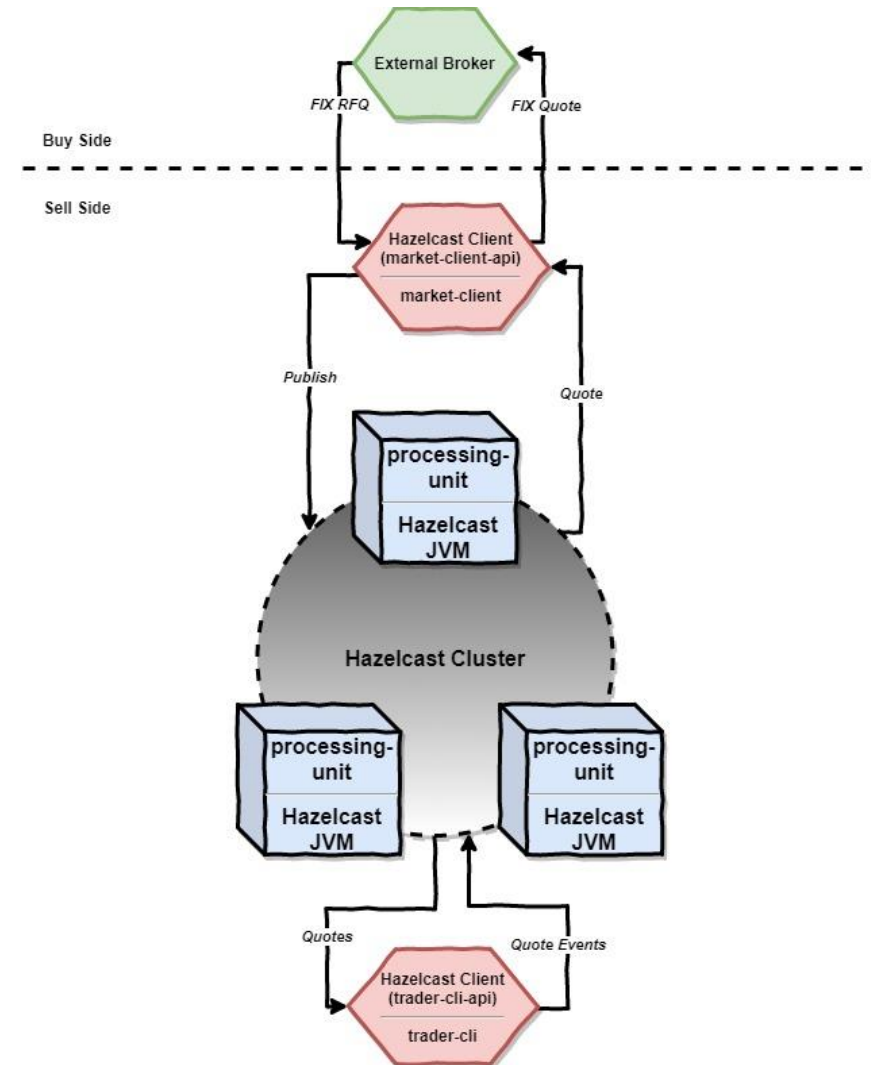
Business scenario and HLA - overview

Business scenario:

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

The system is consisting of:

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



Hazelcast features used

Deployment model:

- Client-Server

Cluster discovery mechanism:

- TCP/IP unicast discovery

Data structures used:

- Replicated Map
- Partitioned Map

Message broker:

- MapEntryListener

Client-Server Deployment Model

Hazelcast Client

For creating a Hazelcast Client Java application we must add the following dependencies:

- Prior to **Hazelcast 4.x**:
 - `com.hazelcast:hazelcast:3.x`
 - `com.hazelcast:hazelcast-client:3.x`
- For projects which are using **Hazelcast 4.x**:
 - `com.hazelcast:hazelcast:4.x`

Hazelcast Cluster Member

For creating a Hazelcast Cluster Member Java application we must add the following dependency:

- `com.hazelcast:hazelcast`

Hazelcast Cluster Discovery

There are multiple ways to establish a discovery mechanism inside our Hazelcast cluster:

- TCP/IP multicast
- TCP/IP unicast
- Discovery plugins for Cloud:
 - Eureka
 - Zookeeper
 - Kubernetes
 - OpenShift
 - Pivotal Cloud Foundry (PCF)
 - Google Cloud Platform (GCP)
 - AWS
 - Azure
- Custom discovery mechanism via Discovery SPI

Hazelcast Cluster Discovery

In our cluster member we use **TCP/IP unicast discovery**.

In *com.ing.trading.fx.processingunit.infrastructure.imdg.IMDGConfiguration.kt*:

```
import com.hazelcast.config.*

val config: Config = Config()
config.networkConfig.join.tcpIpConfig.isEnabled = true
config.networkConfig.join.multicastConfig.isEnabled = false
config.networkConfig.join.tcpIpConfig.members = listOf("localhost:5701", "localhost:5702")
```

Hazelcast Replicated Map

In the **processing-unit** Hazelcast Cluster Java application we are using a **replicated map** data structure for storing the quote prices published by the **market-client**.

The map must have a name, which in our case is “**QUOTES_MAP**” and is stored in binary format in each cluster member instance.

In *com.ing.trading.fx.processingunit.infrastructure.imdg.IMDGConfiguration.kt*:

```
private fun Config.addQuotesMap() {  
    val quotesMapConfig: ReplicatedMapConfig = this.getReplicatedMapConfig("QUOTES_MAP")  
    quotesMapConfig.inMemoryFormat = InMemoryFormat.BINARY  
}
```

Hazelcast Partitioned Map

In the **processing-unit** Hazelcast Cluster Java application we are using a **partitioned map** data structure for storing the all commands (Buy and Sell) published by each trader (**trader-cli** microservice).

In *com.ing.trading.fx.processingunit.infrastructure.imdg.IMDGConfiguration.kt* :

```
private fun Config.addTraderHistoryMap(imdgProperties: IMDGProperties) {  
    val traderHistoryMapConfig = MapConfig()  
    traderHistoryMapConfig.name = "TRADER_HISTORY_MAP"  
    traderHistoryMapConfig.backupCount = 2  
    traderHistoryMapConfig.timeToLiveSeconds = 3600  
    traderHistoryMapConfig.evictionConfig.evictionPolicy = EvictionPolicy.NONE  
    traderHistoryMapConfig.evictionConfig.maxSizePolicy = MaxSizePolicy.PER_NODE  
    traderHistoryMapConfig.evictionConfig.size = imdgProperties.maxSize.toInt()  
  
    this.addMapConfig(traderHistoryMapConfig)  
  
}
```

User code deployment

Hazelcast-Spring

User code deployment

- Not enabled by default.
- Allows us to load client classes inside cluster members.
- There are necessary configurations that must be done in both the client and the cluster member.

User code deployment

Client configuration

```
private fun ClientConfig.enableUserCodeDeployment() {  
    this.userCodeDeploymentConfig.isEnabled = true  
  
    this.userCodeDeploymentConfig.addClass(BuyTask::class.java)  
    this.userCodeDeploymentConfig.addClass(BuyTask.Companion::class.java)  
    this.userCodeDeploymentConfig.addClass(SellTask::class.java)  
    this.userCodeDeploymentConfig.addClass(SellTask.Companion::class.java)  
    this.userCodeDeploymentConfig.addClass(QuotesMapEntryListener::class.java)  
  
    classLoader = TraderCliApplication::class.java.classLoader  
}
```

Cluster Member Configuration

```
val config = Config()  
config.userCodeDeploymentConfig.isEnabled = true
```

Hazelcast-Spring

- com.hazelcast:hazelcast-spring
- Dependency Inversion
- *@SpringAware*

```
package com.ing.fx.trading.tradercli.infrastructure.pu

import com.hazelcast.spring.context.SpringAware
import com.ing.fx.trading.tradercli.api.model.BuyCommand
import com.ing.fx.trading.tradercli.api.model.BuySucceeded
import com.ing.fx.trading.tradercli.api.service.Trader
import org.springframework.beans.factory.annotation.Autowired
import java.io.Serializable
import java.util.concurrent.Callable

@SpringAware
class BuyTask(
    private val command: BuyCommand
) : Callable<BuySucceeded>, Serializable {

    companion object {
        private const val serialVersionUID = -3213576961319161714L
    }

    @Autowired
    @Transient
    lateinit var trader: Trader


    override fun call(): BuySucceeded {
        return trader.buy(command)
    }
}
```

Demo Time !

Thank you !

Source-code available at:



Follow me on Twitter:  @dinabogdan03.

Poll:

