hazelcast IMDG

Getting Started with Microservices Using Hazelcast IMDG and Spring Boot

White Paper

By Neil Stevenson
Senior Solution Architect, Hazelcast

hazelcast

Getting Started with Microservices Using Hazelcast IMDG and Spring Boot

Microservices, as an architectural approach, has shown a great deal of benefit over the legacy style of monolithic single applications. Nevertheless, microservices are not without their drawbacks.

The purpose of this white paper is to show the first steps for using Spring Boot and Hazelcast IMDG® and how they contribute to the microservices landscape, enhancing the benefits and alleviating some of the common downsides.

What is Spring Boot?

Spring Boot is part of the Spring Framework family of modules that replace boilerplate code in Java applications.

Specifically, Spring Boot is concerned with building "bootable" applications, ones which can boot up on bare metal or virtualized infrastructure, without any elaborate ecosystem such as application servers or web containers.

What is Hazelcast IMDG and why Microservices ?

Hazelcast In-Memory Data Grid (IMDG) is a group of processes that join together to share responsibility for data storage and processing. Data is stored in memory, so access times can be hundreds or thousands of times faster than disk. Data is spread across the processes, so capacity can be varied by simply adding or removing processes without any outage.

For microservices, what IMDG brings is communal storage that is fast, scalable and resilient.

Hazelcast IMDG is simple — a single JAR with no external dependencies.

One of the core values of Hazelcast® from its inception was simplicity and ease of use, and that still exists today. When you move an application or system into a distributed environment, having a simple and configurable backbone makes the task a lot easier.

You can leverage Hazelcast because of its highly scalable in-memory data grid (e.g. IMap and JCache). Hazelcast also supports other standard JDK Collections such as Lists, Sets, Queues and some other constructs such as Topics and Ringbuffers that can be easily leveraged for inter-process messaging. All of these attributes can offer functionality that is highly desirable within a microservices platform.

I live by the rule that if I can't get any software running within 15 minutes, I move on. Hazelcast meets this standard.

Simplistically, in a grid of 10 processes each holds 1/10th of the data copies and backups.

Should more be needed, 2 more processes could be added, the data is automatically rebalanced, with the net effect that each now holds 1/12th of the total data, and capacity has increased by 20%. A configurable level of data mirroring within the grid protects from data loss should a process fail.

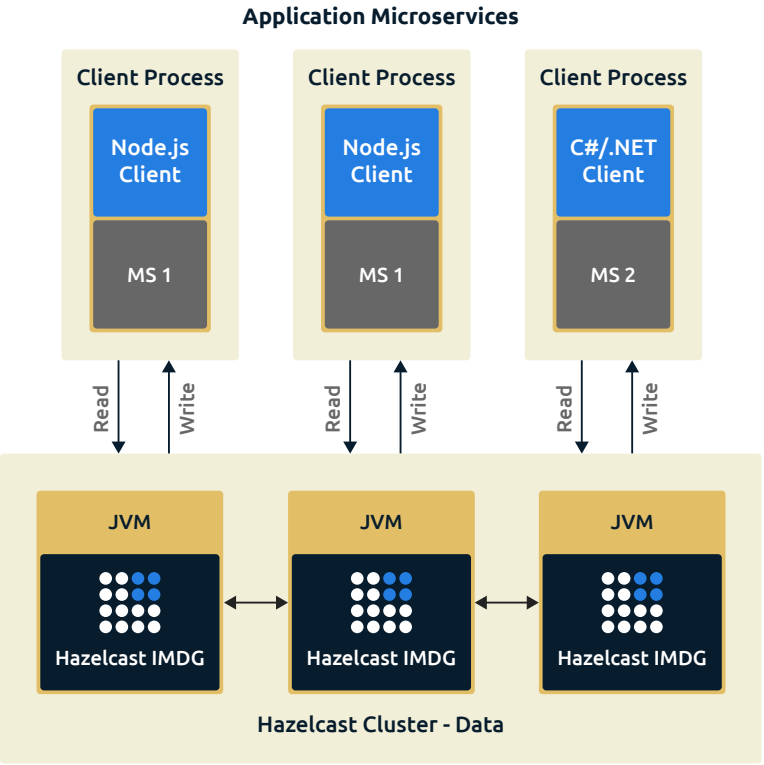

Figure 1: Data sharing across different microservices

Although the grid can do processing, for a microservice architecture it is more of a shared store. The clients that share it, the microservices, can be in a variety of languages including Node..js, Golang, C#/.NET, C++, Python, and of course Java and Scala.

Spring Boot and Hazelcast IMDG

Let's look at how we can incorporate Hazelcast IMDG and Spring Boot into a microservices platform.

As a Java program, the following is enough to create a standalone "bootable" jar file that can be run from the command line. It won't do anything, but it will run successfully.

```
@SpringBootApplication
public class Application {
   public static void main(String[] args) {
      SpringApplication.run(Application.class, args);
   }
}
```

Hazelcast is usually configured from XML. To turn that "bootable" jar into a Hazelcast server, all that needs done is to add a file 'hazelcast.xml' to the classpath with this content:

```
<?xml version="1.0" encoding="UTF-8"?>
<hazelcast xsi:schemaLocation=
   "http://www.hazelcast.com/schema/config hazelcast-config-3.8.xsd"
      xmlns="http://www.hazelcast.com/schema/config"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
</hazelcast>
```

Spring Boot relies heavily on auto-configuration. If it finds a Hazelcast IMDG configuration file such as the above, and has the Hazelcast classes as dependencies, it does the necessary boilerplate steps to create a Hazelcast IMDG server instance, configured from that file.

So these two components are enough to create a standalone Jar file containing a full Hazelcast IMDG server that can participate in an IMDG grid.

This is a production quality Hazelcast IMDG server, run as a Spring '@Bean' in an executable Jar file that can be pushed out to run on any host with a JRE. It's one file, easy to deploy.

It is the minimum set-up, but extension is equally easy.

Six Problems

Now it's clear how easy it is to set up a Hazelcast IMDG with Spring Boot, it is time to return to microservices in general, to see how this is going to be useful.

There are 6 main problems that Spring Boot and Hazelcast IMDG help solve.

1. Sharing
2. Asynchronicity
3. Security
4. Simplicity
5. Evolution
6. Health

Let's look at each in turn, using the common scenario of online shopping, where the customer adds items to a virtual basket over time and hopefully decides to buy these.

Problem 1: Sharing

What could be a monolith has been split into multiple microservices, and for highest availability each microservice will usually be run as a number of instance clones.

For an online shop, basket operations are crucial, and there might be a microservice responsible for adding and removing items. Rapid response for basket operations is obviously crucial, so to cope with surges in the number of customers such as Black Friday/Cyber Monday it will likely be necessary to run multiple instances of this microservice.

This presents a problem in where this core piece of data, the basket, should be held.

The basket could be held in the memory of the microservice instance. This would be fast, but would require a request routing "affinity" system, so that each customer is pinned to the microservice instance that hosts their basket.

Affinity is convoluted, as the microservice itself is not the outmost point in the web stack that the user interacts with, and there could be a chain of microservices involved and routing information has to be passed all the way through. If the microservice instance goes offline, the baskets held on that instance are lost, which immediately represents lost revenue plus the poor user experience of their basket vanishing might lead them not to use this website again.

A RDBMS is another choice. All instances of a microservice could reach this to save and retrieve the basket, so wouldn't need to store it in their memory.

Problems here are more around speed, can the database cope without hundreds, thousands or more updates per second at peak time. Plus if the database fails, as a single component all baskets are lost which is worse than losing a share of them.

Plus from an architecture perspective, baskets are transient information, not really needing disk persistence.

The solution of course is a bit of both. Use memory storage for speed, but distribute for scaling and resilience. This is what a Hazelcast IMDG does.

With code such as this, a reference to a remote storage container (that behaves like a Java map) can be obtained and used.

```
IMap<String, String> m1 = this.hazelcastInstance.getMap("basket");
m1.put("user1", "apples, celery, walnuts, grapes");
```

As previously mentioned, a Hazelcast IMDG grid is scalable and robust. While basket expiry is an option, if the number of live baskets grows more IMDG servers can be added to accommodate extra data, and this may ultimately be more profitable than the traditional method of expiring baskets to make space. Baskets contain things that customers are thinking of purchasing, they should be given every chance to spend money.

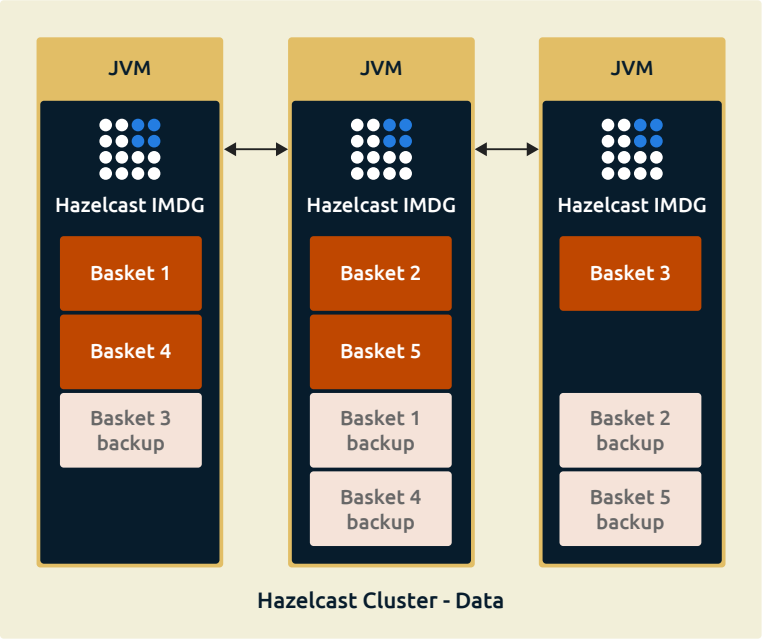

Figure 2: Data sharing across different microservices

As a final point, analytics may be interested in what is removed from baskets, in case trends are visible. This wouldn't really be an appropriate task for the basket microservice to do, as it would be crossing business domains and so bad design, but it could send the change event asynchronously to another microservice that does as, if the next problem is addressed.

Problem 2: Asynchronous Communications

Loose coupling and fault tolerance are important design goals for microservices. Microservices can invoke each other, but should not rely on the called process responding quickly or at all. Consequently, asynchronous communications play a key part in inter-process communications.

Consider again the online shopping example. At some stage, the customer is happy with the basket contents, shipping, delivery, and so on and presses the “Buy” button.

Multiple things need to happen, spanning several business domains. For example, a confirmation email needs to be sent, stock needs to be dispatched, more stock may need to be ordered, and of course payment needs to be collected. This requires the orchestration of several actions, and these actions would be best handled by individual modules such as microservices.

Taking the confirmation email part, it would be naive to do this synchronously. The customer is not waiting for the email to be received before the order is confirmed, and even sending synchronously does not guarantee deliver given the store-and-forward model that underpins email. Far worse would be to have to abandon the sale because the mail server is down for maintenance so breaking the chain of steps that must complete, and complete quickly.

Naturally here the solution is asynchronous, to use some sort of messaging solution to request the email be sent so that processing of the order commitment can progress to the next part, which here is initiating dispatch.

Hazelcast IMDG supports named Queues and Topics, for point-to-point and publish-subscribe messaging. Handles onto these are easily obtained and used, for example:

```
IQueue q1 = hazelcastInstance.getQueue("q1");
ITopic t2 =hazelcastInstance.getTopic("t2");
q1.put("hello");
t2.publish("world");
```

The key attraction here is the simplicity, making this a very lightweight ESB, especially if the Hazelcast IMDG grid is already present to solve the proceeding problems.

A Queue is requested by name, as you can have several. If the Queue doesn’t exist, it is built on demand, no additional configuration is necessary. Reading from these Queues and Topics is similarly easy, and can be done from the same or other microservices.

```
Object o1 = q1.poll();
MessageListener messageListener = new MessageListener() {
   @Override
   public void onMessage(Message message) {
      Object o2 = message.getMessageObject();
   }
};
t2.addMessageListener(messageListener);
```

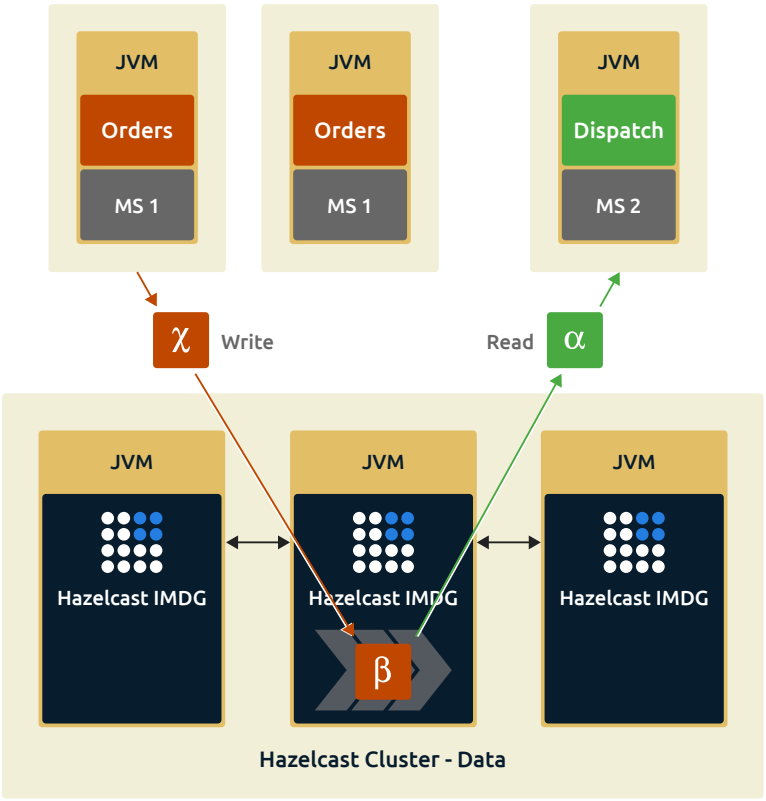

Figure 3: Message passing from microservice to microservice

Although this massively simplifies asynchronous communications, all problems do not go away.

The astute reader will have spotted that although the object sent and received doesn't have to be a string, this still introduces a coupling between the sender and receiver — both sides have to understand the format of the message.

Whilst this common message format is unavoidable, great care must be taken to ensure the format remains as neutral as possible and that application specifics don't "leak" into the format. For example, use of `"YY-DD-MM"` date format pre-supposes the USA and stops the whole application from evolving in stages to support other countries.

Problem 3: Security

The rule in microservices is isolation, isolation, isolation. The exception to this rule is authentication.

For a web facing application some sort of login is inevitable. However, if internally there are multiple microservices then to log in to each would be a poor user experience.

Fortunately, Spring does most of the work here, providing easy set-up to a widely used and solid security implementation. Following the pattern above, the single addition of `spring-boot-starter-security` to the dependencies is sufficient to include all the necessary modules for web based security authentication.

Configuration follows a more cautious approach, which seems reasonable. The application coder has to specify which URLs are protected. Coding here could be as simple as:

```
@EnableWebSecurity
public class MySecurityConfig extends WebSecurityConfigurerAdapter {
   @Override
   protected void configure(HttpSecurity http) throws Exception {
      http
      .authorizeRequests()
         .antMatchers("/**").hasRole("USER")
```

The above would secure every URL (the `"/**"` matcher line). It's likely more configuration than this would be needed, for example to restrict monitoring URLs to selected users, but this should not run to many lines.

One further piece of magic brings Hazelcast IMDG into play. Adding this annotation

```
@EnableHazelcastHttpSession
```

directs Spring Boot's `spring-boot-starter-security` to use Hazelcast IMDG to store the web session in one of Hazelcast's distributed maps.

This simple change brings a lot to microservices. Session storage is offloaded to the Hazelcast IMDG grid, bringing the benefit of scalable storage, resilience and memory speed of access.

Really though, the benefit is accessibility. This session is available to all microservice instances and to all microservices, so a signed-in user is authenticated across all parts of the application, whether disseminated or not.

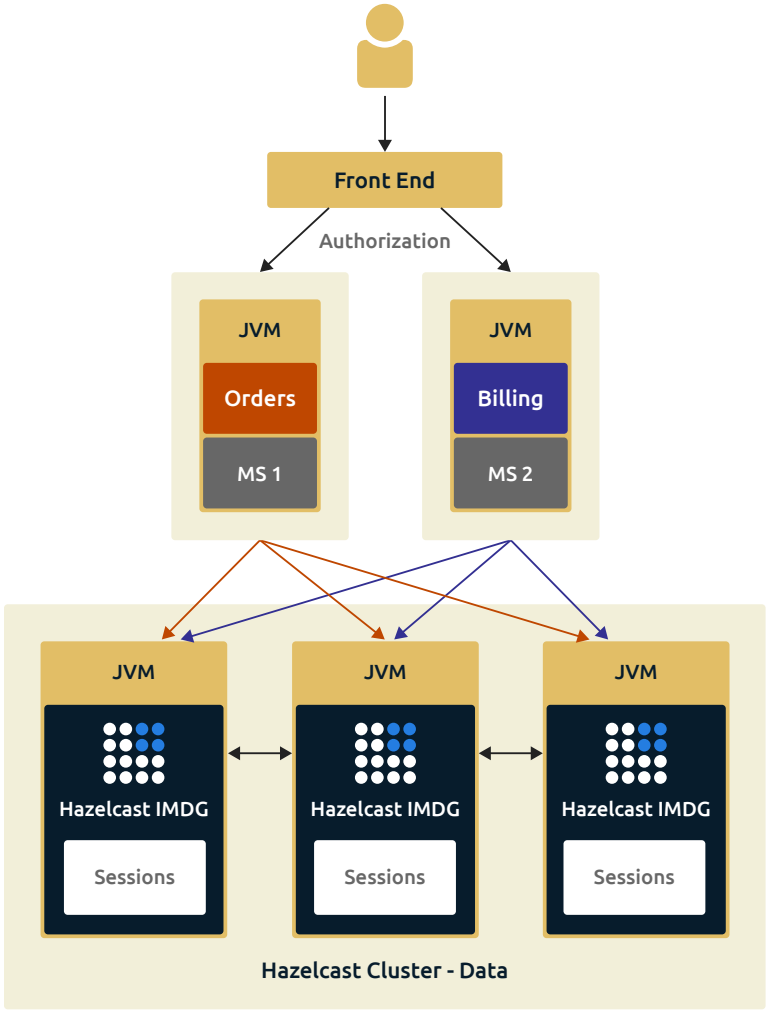

Figure 4: Common security store

In practice, this won't quite be as simple as a one line change to the code. But if more than ten lines need written then that too would be unrealistic.

The counterpart to authentication is authorisation, and the same rules do not apply. While we wish the one sign-on to be shared across all microservices, access rights can vary. For the online shop application, an order placement microservice needs write access, but an order history microservice should only be awarded read access.

As a recap on this point, the one data model should not be used by two or more applications, as this introduces a tight coupling. Overriding this goal for security is viewed as tolerable for the wider benefit of single-sign-on, since this data model won't evolve.

Problem 4: Simplicity

Simplicity is a hidden problem. A clean design is needed, and this will require a range of technologies to be used, such as JSON for a message interchange format, REST for transport, and Hazelcast IMDG for storage. The challenge is to do this without spending large amounts of time on pointless boilerplate code, when time can be more productively spent on business logic.

This is where Spring Boot brings the efficiency gains. A single dependency, on `spring-boot-starter`, is sufficient to build a standalone Java module. It is of course trivial to change this one dependency from `spring-boot-starter` to `spring-boot-starter-web`, merely 4 characters of typing. The effect though is that the Java module is now HTTP enabled, with an embedded web container listening on port 8080 and routing any incoming requests to user code.

Once `spring-boot-starter-web` is present, this would be enough to return JSON to a REST request for t he base url:

```
@RestController
public class MyController {
   @GetMapping("/")
   public List index() {
      return Arrays.asList("one", "two", "three");
   }
```

To extend this with Hazelcast connectivity is equally simple.

```
@Autowired
private HazelcastInstance hazelcastInstance;
@GetMapping("/")
public List index() {
   List result = new ArrayList();
   Object o;
   while ((o=hazelcastInstance.getQueue("q1").poll())!=null) {
      result.add(o);
   }
   return result;
}
```

The Queue example from the previous section now has it's contents available over REST.

Architecturally microservices should be independent, and as part of this independence instances should not be hosted in the same web container or application server as other instances. Each microservice should have it's own web environment, and since it is a 1:1 mapping it's simpler to bundle the microservice and web container into the same deployment unit, the jar file.

Problem 5: Evolution

Getting things right the first time is commendable, and even if achieved things won't remain correct. Applications have to change over time to suit new requirements.

Partly this is a solved problem for microservices with the loose coupling and fault tolerant principles. Microservices should be designed to handle other microservices that they call failing, and this can be exploited to simply bounce when the code needs to change.

The difficulty comes when the data format needs to change. As has been mentioned, common data structures between microservices is almost an anti-pattern, but the scenario here is a data structure isolated to one microservice.

For the online shop example, a field might need to be added to orders to record the dispatch date. Ordinarily you can't change the data model without change the data, and the time to drop and reload the data could be lengthy, as it depends how much data exists.

Hazelcast IMDG supports a `"VersionedPortable"` base class for data types. As the name suggests, this allows for versions of the data to exist concurrently. Metadata allows the new data model to be applied to the old data, any field that doesn't exist in the data but does exist in the data model is gets a default value.

If this is combined with a rolling upgrade across the IMDG cluster, then as the data is re-hosted from process to process, gradually the data is transitioned to the new format without any outage on cluster functionality. One IMDG server at a time is the usual method, but so long as sufficient processes are kept running, only capacity dips — the data hosted remains safely backed up and available.

Problem 6: Health

A final piece of the puzzle is health-checking. With so many more processes running for microservices than for a monolith, keeping track of any that are overloaded is all the harder.

Recalling again the design goal of fault tolerance, one failing microservice should not be catastrophic. But when the application as a whole moves into an impaired state, where a problem occurs and where a problem appears may be different.

For the online shop example, a failure on the email handler should be self-contained, order confirmations messages would just sit in the Hazelcast queue until it is recovered. If stock re-ordering is broken for a long time, the shop may run out of things to sell.

On Spring Boot, adding `spring-boot-starter-actuator` as a dependency automatically makes monitoring information available as REST endpoints on a microservice.

For example, a call to the endpoint `"/health"` on a running service will show:

```
{"status":"UP"}
```

And for rather more detail, unhide the `"/metrics"` endpoint to see more, such as:

```
{"mem":391240,
"mem.free":281696,
"threads":49
}
```

(There are many more measurements output, these have been omitted for brevity)

Naturally it's easy to enhance these metrics. This is what is needed to expose the backlog in the queue from the previous section.

```
@Component
public class MyMetrics implements PublicMetrics {
   @Autowired
   private HazelcastInstance hazelcastInstance;
   @Override
   public Collection metrics() {
      Collection metrics = new HashSet();
      IQueue q1 = hazelcastInstance.getQueue("q1");
      metrics.add(new Metric(q1.getName(), q1.size()));
      return metrics;
}
```

So this now shows

```
{"mem":391425,
"mem.free":280331,
"q1":0,
"threads":49
}
```

as the Queue `"q1"` is currently empty.

With delivery over HTTP (or HTTPS) and a REST/JSON style, such health information should easily dovetail into most monitoring solutions, and is easily interpreted by humans for adhoc queries.

Being able to simply extend the metrics to track items of specific interest to the application or any of the microservices makes it all the more powerful.

Other Considerations

Hazelcast Topology

Hazelcast IMDG server processes are easily wrapped by Spring Boot for simple deployment, monitoring and the like. As Java processes, they could easily also be microservices.

In general, this would be a bad idea. In specific cases, a very good idea.

The reasons follow, and while none are individually significant, there are more arguments for using Hazelcast IMDG's client-server topology then embedding the Hazelcast IMDG server in the same process as the microservice.

Simplicity

A microservice instance that embeds a Hazelcast IMDG server makes for less processes to deploy. If the architectural ideal that each process runs on a machine dedicated to it is affordable, then fewer modules require fewer hardware resources.

Scaling

The microservice might need a certain number of instances to cope with demand, and the Hazelcast IMDG grid needs a certain number of instances for capacity.

It is unlikely these two numbers start the same and stay the same. In client-server mode, they could be varied independently, but when embedded the larger number applies to both. While some spare capacity is sensible, excessive spare capacity is wasteful of resources.

Separation of Concerns

Data stored in a Hazelcast IMDG server is striped horizontally to give an even balance.

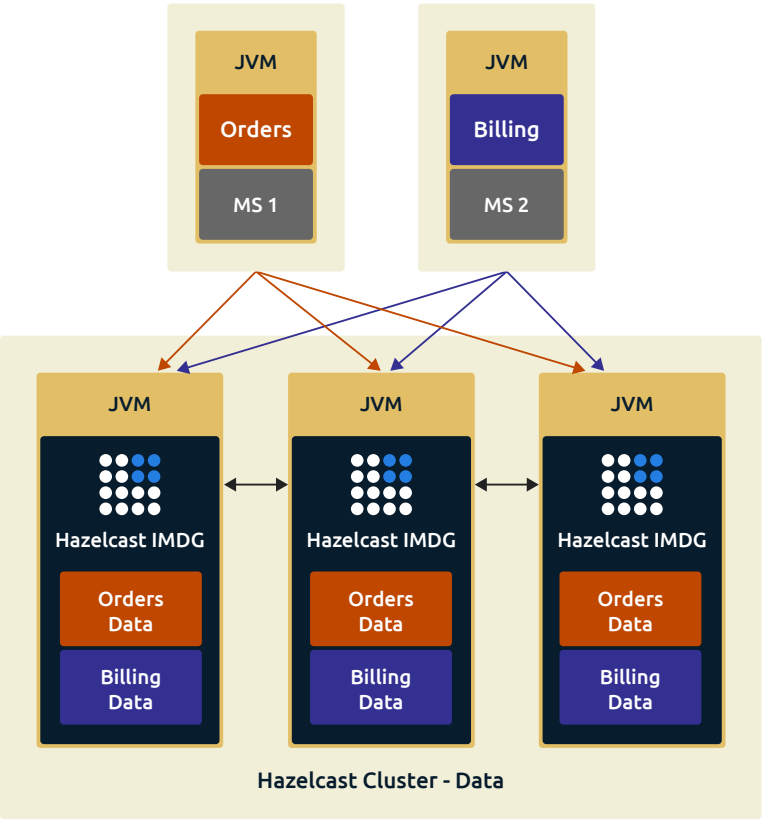

Figure 5: Horizontally striped data silos

If there are 3 Hazelcast IMDG servers, then each holds 1/3rd of the data. This could be 1/3rd of the data for microservice A, and 1/3rd of the data for microservice B.

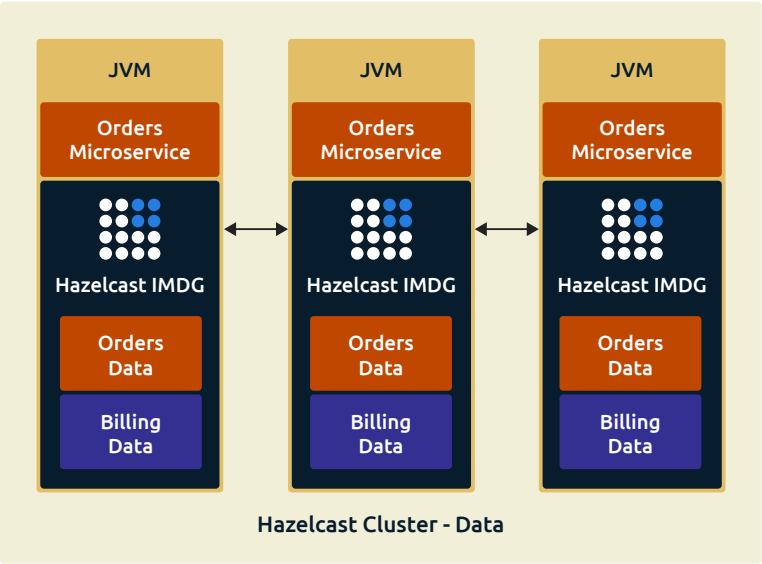

Figure 6: Embedded data silos do not isolate one microservice from another

If this Hazelcast IMDG server is embedded in microservice A, then this violates the separation of concerns tenet for microservices. Simply, microservice A is not isolated from the data for other microservices.

Rolling Upgrades

Fault tolerance is one of the usual goals of microservices. One or all of the instances of a microservice should be able to go offline, and other microservices should be able to continue operating. If they can't, that's usually an indicator that service isolation is not correctly designed.

This normally allows the total outage option when a microservice has to be upgraded. All instances of a particular microservice can be taken offline for upgrade, as all other microservices should continue to operate.

However, if the microservice embeds Hazelcast IMDG server and therefore data, taking a total outage on the microservice means a total outage on the Hazelcast IMDG grid. Being "in-memory" storage, contents are lost and would have to be reloaded.

To avoid this would require a rolling upgrade strategy for the microservice with the embedded Hazelcast IMDG server. This makes for a longer upgrade to do processes individually. During the rolling upgrade, both old and new versions of the microservice are live, which is an extra complication to accommodate.

Security

Proper security of data access is not possible if the data accessor (the microservice) and the data storage (Hazelcast IMDG server) exist in the same process.

Although Hazelcast IMDG can provide role-based authorization and access control, if the microservice is in the same process it could cheat by looking in the processes memory, bypassing any controls.

Java

The Hazelcast IMDG server is Java based. If this is hosted in the same process as the microservice, then that process has to be a JVM.

This forces the microservice to be a JVM compatible language, such as Java.

Polyglot Support

Although Spring Boot enhances Java development, Java will not always be the best choice for every microservice.

Hazelcast IMDG supports different options for clients that connect to it, currently Java, C#/.NET, C++, Scala, Python, Clojure, Golang, and Node.js, and this allows more flexibility. Microservices could be written in a mix of languages, rather than forcing all to be the same.

This is where the notion of a portable data type comes in. Data can be written by a microservice in one language and can be read by a microservice in another language. Yet again though, whether this should be done needs close attention. Sharing a data format is close coupling not loose coupling, and at the very least an area of the application's architecture where close scrutiny and validation are in order.

A more reasonable consideration is that data used by collection of .NET instances for one microservice can be hosted on the IMDG (which is Java). One .NET client creates an order for example, that is stored i n the IMDG grid, and later another .NET client for the same microservice retrieves and updates that order. This doesn't cause the data to cross boundaries between microservices, only between instances of a particular microservice.

Conclusion

Microservices as an architectural pattern has proven its worth the benefits of decomposition into manageable units outweigh the downsides of the same.

For Java developers, and Spring users in particular, Spring Boot is a major productivity gain. Convenient plugins and self-configuration achieve common tasks such as security, sessions, HTTP transport, monitoring and so on with ease.

It's heavily opinionated approach is ultimately a benefit, as applications end up with a homogenous pattern, whether microservice or monolith, that makes them easy to deploy and manage on emergent virtualization technologies.

Hazelcast has a more cross-cutting role, integrating easily with Spring Boot. That value is driven from the simple approach to solving the problem for sharing transient state amongst loosely coupled and evolving modules where the language choice may vary, with the bonus of a lightweight approach to asynchronous messaging between and across modules.

350 Cambridge Ave, Suite 100, Palo Alto, CA 94306 USA
Email: sales@hazelcast.com Phone: +1 (650) 521-5453
Visit us at www.hazelcast.com