This book is an easy-to-follow, hands-on introduction that guides you through this innovative new technology. It covers everything from data grids to the simple-to-use distributed data storage collections. Queuing and topic messaging capabilities, as well as locking and transaction support to guard against concurrency race-conditions, are some of the topics that we will cover. We will then move on to distributed task execution, in-place data manipulations and big data analytical processing using MapReduce.

At the end of all this, you will be armed with everything you need to bring amazing power and data scalability to your application, as well as making them truly global and ready for a worldwide audience.

Who this book is written for

This book is a great introduction for Java developers, software architects, or DevOps looking to enable scalable and agile data within their applications. Providing in-memory object storage, cluster-wide state and messaging, or even scalable task execution, Hazelcast helps solve a number of issues that have troubled technologists for years.

What you will learn from this book

- Learn and store numerous data types in different distributed collections
- Set up a cluster from the ground up
- Work with truly distributed queues and topics for cluster-wide messaging
- Make your application more resilient by listening into cluster internals
- Run tasks within and alongside our stored data
- Filter and search our data using MapReduce jobs
- Disclose the new JCache standard and one of its first implementations

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 8 'Typical Deployments'
- A synopsis of the book’s content
- More information on Getting Started with Hazelcast Second Edition
About the Author

Mat Johns is an Agile software engineer, hands-on solution architect, and a general technologist based in London. His experience with the Web reaches all the way back to his misspent youth and some rather hacktastic code, but eventually, he grew up to graduate from the University of Southampton with a master’s in computer science with distributed systems and networks. Since then, he has worked for a number of start-ups on various web projects and systems, and nowadays, he specializes in designing and creating high-performance and scalable web services, currently in the Internet TV world.

Away from technology, he is an avid explorer and endeavors to seek out new destinations and adventures as much as possible. He is also a qualified yacht skipper and regularly races in, around, and beyond the Solent.

You can follow him on Twitter at @matjohns.

Mat was also the author of the first edition of this book.
Preface

Hazelcast is an innovative new approach to data, in terms of storage, processing, and distribution; it provides an accessible solution to the age-old problem of application and data scalability. This book introduces this great open source technology in a step-by-step, easy-to-follow manner, from the why to the how and the wow.

What this book covers

Chapter 1, What is Hazelcast?, introduces the technology, its place in an application's stack, and how it has evolved from traditional approaches to data.

Chapter 2, Getting off the Ground, helps us start coding and get acquainted with the standard distributed data store collections on offer.

Chapter 3, Going Concurrent, looks at more distributed and concurrent capabilities that we can bring into our applications.

Chapter 4, Divide and Conquer, looks at how data is split up and split across many nodes to provide both performance and resilience.

Chapter 5, Listening Out, discusses how we can register to receive notifications from the cluster in order to enable our application to be aware of what's going on.

Chapter 6, Spreading the Load, moves beyond data storage and investigates the distributed execution service and how Hazelcast is more than just a database.

Chapter 7, Gathering Results, looks at MapReduce for analyzing and discovering hidden values within our raw data and how it searches and aggregates data.

Chapter 8, Typical Deployments, explores the various ways in which we can use or install Hazelcast into our application or infrastructure, looking at the architectural decisions, reasons, and trade-offs behind each.
Preface

Chapter 9, *From the Outside Looking In*, rather than the use of the provided drivers for integrating with a Hazelcast cluster, looks at the popular alternative access that we have to our data.

Chapter 10, *Going Global*, discusses how we can explode onto the world stage by using the public cloud infrastructure and asynchronous remote replication to take our data all around the globe.

Chapter 11, *Playing Well with Others*, brings the technology together with popular companion frameworks to see how we might start to bring the technology to work with legacy applications.

Appendix, *Configuration Summary*, provides an overview of the configurations that we have used throughout the book.
Typical Deployments

So far, we have been looking at Hazelcast in one particular type of deployment; however, there are a number of configurations that we could use depending on our particular architecture and application needs. Each deployment strategy tends to be best suited to a certain type of configuration or application deployment; therefore, in this chapter, we will look at the following:

- The issues of co-locating data too close to the application
- Client connectivity, where it’s best used and the issues that come with it
- Overview of the architectural choices

All heap and nowhere to go

One thing that we may have noticed with all the examples that we have been working on so far is that as we are running Hazelcast in an embedded mode, each of the JVM instances will both provide the application's functionality and house the data storage. Hence, the persisted cluster data is held within a heap of the various nodes, and this means that we will need to control the provisioned heap sizes more accurately, as it is now more than just a non-functional advantage to have more; that is, size matters.

However, depending on the type of application that we are developing, it may not be convenient or suitable to directly use the application's heap for storing data. A pertinent example of this situation would be a web application, particularly one that runs in a potentially shared web application container (for example, Apache Tomcat).
Typical Deployments

This would be rather unsuitable for storing extensive amounts of data within a heap, as our application's storage requirements would drastically increase and the amount of memory available to traditionally run our application would be reduced. We will either need to provision more web application containers or potentially put ours and other applications running within that container cluster at risk from excessive garbage collection, or worse still, running out of heaps altogether.

Stepping back from the cluster

To avoid this situation, we can separate our application from the data cluster through the use of a client driver that looks and appears very similar to a direct Hazelcast instance. However, in this case, the operations are performed and delegated to a wider cluster of real instances. This has the benefit of separating our application from the scaling of the Hazelcast cluster, allowing us to scale up our own application without having to scale everything together, thereby maximizing the utilization efficiency of the resources that we are running the application on. However, we can still scale up our data cluster by adding more nodes to avoid a bottleneck at this layer, either for increasing memory storage requirements or performance and computing necessities.

If we create a server-side vanilla instance to provide us with a cluster of nodes, we can connect out to/from a client.

```java
public class VanillaInstanceExample {
    public static void main(String[] args) {
        Config conf = new Config();
        HazelcastInstance hz = Hazelcast.newHazelcastInstance(conf);
    }
}
```

We can run this code a few times to establish a cluster of a number of instances:

```
Members [3] {
    Member [127.0.0.1]:5701
    Member [127.0.0.1]:5702
    Member [127.0.0.1]:5703 this
}
```
Now we need to bring in a new dependency, where our original downloaded archive is `hazelcast-client-3.5.jar`, and we can use this to create `ClientExample` to connect to the cluster and perform operations against the data that is held there. As the client is delegating the operations to a wider cluster, the data persisted will outlive the client.

```java
public class ClientExample {
    public static void main(String[] args) {
        ClientConfig conf = new ClientConfig();
        conf.getNetworkConfig().addAddress("127.0.0.1:5701");
        HazelcastInstance hzc = HazelcastClient.newHazelcastClient(conf);
        IMap<String, String> capitals = hzc.getMap("capitals");
        if (capitals.isEmpty()) {
            System.err.println("Empty capitals map, adding entries");
            capitals.put("GB", "London");
            capitals.put("FR", "Paris");
            capitals.put("US", "Washington DC");
            capitals.put("AU", "Canberra");
        }
        System.err.println("Known capital cities: " + capitals.size());
        System.err.println("Capital city of GB: " + capitals.get("GB"));
        hzc.shutdown();
    }
}
```

By running our client multiple times, we see that the first run initializes the capitals map with our starting set of data, before shutting down the client, which will allow the JVM instance to complete and exit cleanly. However, when we run the client again, the data is successfully persisted by the still running cluster so that we won't have to repopulate it a second time. Initially, our client connects to one of the configuration-defined server nodes; however, in doing so, it learns about the existence of the other nodes once it is running and connected. Therefore, if our supporting member node dies, the client will simply connect to another one of the other nodes and continue as normal.
The only critical phase is the initial connection, but unlike the member nodes, we don't have an auto-discovery mechanism in place; therefore, the location of the cluster needs to be configured explicitly. If the node that we have listed is down at the time of our client starting up, we will fail to connect to the cluster irrespective of the state of the other nodes or the cluster as a whole. We can address this by listing a number of seed nodes within our client's configuration; as long as one of these nodes is available, we can connect to the cluster and go from there.

```java
ClientConfig conf = new ClientConfig();
conf.getNetworkConfig().addAddress("127.0.0.1:5701");
conf.getNetworkConfig()
    .addAddress("127.0.0.1:5702", "127.0.0.1:5703");
```

By default, the ordering of the nodes that we attempt to connect to is shuffled. Therefore, if the first node that we try to connect to is down or is having networking issues, we will have to wait until the configured connection timeout is reached (3 seconds by default) before moving on to the next one to try. However, because there is a random ordering to our attempts, we will achieve a faster connection time if there is a partial failure, at least for a certain proportion of time; this is preferable to a possible consistently delayed connection issue.

## Serialization and classes

One issue that we do introduce when using the client driver is that while our cluster can hold, persist, and serve classes, it doesn't have to and might not actually hold the POJO class itself in its classpath; rather, a serialization of the persisted object. This means that as long as each of our clients holds the appropriate class in its classpath, we can successfully serialize (for persistence) and de-serialize (for retrieval) the objects, but our cluster nodes can't. You can most notably see this when we try to retrieve entries via the TestApp console for custom objects stored to the cluster by a client, as this will produce `ClassNotFoundException`.

The process used to serialize objects to the cluster starts by checking whether the object is a well-known primitive-like class (String, Long, Integer, byte[], ByteBuffer, and Date); if so, these are serialized directly. If not, Hazelcast next checks whether the object implements com.hazelcast.nio.DataSerializable, and if so, it uses the appropriate methods provided to marshal the object. Otherwise, it falls back to the standard Java serialization.
However, when we use the distributed executor, MapReduce, or entry processor, these classes must be present on the classpath of each cluster node, as these operations will actually be performed on the cluster nodes themselves. In addition, we can also use a relatively new configuration option that allows us to store map objects in a native object format. By setting `in-memory-format` of a map to `OBJECT`, we can avoid the need to serialize/de-serialize the object when interacting with this map. This has the greatest benefit when we are mostly processing and querying this map, and as you might expect, this configuration also requires that the object's class be available to the server's classpath.

**Getting straight to the point**

One historical issue with the client method of connecting to the cluster is that most operations require multiple hops to perform an action. This is because we only maintain a connection to a single node of the cluster and run all our operations through it. With more recent versions of Hazelcast, this issue has been addressed; clients have been made smarter. As the strategy of data partitioning is relatively simple and operates on a consistent hashing algorithm, this knowledge and awareness has been added to the clients, allowing them to route cluster operations directly to the node that owns the data being interacted with. For operations that are not key based and hence, don't have a nominal owner, `LoadBalancer` within the client will attempt to spread the load between the available member nodes; by default, this uses a round-robin strategy but is a configurable option within `ClientConfig` used to create the client.

However, these routing smarts might not work under all circumstances; if we have an overly strict firewall that does not allow our client to connect to all the nodes in a cluster, we can disable this client capability. This means that the client will only ever connect to nodes that it is specifically told about and will connect to one of them to act as a coordinating proxy for any operations that the client wishes to perform to interact with a wider cluster.

```java
conf.getNetworkConfig().setSmartRouting(false);
```
Typical Deployments

Architectural overview
As we have seen, there are a few different types of deployment we could use; which one you choose really depends on your application's make-up. Each has a number of trade-offs, but most deployments tend to use one of the first two, with the client and server cluster approach the usual favorite unless we have a mostly compute-focused application where the former is a simpler setup.

So, let's have a look at the various architectural setups that we can employs and what situations they are best suited to.

Peer-to-peer clusters
This is the standard example that we have been mostly using until now: each node houses both our application and the data persistence and processing. It is most useful when we have an application that is primarily focused towards asynchronous or high-performance computing and executes a lot of tasks on the cluster. The greatest drawback is the inability to scale our application and data capacity separately.
Smart clients and server clusters

This is a more appropriate setup for a situation wherein we mostly store data in our cluster rather than running tasks. A cluster of server nodes is independently created, scaled, and managed. It is then interacted with via a smart client driver from our application. While this provides good separation between our application and the Hazelcast cluster, it does require more awareness of the classpaths of both our application and the cluster nodes.
Typical Deployments

**Dummy client proxying through a single node**

This is an emergency option that gives our application more direct networking control over where operations actually go, at least from a networking perspective. Relying on a cluster node to helpfully proxy our activities as required allows us to leapfrog through a single node in order to get to the others in the cluster. The only real use case for this strategy is where the smart client doesn't work given a very particularly restrictive network topology. One additional negative side effect of this option is that a proportion of server resources are consumed for handling client operations. If a node has to handle more than its fair share by acting as a coordinator and we haven’t balanced our application instances across multiple nodes, we might find an imbalanced saturation of resources.

![Diagram of cluster nodes and applications]

**Summary**

We have seen that we have a number of strategies at our disposal for deploying Hazelcast within our architecture, such as treating it like a clustered standalone product akin to a traditional data source but with more resilience and scalability. For more complex applications, we can directly absorb the capabilities into our application, but that does come with some strings attached. However, whichever approach we choose for our particular use case, we have easy access to scaling and control at our finger tips.

In the next chapter, we will look beyond just Hazelcast and the alternative methods of getting access to the data stored in the cluster.
Where to buy this book

You can buy Getting Started with Hazelcast Second Edition from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.