The Hazelcast press release goes something like this: "Hazelcast ([www.hazelcast.com](http://www.captaindebug.com/2013/10/www.hazelcast.com)) is reinventing in-memory data grid through open source. Hazelcast provides a drop-in library that any Java developer can include in minutes to enable them to build elegantly simple mission-critical, transactional, and terascale in-memory applications".

### So, what does that really mean?

Okay, so that's just marketing/PR bumpf. What is Hazelcast… in real life? The answer can be succinctly given using code. Imaging you're writing an application and you need a Map<'String,String> and when you're in production you'll have multiple instances of your app in a cluster. Then writing the following code:

HazelcastInstance instance = Hazelcast.newHazelcastInstance();

loggedOnUsers = instance.getMap("Users");

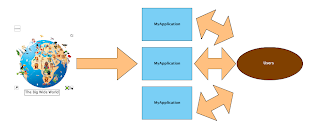
…means that data added to your map by one instance of your application is available to all the other instances of your application2  
  
There are a few points that you can deduce from this. Firstly, Hazelcast nodes are 'masterless', which means that it isn't a client-server system. There is a cluster leader, which is by default the oldest member of the cluster, which manages how data is spread across the system; however, if that node went down, then next oldest will take over.  
  
Having a bunch of distributed Maps, Lists, Queues etc, means that everything is held in memory. If one node in your cluster dies, then you're okay, there's no loss of data; however, if a number of nodes die at the same time, then you're in trouble and you'll get data loss as the system won't have time to rebalance itself. It also goes without saying that if the whole cluster dies, then you're in big trouble.

### So, why is Hazelcast a good bet?

1. It's open source. This is [usually a good thing…](http://www.captaindebug.com/2011/09/benefits-and-dangers-of-using.html" \l ".UmFlj2Tk8jc)
2. Hazelcast have just received a large cash injection to 'commoditize' the product. For more on this take a look [here](http://www.prnewswire.com/news-releases/hazelcast-announces-25-million-in-series-a-funding-to-commoditize-in-memory-computing-224634081.html) and [here](http://www.infoq.com/news/2013/09/hazelcast-vc-funding).
3. Rod Johnson, yes Mr Spring, is now on the board of Hazelcast.
4. It just works1.
5. Getting started is pretty easy.

### The Scenario

To demonstrate Hazelcast imagine that you're writing an application, in this case modelled by the MyApplication class and then there's a big, wide world of users as modelled by the BigWideWorld class. As expected, users from the BigWideWorld log in and out of your application. Your application is very popular and you're running multiple instances of it in a cluster, so when a user logs in an instance of the app it stores their details (as modelled by the User class) in a Map and the contents of the map are synchronised with the maps held by other instances of your application.



### POM Configuration

The first thing to do is to setup the POM.xml and there's only one entry to consider:

<dependency>

<groupId>com.hazelcast</groupId>

<artifactId>hazelcast</artifactId>

<version>3.1</version>

</dependency>

### The Code

The BigWideWorld is the starting point for the code and it's a very small class for such a large concept. It has one method,nextUser(), which randomly chooses the name of the next user to log in or out from a collection of all your application's users.

public class BigWideWorld {

private static Random rand = new Random(System.currentTimeMillis());

private final Users users = new Users();

private final int totalNumUsers = users.size();

public String nextUser() {

User user = users.get(rand.nextInt(totalNumUsers));

String name = user.getUsername();

return name;

}

}

The collection of users is managed by the Users class. This is a sample code convenience class that contains a number of hard coded users' details.

public class Users {

/\*\* The users in the database \*/private final User[] users = { new User("fred123", "Fred", "Jones", "fredj@a.com"),new User("jim", "Jim", "Jones", "jimj@a.com"),new User("bill", "Bill", "Jones", "bill@a.com"),new User("ted111", "Edward", "Jones", "tedj@a.com"),new User("annie", "Annette", "Jones", "annj@a.com"),new User("lucy", "Lucy", "Jones", "lucyj@a.com"),new User("jimj", "James", "Jones", "jimj@a.com"),new User("jez", "Jerry", "Jones", "fredj@a.com"),new User("will", "William", "Jones", "willj@a.com"),new User("shaz", "Sharon", "Jones", "shazj@a.com"),new User("paula", "Paula", "Jones", "pauj@a.com"),new User("leo", "Leonardo", "Jones", "leoj@a.com"), };

private final Map<String, User> userMap;

public Users() {

userMap = new HashMap<String, User>();

for (User user : users) {userMap.put(user.getUsername(), user);}

}

/\*\*

\* The number of users in the database

\*/public int size() {return userMap.size();}

/\*\*

\* Given a number, return the user

\*/public User get(int index) {return users[index];}

/\*\*

\* Given the user's name return the User details

\*/public User get(String username) {return userMap.get(username);}

/\*\*

\* Return the user names.

\*/public Set<String> getUserNames() {return userMap.keySet();}

}

This class contains a few database type of calls, such as get(String username) to return the user object for a given name, orget(int index) to return a given user from the DB, or size() to return the number of users in the database.  
  
The user is described by the User class; a simple Java bean:

public class User implements Serializable {

private static final long serialVersionUID = 1L;private final String username;private final String firstName;private final String lastName;private final String email;

public User(String username, String firstName, String lastName, String email) {super();this.username = username;this.firstName = firstName;this.lastName = lastName;this.email = email;}

public String getUsername() {return username;}

public String getFirstName() {return firstName;}

public String getLastName() {return lastName;}

public String getEmail() {return email;}

@Overridepublic String toString() {

StringBuilder sb = new StringBuilder("User: ");

sb.append(username);

sb.append(" ");

sb.append(firstName);

sb.append(" ");

sb.append(lastName);

sb.append(" ");

sb.append(email);

return sb.toString();}

}

Moving on the crux of the blog, which is the MyApplication class. Most of the code in this blogs is merely window dressing, the code that's of importance is in MyApplication's constructor. The construct contains two lines of code; the first gets hold of a new Hazelcast instance, whilst the second uses that instance to create a Map<String, User> with a namespace of "Users". This is all the Hazelcast specific code that's needed. The other methods: logon(), logout() and isLoggedOn() just manage the users.

public class MyApplication {

private final Map<String, User> loggedOnUsers;

private final Users userDB = new Users();

private final SimpleDateFormat sdf = new SimpleDateFormat("kk:mm:ss-SS");

private long lastChange;

public MyApplication() {

HazelcastInstance instance = Hazelcast.newHazelcastInstance();

loggedOnUsers = instance.getMap("Users");

}

/\*\*

\* A user logs on to the application

\*

\* @param username

\* The user name

\*/

public void logon(String username) {

User user = userDB.get(username);

loggedOnUsers.put(username, user);

lastChange = System.currentTimeMillis();

}

/\*\*

\* The user logs out (or off depending on your pov).

\*/

public void logout(String username) {

loggedOnUsers.remove(username);

lastChange = System.currentTimeMillis();

}

/\*\*

\* @return Return true if the user is logged on

\*/

public boolean isLoggedOn(String username) {

return loggedOnUsers.containsKey(username);

}

/\*\*

\* Return a list of the currently logged on users - perhaps to sys admin.

\*/

public Collection<User> loggedOnUsers() {

return loggedOnUsers.values();

}

/\*\*

\* Display the logged on users

\*/

public void displayUsers() {

StringBuilder sb = new StringBuilder("Logged on users:\n");

Collection<User> users = loggedOnUsers.values();

for (User user : users) {

sb.append(user);

sb.append("\n");

}

sb.append(loggedOnUsers.size());

sb.append(" -- ");

sb.append(sdf.format(new Date(lastChange)));

sb.append("\n");

System.out.println(sb.toString());

}

}

All the above is tied together using a simple Mainclass:

public class Main {

public static void main(String[] args) throws InterruptedException {

BigWideWorld theWorld = new BigWideWorld();

MyApplication application = new MyApplication();

while (true) {

String username = theWorld.nextUser();

if (application.isLoggedOn(username)) {application.logout(username);} else {application.logon(username);}

application.displayUsers();

TimeUnit.SECONDS.sleep(2);}

}

}

This code creates an instance of the BigWideWorld and MyApplication. It then infinitely loops grabbing hold of the next random user name. If the user is already logged in, then the user logs out. If the user is not logged in, then the user logs in. The logged in users are then displayed so that you can see what's going on.

### Running the App

After building the app, open a terminal and navigate to the projects target/classes directory. Then type in the following command:

java -cp /your path to the/hazelcast-3.1/lib/hazelcast-1.jar:. com.captaindebug.hazelcast.gettingstarted.Main

When running, you'll get output that looks something like this:

Logged on users:

User: fred123 Fred Jones fredj@a.com

User: jimj James Jones jimj@a.com

User: shaz Sharon Jones shazj@a.com

User: paula Paula Jones pauj@a.com

User: lucy Lucy Jones lucyj@a.com

User: jez Jerry Jones fredj@a.com

User: jim Jim Jones jimj@a.com

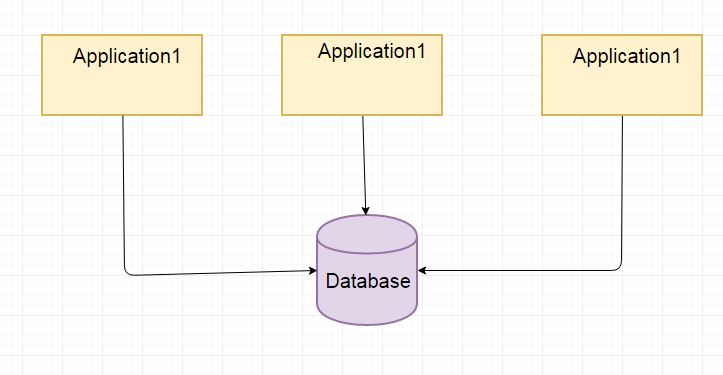
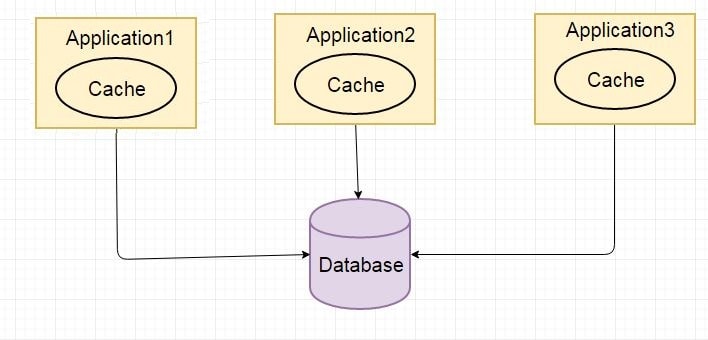
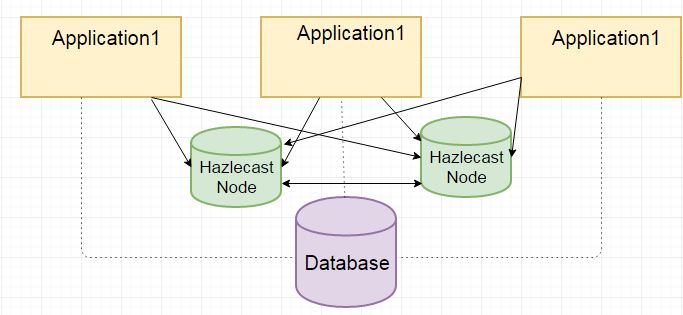
7 -- 14:54:16-17

Next, open more terminals and run a few more instances of your application.

NEW

Hazelcast is a radical, new approach towards data that was designed from the ground up around distribution. It embraces a new, scalable way of thinking in that data should be shared for resilience and performance while allowing us to configure the trade-offs surrounding consistency, as the data requirements dictate. Hazelcast is a distributed, highly available and scalable

Why Hazelcast is required?

* A traditional application architecture consists of various modules interacting with common Database.   
    
  This allows us to easily scale our application by adding more hardware to increase processing capacity. **However as there is only a single database layer this leads to resource saturation leading to performance issues.**
* This issue can be resolved by making use of cache. These are stand-alone single instances.   
    
  **There are cache consistency issues.** If there is a change in database, the cache might not get updated.
* Hazelcast is a radical, new approach towards data that was designed from the ground up around distribution.   
    
  **It embraces a new, scalable way of thinking in that data should be shared for resilience and performance.**

Features of Hazelcast-

* **The data is always stored in-memory (RAM) of the servers**.This makes it incredibly fast.
* **Multiple copies are stored in multiple machines for automatic data recovery in case of single or multiple server failures**. In the event of failure, the overall cluster will not suffer any data loss.
* Major feature of Hazelcast is its masterless nature. Each node is configured to be functionally the same and operates in a peer-to-peer manner.
* The data model is object-oriented and non-relational.
* Servers can be dynamically added or removed to increase the amount of CPU and RAM.
* The data can be persisted from Hazelcast to a relational or NoSQL database.
* A Java Map API accesses the distributed key-value store.