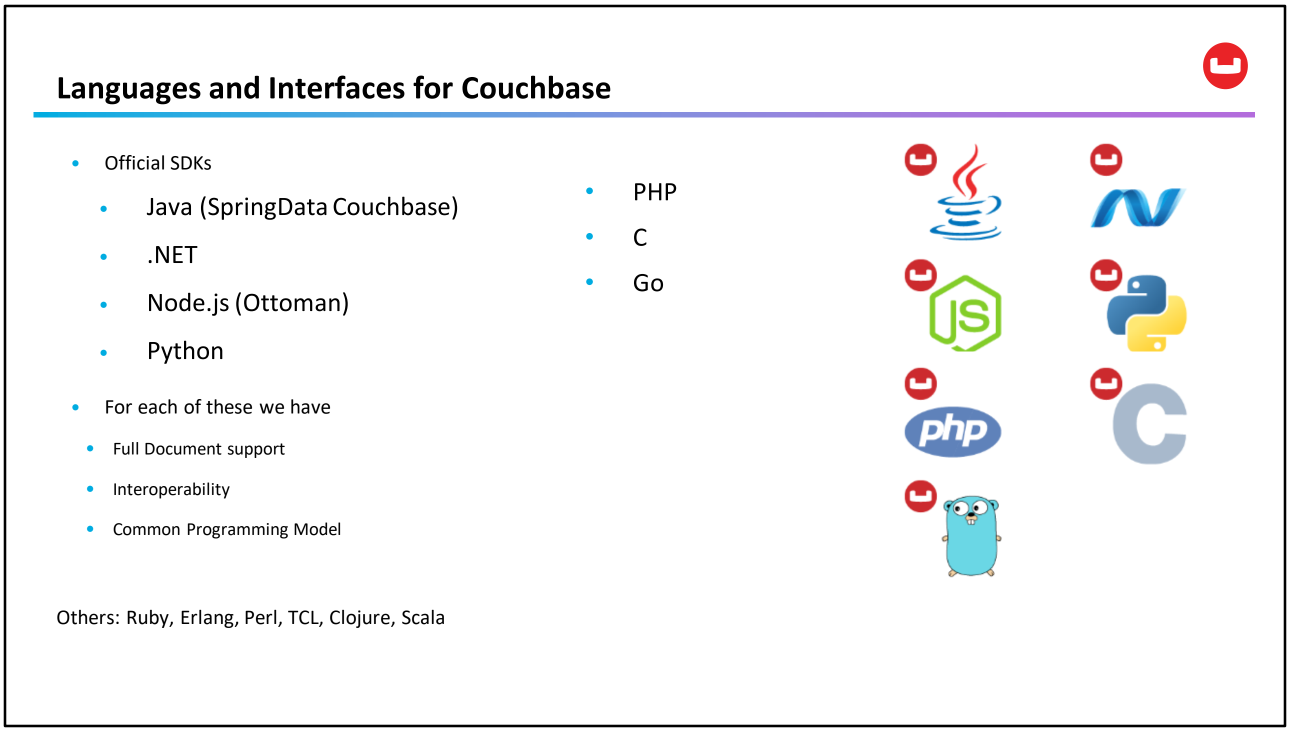
Couchbase SDK for 6.5

Revised: June 22nd, 2020

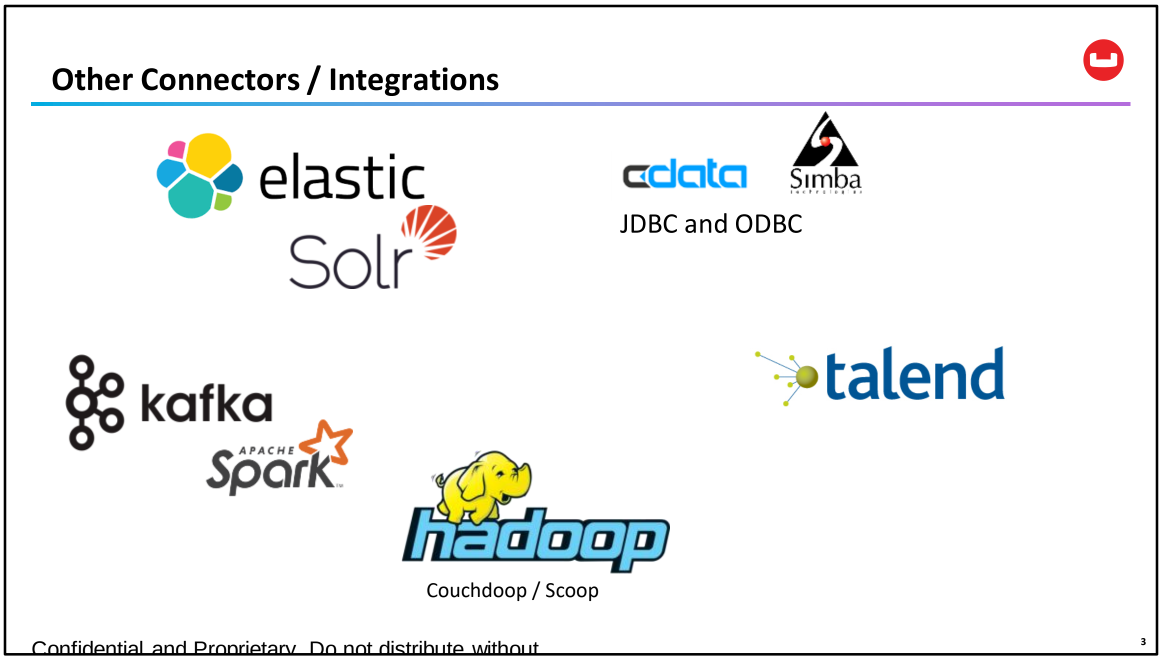


Languages and Interfaces for Couchbase

- Official SDKs
  - Java (SpringData Couchbase)
  - .NET
  - Node.js (Ottoman)
  - Python
- For each of these we have
  - Full Document support
  - Interoperability
  - Common Programming Model

- PHP
- C
- Go

Others: Ruby, Erlang, Perl, TCL, Clojure, Scala



Other Connectors / Integrations

elastic

Solr

cdata Simba

JDBC and ODBC

kafka

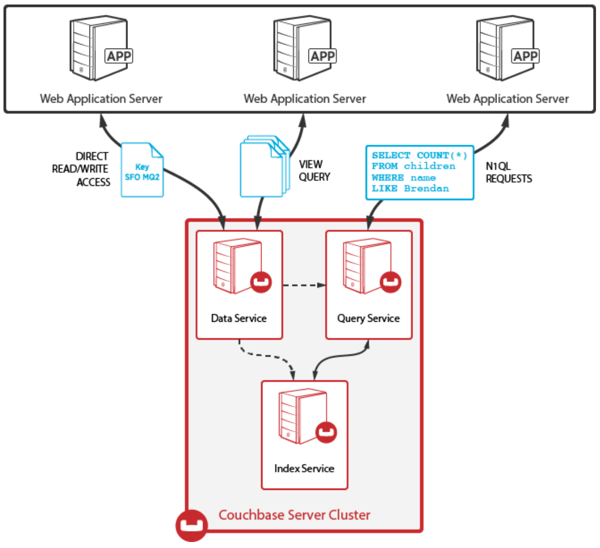
APACHE Spark

talend

hadoop

Couchdoop / Scoop

Application to Database Interaction (Recap)

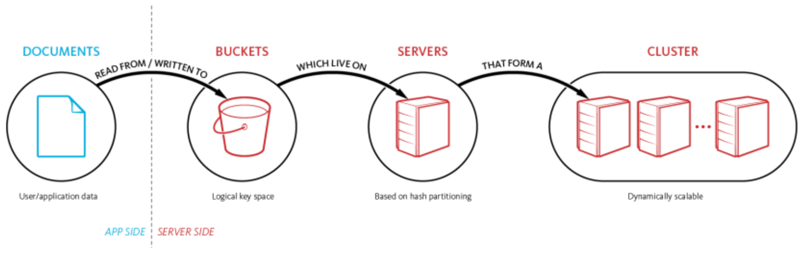


<u>KEY POINT: Applications communicate directly to the services they need to fulfill the application request and the application does not need to be topology aware as the SDK has that already.</u>

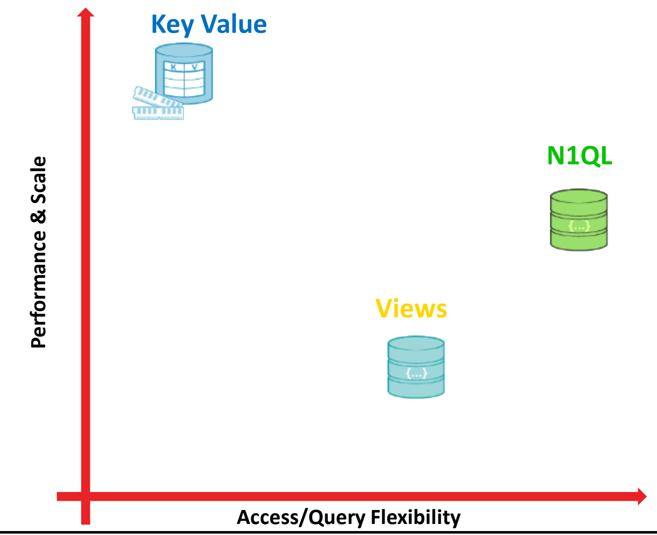
- Single node type, services defined dynamically
- One node acts the same as 100, just the services are spread out in the cluster
- Query service accesses Index and Data to formulate response
- All query and document access is topology aware and dynamically scalable
- Develop with one node, deploy against multiple production nodes
- The Couchbase SDK handles knowing about where in the cluster it needs to go to satisfy whatever the application is requesting, be I CRUD or cluster management.

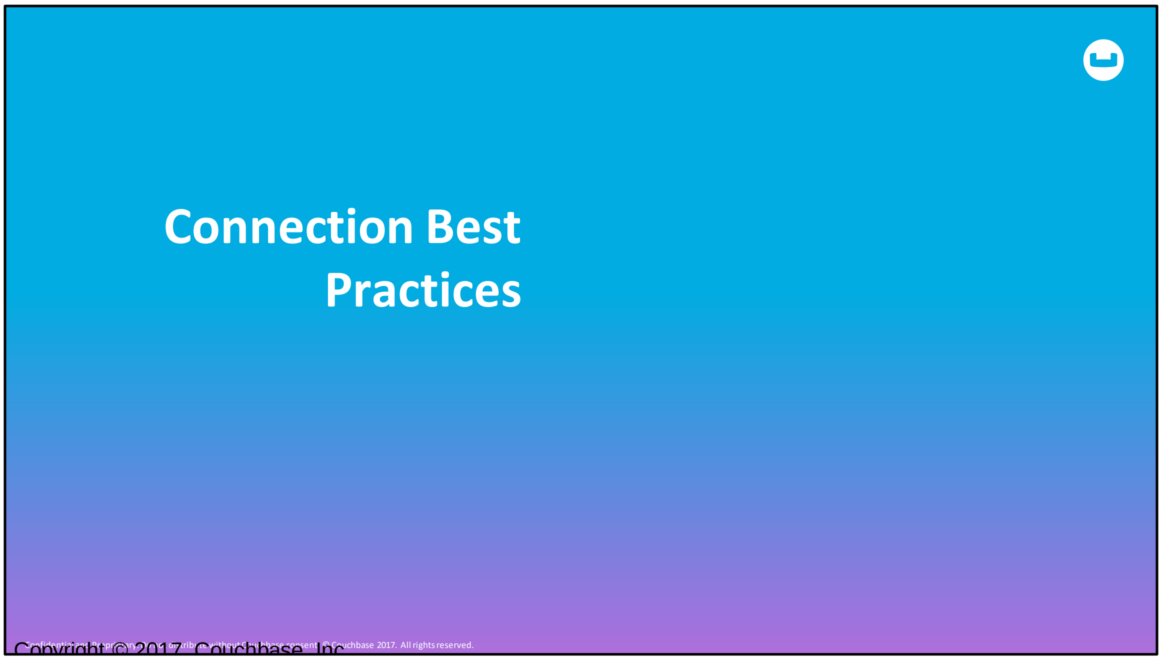
Buckets - When to use more than one ? (RECAP)

- When you need to treat or access the data differently
  - Different High Availability requirements (1,2 or 3 replicas)
  - Different performance / residency needs (how much data to cache)
  - Security / Multi-tenancy
  - Segregating Binary and JSON data – especially with view usage
  - Need to track metrics separately

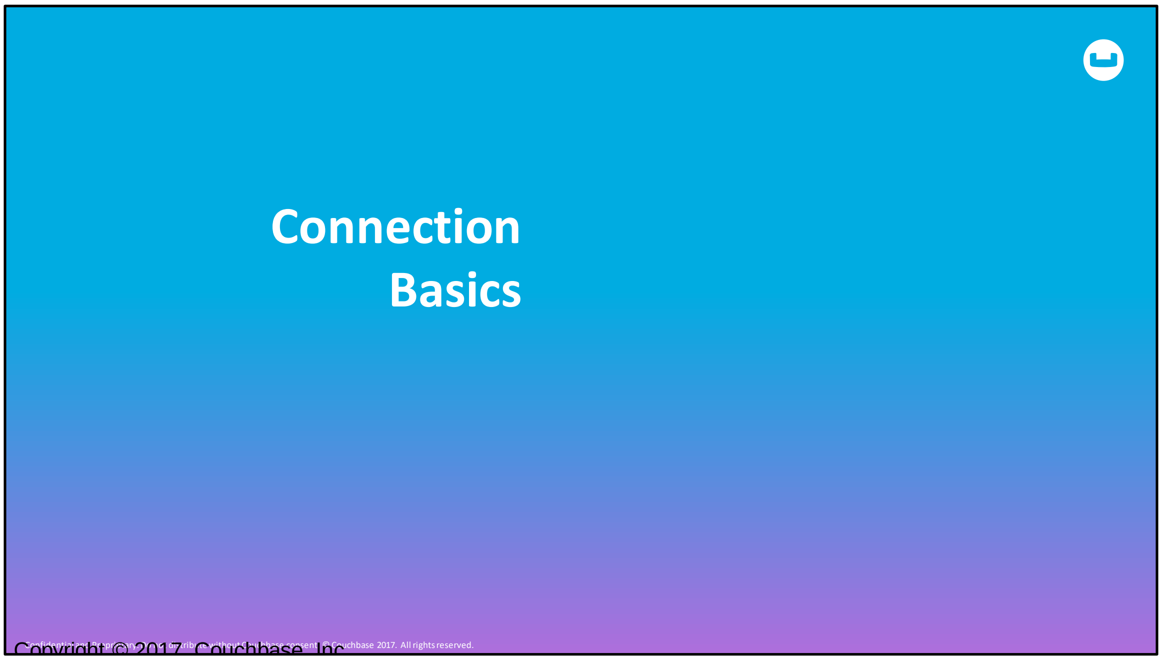


Use the right tool for the job

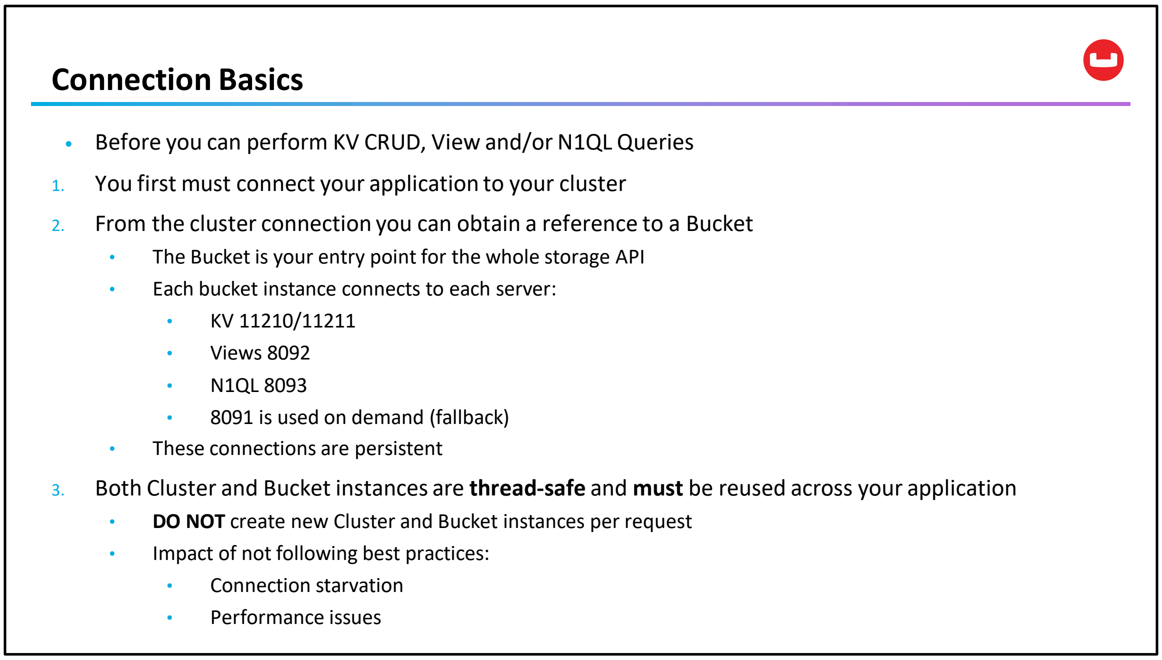




Connection Best Practices



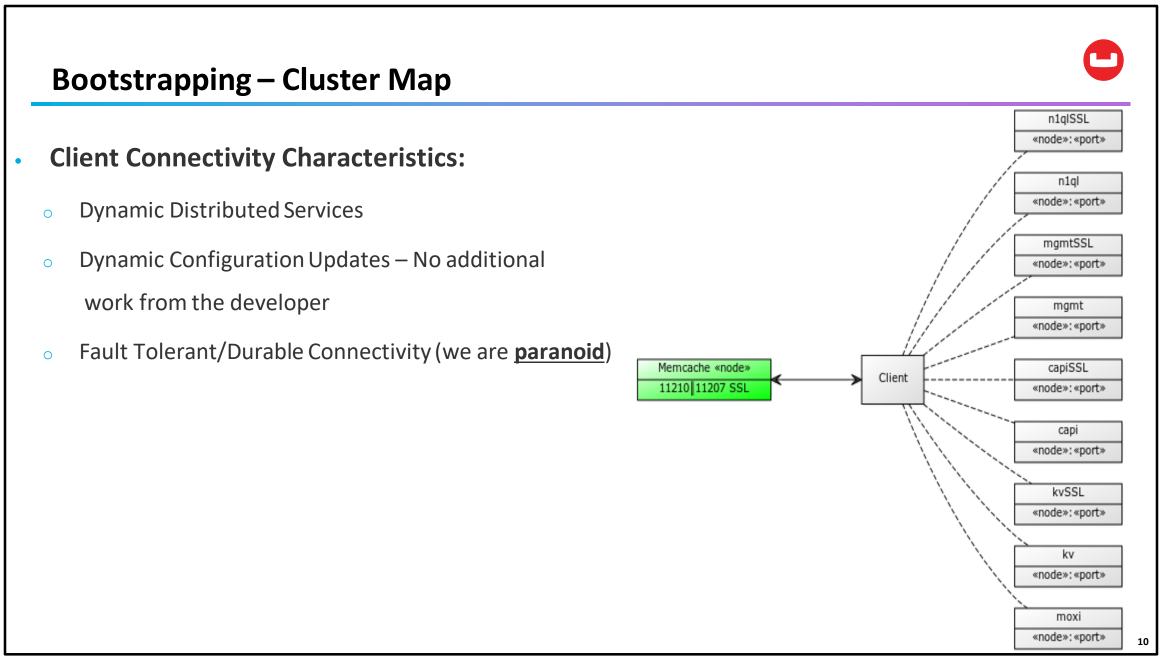
Connection Basics



Connection Basics

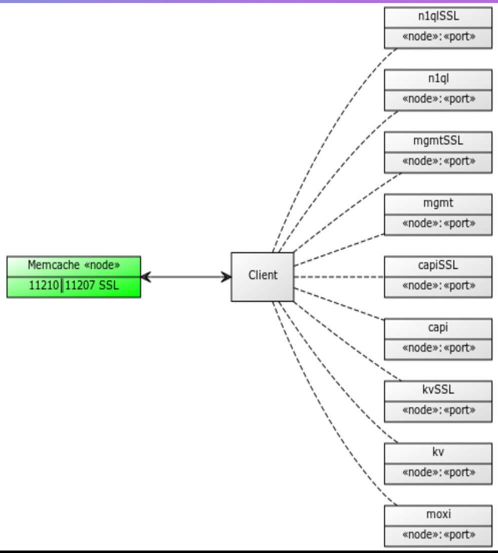
- Before you can perform KV CRUD, View and/or N1QL Queries

1. You first must connect your application to your cluster
2. From the cluster connection you can obtain a reference to a Bucket
    - The Bucket is your entry point for the whole storage API
    - Each bucket instance connects to each server:
        - KV 11210/11211
        - Views 8092
        - N1QL 8093
        - 8091 is used on demand (fallback)
    - These connections are persistent
3. Both Cluster and Bucket instances are **thread-safe** and **must** be reused across your application
    - **DO NOT** create new Cluster and Bucket instances per request
    - Impact of not following best practices:
        - Connection starvation
        - Performance issues



Bootstrapping – Cluster Map

- **Client Connectivity Characteristics:**
  - Dynamic Distributed Services
  - Dynamic Configuration Updates – No additional work from the developer
  - Fault Tolerant/Durable Connectivity (we are **paranoid**)



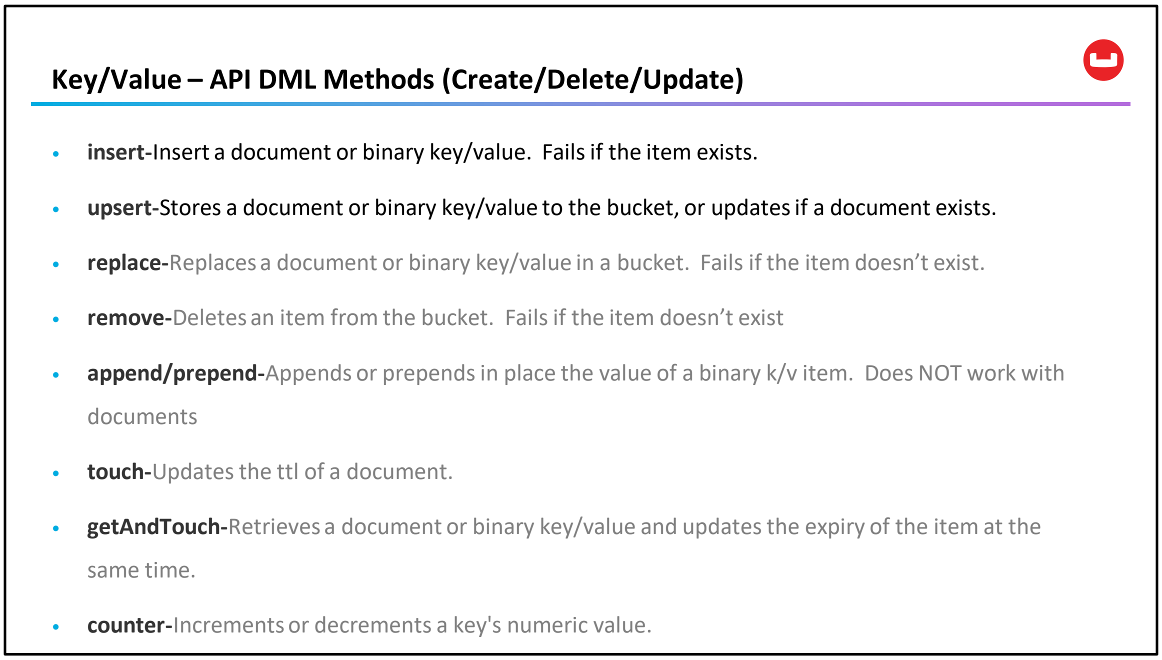
10

Connection Bootstrap Process

1. Connect to the cluster using the Cluster class
   - Default connection is made on 127.0.0.1
   - To connect to your cluster you need to specify at least one Couchbase endpoint
   - Endpoints can be IP addresses or FQDNs
   - **Best Practice:** Use FQDNs
   - **Best Practice:** Supply at least 3 Couchbase endpoints as a seed
2. Using the cluster instance connect to a bucket
   - By default, the bucket connection will connect to the “default bucket” (on 11211)
   - Must supply the **bucket name** and optionally the **bucket password**
   - The bucket will connect to one of the Couchbase servers in the seed list (1st one in the list)
3. The Couchbase server will return a **CLUSTER MAP**
   - Using the **CLUSTER MAP**, the bucket instance will create 1 to 3 connections per Couchbase server.

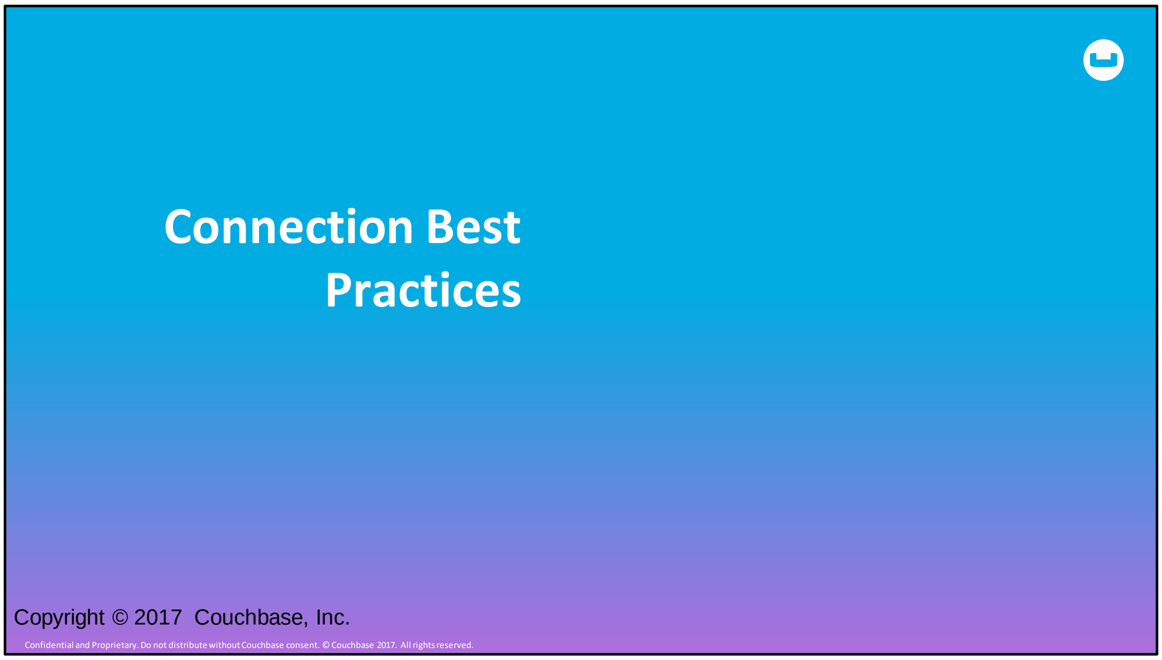
Key/Value – API DML Methods (Retrieval)

- **get**-Retrieves a document or binary key/value.
- **getAndLock-**Lock the document or binary key/value on the server and retrieve it. When an document is locked, its CAS changes and subsequent operations on the document (without providing the current CAS) will fail until the lock is no longer held.
- **getReplica-**Get a document binary key/value from a replica server in your cluster.
- **unlock-**Unlock a previously locked document or binary key/value on within a bucket.

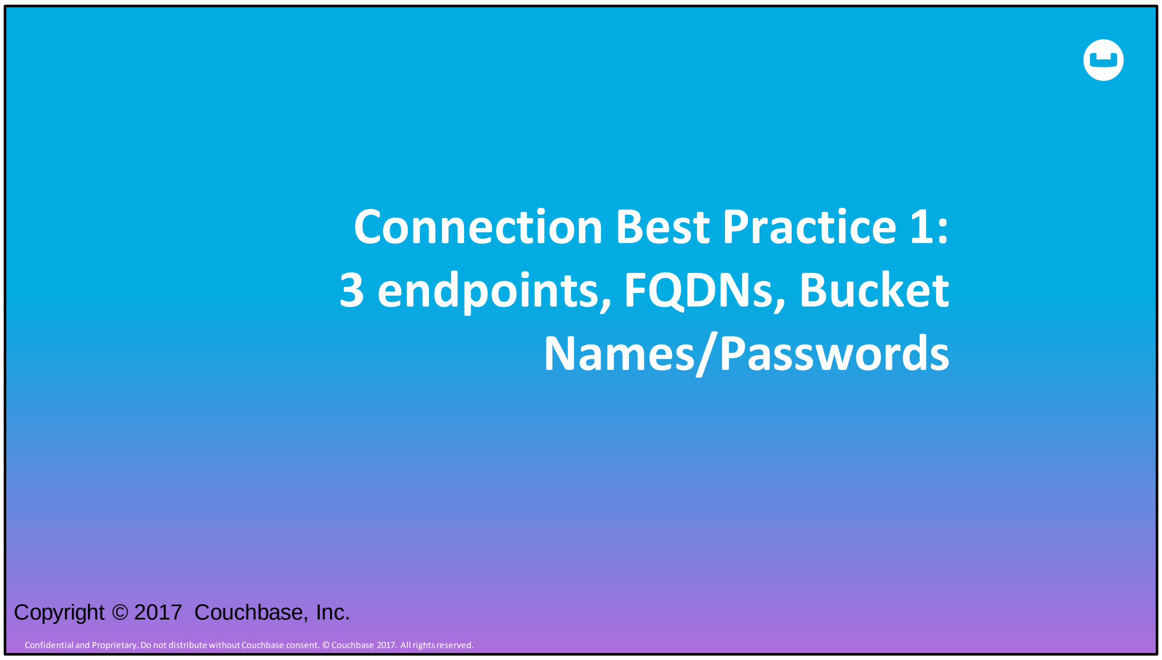


Key/Value – API DML Methods (Create/Delete/Update)

- **insert**-Insert a document or binary key/value. Fails if the item exists.
- **upsert**-Stores a document or binary key/value to the bucket, or updates if a document exists.
- **replace-**Replaces a document or binary key/value in a bucket. Fails if the item doesn't exist.
- **remove-**Deletes an item from the bucket. Fails if the item doesn't exist
- **append/prepend-**Appends or prepends in place the value of a binary k/v item. Does NOT work with documents
- **touch-**Updates the ttl of a document.
- **getAndTouch-**Retrieves a document or binary key/value and updates the expiry of the item at the same time.
- **counter-**Increments or decrements a key's numeric value.



Connection Best Practices

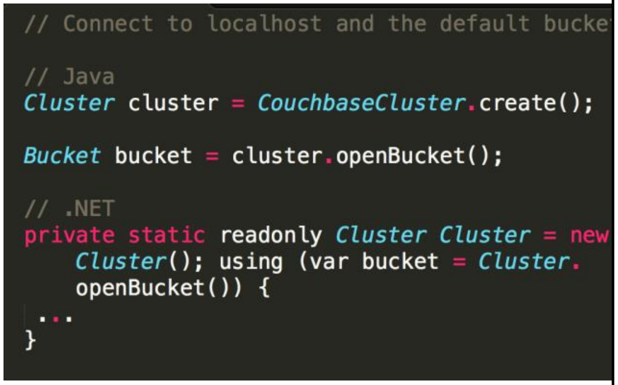


Connection Best Practice 1: 3 endpoints, FQDNs, Bucket Names/Passwords

Connection Basics

- Connecting to a bucket is a 2 step process

1. First, a CouchbaseCluster object (e.g. Java) or Cluster object needs to be initialized (e.g. .NET)
2. Once initialized, the Cluster object can be used to open one or more buckets. This is achieved by calling the openBucket method

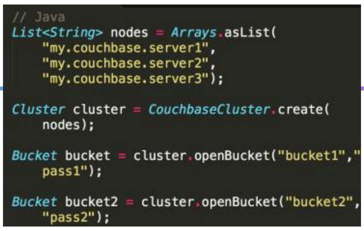


```
// Connect to localhost and the default bucke

// Java
Cluster cluster = CouchbaseCluster.create();

Bucket bucket = cluster.openBucket();

// .NET
private static readonly Cluster Cluster = new
    Cluster(); using (var bucket = Cluster.
    openBucket()) {
 ...
}
```

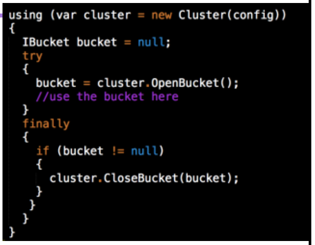
Connection Basics

- **Connecting to a bucket** Best Practices
  - Provide least 3 endpoints
  - Use FQDNs over IP addresses
  - You must know the bucket name and password to connect to a bucket.

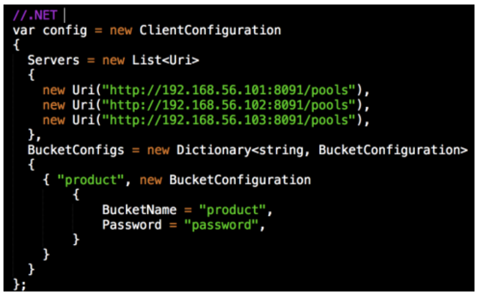
Note: RBAC in Couchbase 5.0 and above requires a user with appropriate access to the bucket



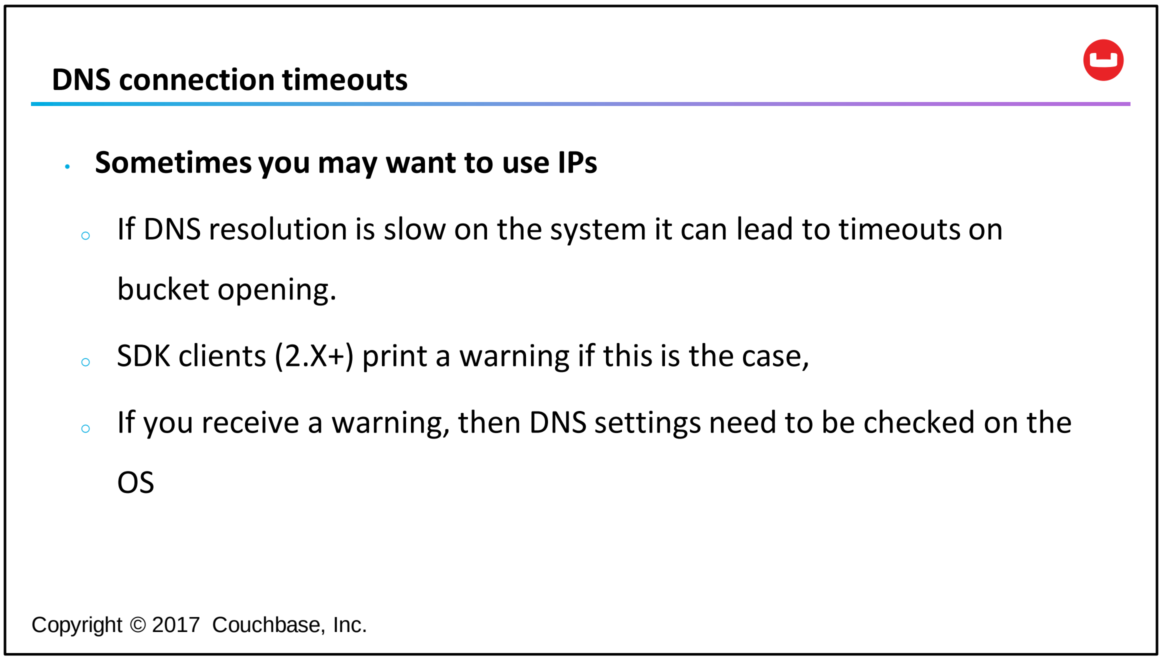
```
// Java
List<String> nodes = Arrays.asList(
    "my.couchbase.server1",
    "my.couchbase.server2",
    "my.couchbase.server3");

Cluster cluster = CouchbaseCluster.create(
    nodes);

Bucket bucket = cluster.openBucket("bucket1","
    pass1");

Bucket bucket2 = cluster.openBucket("bucket2",
    "pass2");
```



```
using (var cluster = new Cluster(config))
{
  IBucket bucket = null;
  try
  {
    bucket = cluster.OpenBucket();
    //use the bucket here
  }
  finally
  {
    if (bucket != null)
    {
      cluster.CloseBucket(bucket);
    }
   }
  }
}
```

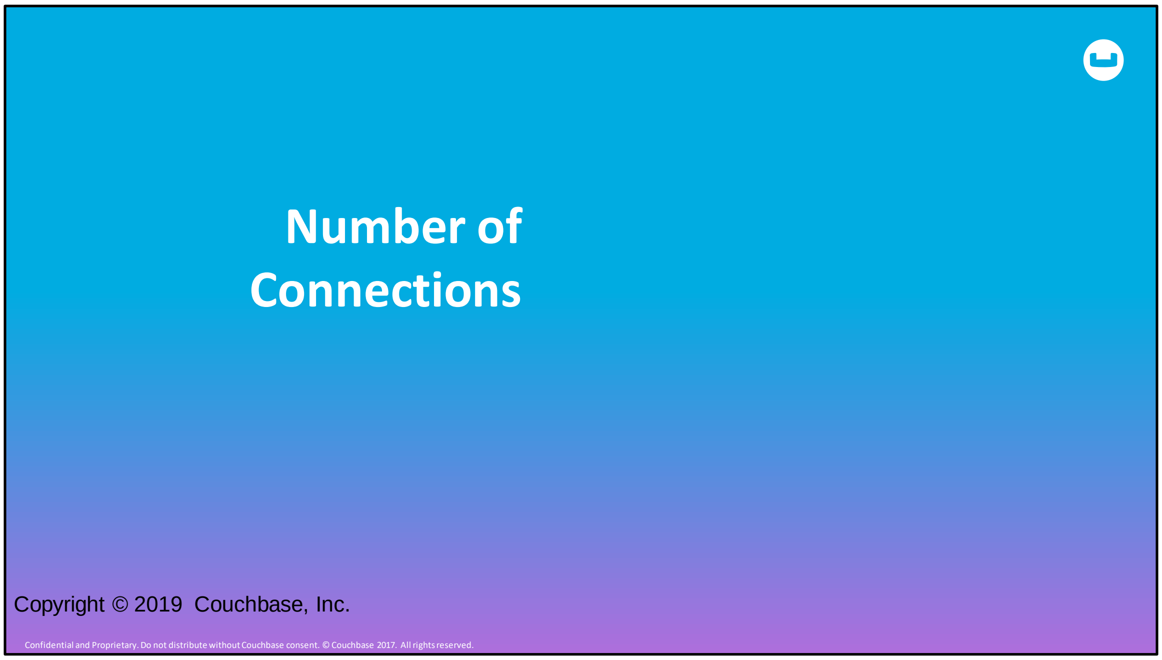


```
//.NET |
var config = new ClientConfiguration
{
  Servers = new List<Uri>
  {
    new Uri("http://192.168.56.101:8091/pools"),
    new Uri("http://192.168.56.102:8091/pools"),
    new Uri("http://192.168.56.103:8091/pools"),
  },
  BucketConfigs = new Dictionary<string, BucketConfiguration>
  {
    { "product", new BucketConfiguration
        {
          BucketName = "product",
          Password = "password",
        }
    }
  }
};
```



DNS connection timeouts

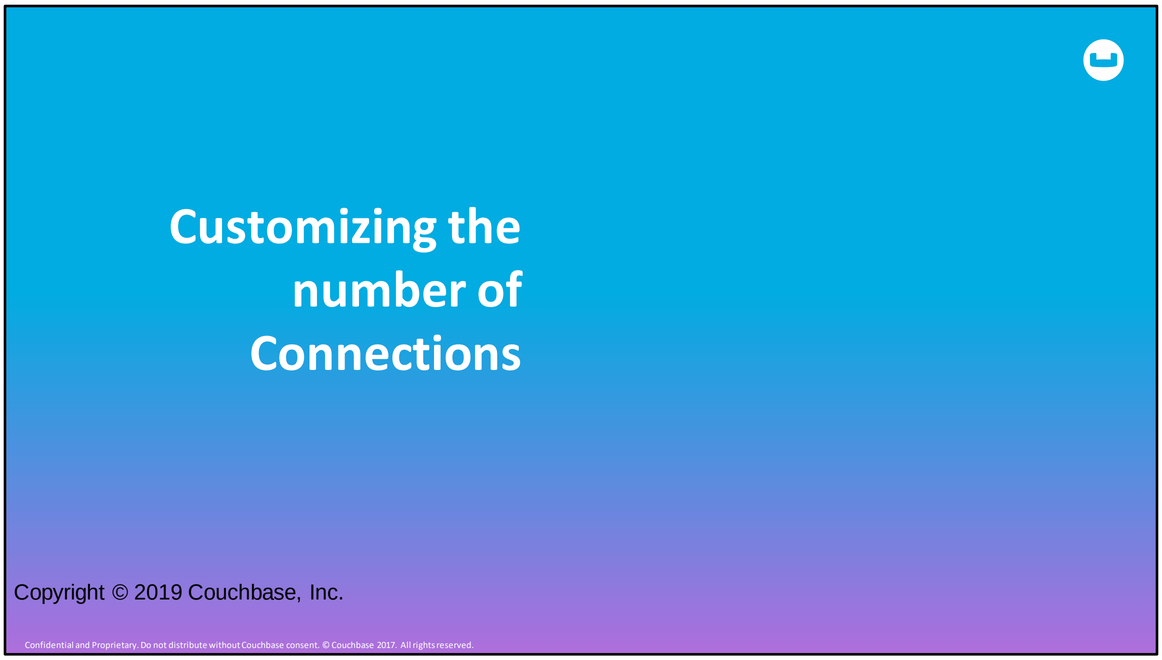
- **Sometimes you may want to use IPs**
  - If DNS resolution is slow on the system it can lead to timeouts on bucket opening.
  - SDK clients (2.X+) print a warning if this is the case,
  - If you receive a warning, then DNS settings need to be checked on the OS



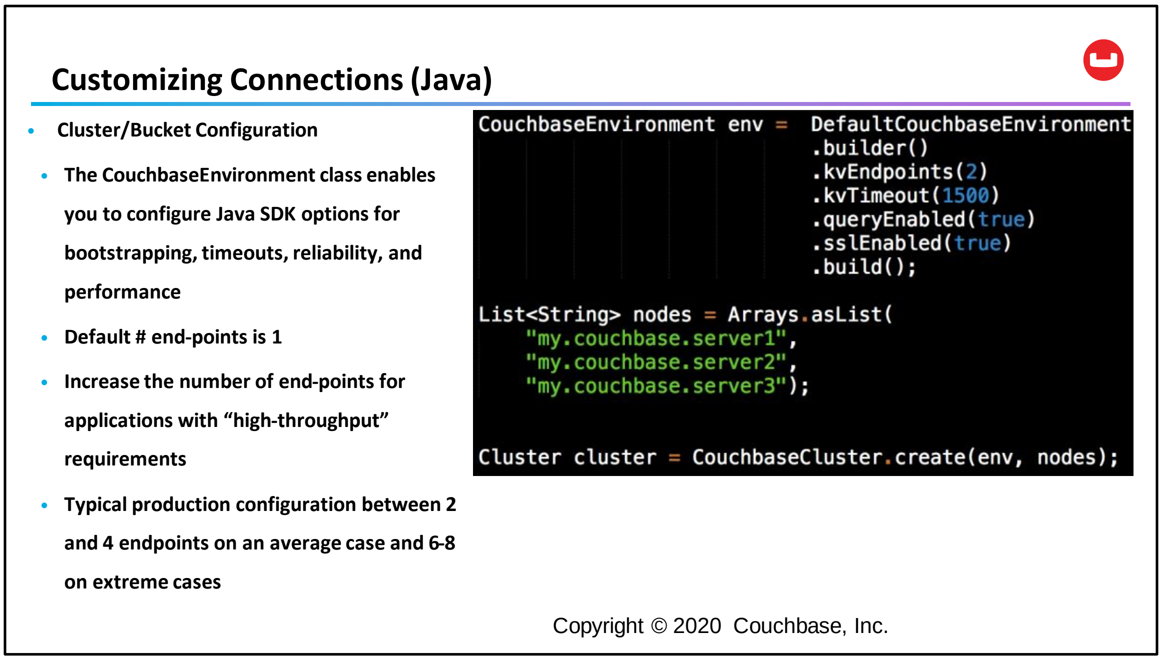
Number of Connections

Number of Connections to Couchbase

- Each connection to Couchbase Server takes up resources (descriptors, memory, management, CPU), so use the # of connections judiciously.
  - There is a hard limit on # of connections that Couchbase will allow per server
  - As of 3.X the limit is 30K per server (Doesn't mean you need to take it to the limit 😜)
- By default each bucket instance creates:
  - 1 connection for KV operations (11210 for all buckets, 11211 for 'default') per server
  - 1 connection for View queries (8092) per server
  - 1 connection for N1QL queries (8093) per server
  - These are persistent connections
- Example
  - Each web application connects to 3 buckets
  - You perform KV, Views and N1QL queries
  - 10 web servers
  - 10 Couchbase Servers
  - Each web server creates (3 buckets * 3 connections * 10 Couchbase servers) = 90 connections
  - Each Couchbase server handles (3 buckets * 3 connections * 10 web servers) = 90 connections
  - The cluster as a whole is handling 90 * 10 Couchbase servers = 900 connections



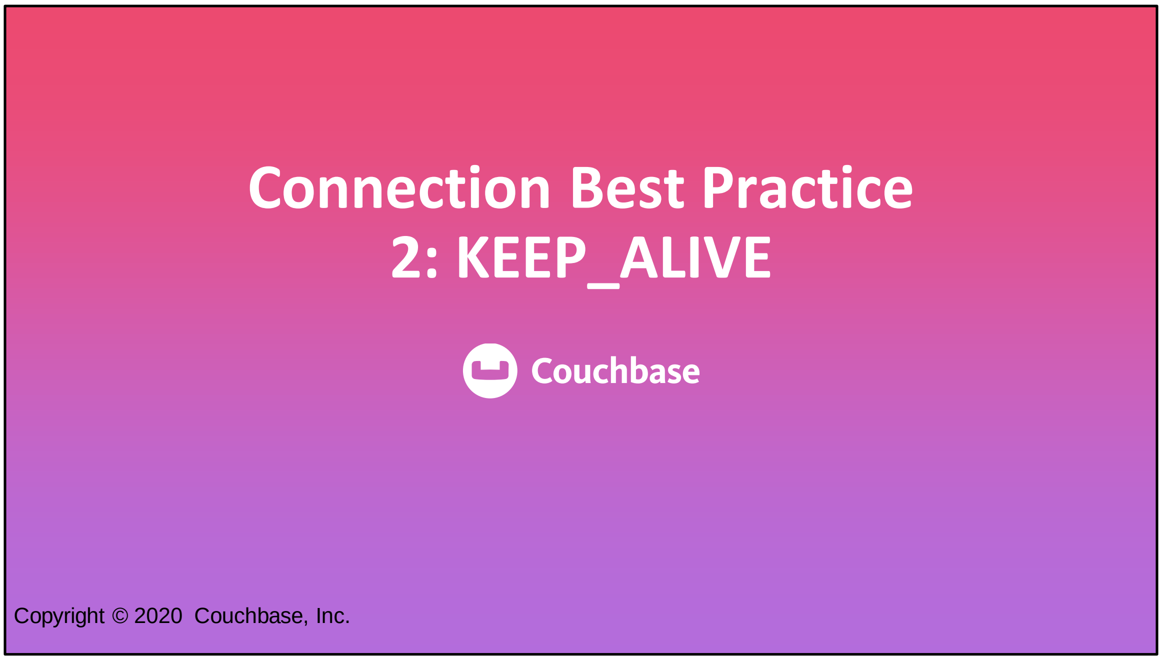
Customizing the number of Connections



Customizing Connections (Java)

- Cluster/Bucket Configuration
  - The CouchbaseEnvironment class enables you to configure Java SDK options for bootstrapping, timeouts, reliability, and performance
  - Default # end-points is 1
  - Increase the number of end-points for applications with “high-throughput” requirements
  - Typical production configuration between 2 and 4 endpoints on an average case and 6-8 on extreme cases

```
CouchbaseEnvironment env =  DefaultCouchbaseEnvironment
                            .builder()
                            .kvEndpoints(2)
                            .kvTimeout(1500)
                            .queryEnabled(true)
                            .sslEnabled(true)
                            .build();

List<String> nodes = Arrays.asList(
    "my.couchbase.server1",
    "my.couchbase.server2",
    "my.couchbase.server3");

Cluster cluster = CouchbaseCluster.create(env, nodes);
```



Connection Best Practice 2: KEEP_ALIVE

Couchbase

Persistent Connections

- Connections are persistent across an application lifecycle
- Firewalls sometimes kill these connections

  When there is no load going through the client to a specific socket,

  there is a chance that a firewall (or something else) is cutting off the connection because it thinks it is stale.

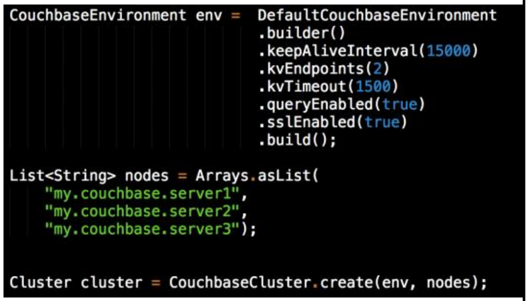
  In order to prevent this, both the JAVA and .NET SDKs can send heartbeat messages.

  The Java SDK (version 2.1+) sends a heartbeat message every 30 seconds over idle sockets. These messages are not sent if regular traffic is flowing in this interval.

  .NET (version 2.1+) sends a heartbeats on Window’s defaults. After 2 hours of idleness, heartbeats are sent every 1 second. Again these messages are not sent if regular traffic is flowing in this interval.

Dealing with idle connections (Java)

- By default a heartbeat every 30 seconds
  - Customize using the **CouchbaseEnvironment** class
  - This is an immutable class for configuring bootstrapping options
  - Uses an embedded Builder class
  - Customize the heart-beat using the **keepAliveInterval(long)**
    - In this example we have customized the heartbeat to 15 seconds

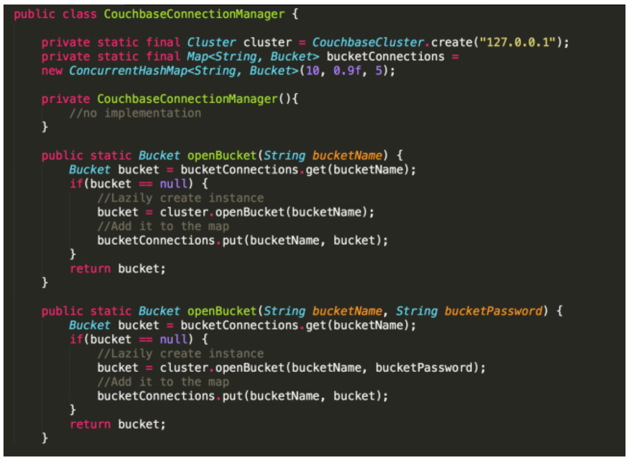


```
CouchbaseEnvironment env =  DefaultCouchbaseEnvironment
                            .builder()
                            .keepAliveInterval(15000)
                            .kvEndpoints(2)
                            .kvTimeout(1500)
                            .queryEnabled(true)
                            .sslEnabled(true)
                            .build();

List<String> nodes = Arrays.asList(
    "my.couchbase.server1",
    "my.couchbase.server2",
    "my.couchbase.server3");


Cluster cluster = CouchbaseCluster.create(env, nodes);
```

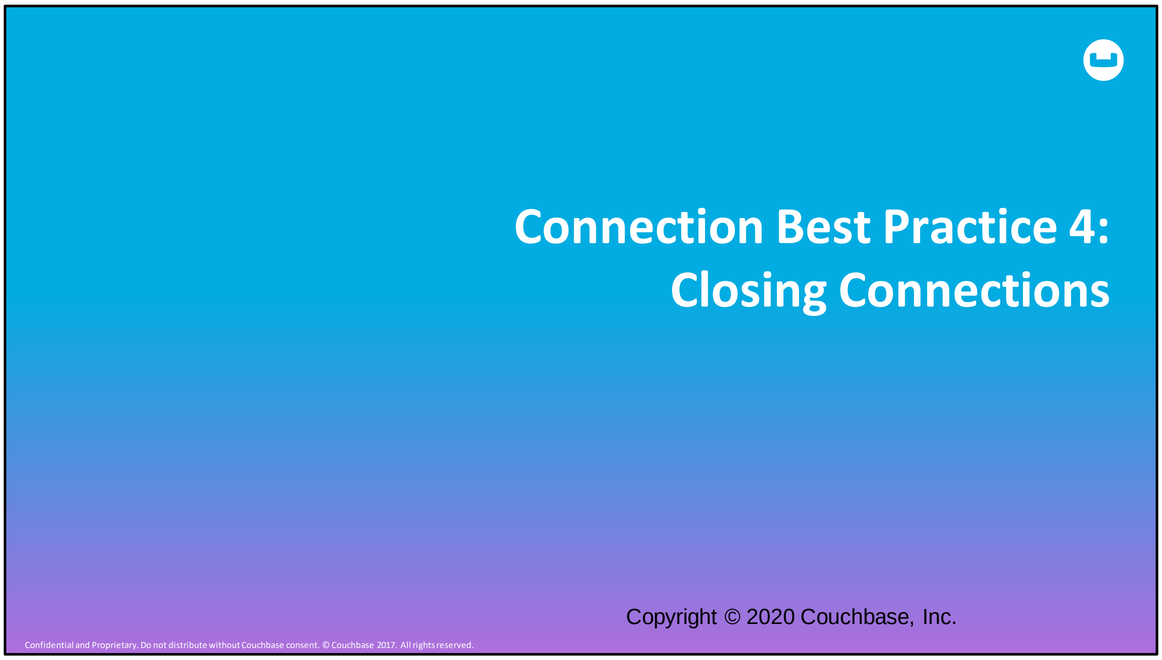
Connection Best Practice 3: Reuse connections via Singleton/Multiton

Bucket Connections should be re-used

- **Re-use a bucket connection for all KV ops & N1QL/View queries for the entire app lifetime!**
    - Use a Singleton Pattern
    - Or multiton Pattern (where you need to connect to multiple buckets
    - Extend multiton if you need to connect to multiple clusters



```java
public class CouchbaseConnectionManager {

    private static final Cluster cluster = CouchbaseCluster.create("127.0.0.1");
    private static final Map<String, Bucket> bucketConnections =
    new ConcurrentHashMap<String, Bucket>(10, 0.9f, 5);

    private CouchbaseConnectionManager(){
        //no implementation
    }

    public static Bucket openBucket(String bucketName) {
        Bucket bucket = bucketConnections.get(bucketName);
        if(bucket == null) {
            //Lazily create instance
            bucket = cluster.openBucket(bucketName);
            //Add it to the map
            bucketConnections.put(bucketName, bucket);
        }
        return bucket;
    }

    public static Bucket openBucket(String bucketName, String bucketPassword) {
        Bucket bucket = bucketConnections.get(bucketName);
        if(bucket == null) {
            //Lazily create instance
            bucket = cluster.openBucket(bucketName, bucketPassword);
            //Add it to the map
            bucketConnections.put(bucketName, bucket);
        }
        return bucket;
    }
```



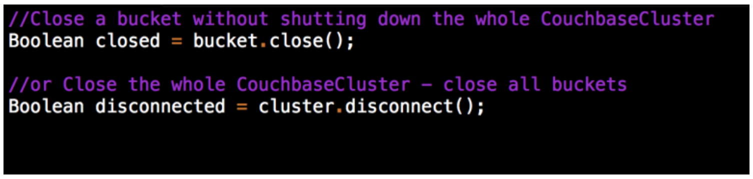
Connection Best Practice 4: Closing Connections

Maximum connection limit

- A Couchbase server has a maximum limit of connections it can keep open (30K per server as of Couchbase 3.X)
- Once a server reaches its maximum then it will ignore new connections
- The more application servers you have connecting to Couchbase, the greater the number of connections a Couchbase server needs to handle
- Very important to therefore close Cluster and Bucket connections.
- If you restart a number of web-servers, then you will have a temporary spike in the number of connections to 11210, 8092 and 8093

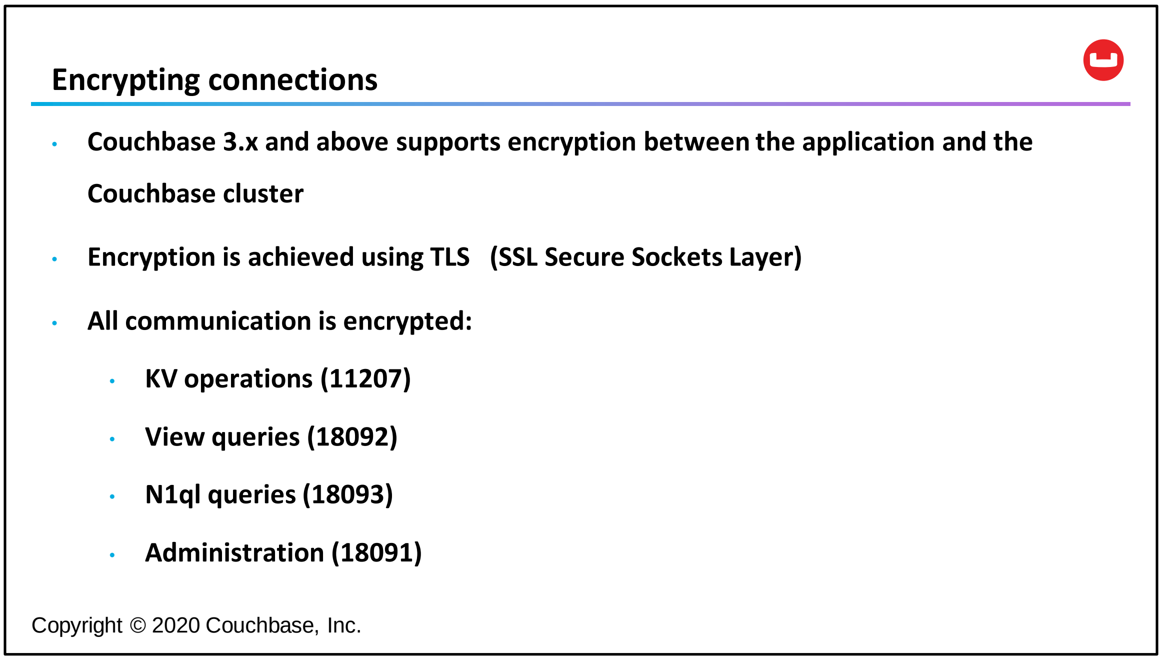
Closing bucket Connections (Java)

- Use the **CouchbaseCluster**
    - **The most common case is to disconnect the whole CouchbaseCluster from the server**
    - **which has the same effect as closing all buckets manually and in addition close all underlying resources like thread pools.**
    - **This also means that once disconnect has happened, you can't reopen buckets from the same Cluster instance.**
    - **if you pass in a custom CouchbaseEnvironement then you are responsible for shutting down the environment.**
    - **So do cluster.disconnect(); and then env.shutdown().toBlocking().single(); at the very end**



```
//Close a bucket without shutting down the whole CouchbaseCluster
Boolean closed = bucket.close();

//or Close the whole CouchbaseCluster - close all buckets
Boolean disconnected = cluster.disconnect();
```

Connection Best Practice 5: Encrypting Connections

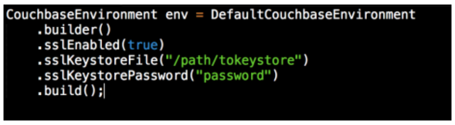


Encrypting connections

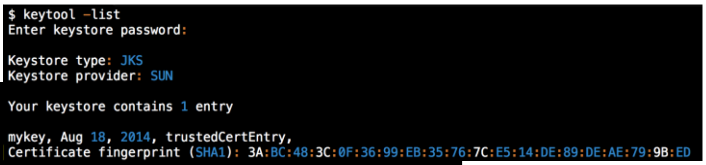
- Couchbase 3.x and above supports encryption between the application and the Couchbase cluster
- Encryption is achieved using TLS (SSL Secure Sockets Layer)
- All communication is encrypted:
  - KV operations (11207)
  - View queries (18092)
  - N1ql queries (18093)
  - Administration (18091)

Enable encryption

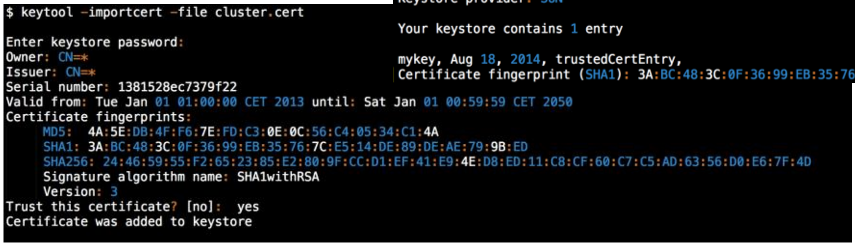
- Enabling in JAVA
  - Securely transfer your SSL certificate from the Couchbase cluster to the application server (using SSH and SCP)
  - Use the Java keytool command to import and verify it into your JVM keystore
  - Enable encryption on your CouchbaseEnvironment class



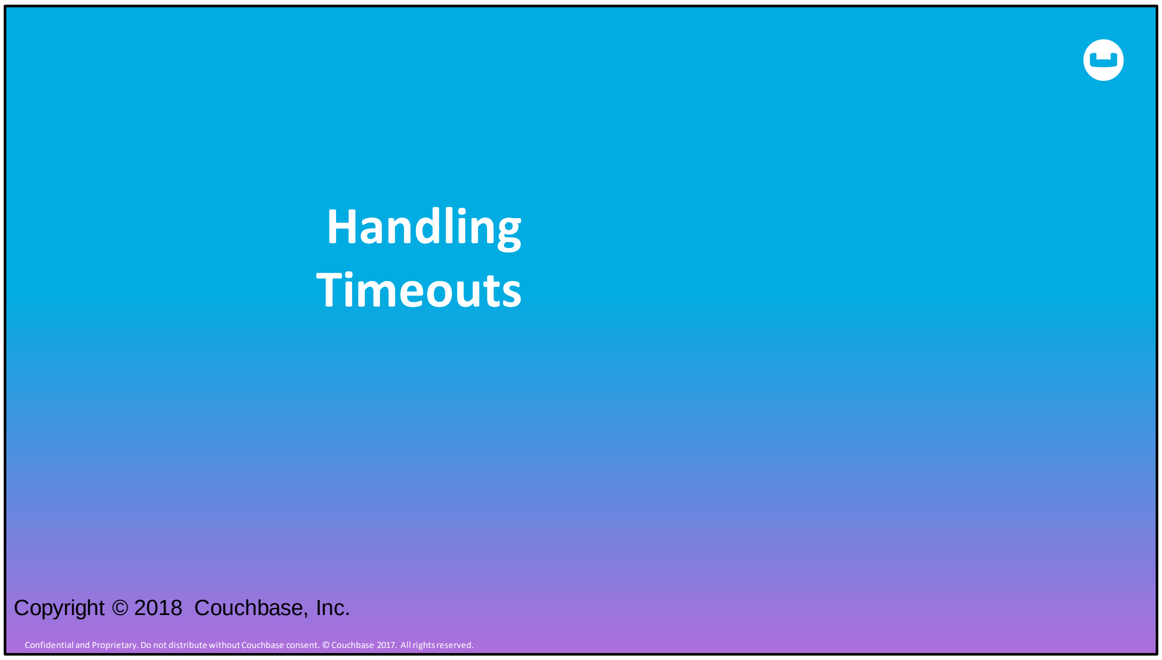
```
CouchbaseEnvironment env = DefaultCouchbaseEnvironment
    .builder()
    .sslEnabled(true)
    .sslKeystoreFile("/path/tokeystore")
    .sslKeystorePassword("password")
    .build();
```



```
$ keytool -list
Enter keystore password:

Keystore type: JKS
Keystore provider: SUN

Your keystore contains 1 entry

mykey, Aug 18, 2014, trustedCertEntry,
Certificate fingerprint (SHA1): 3A:BC:48:3C:0F:36:99:EB:35:76:7C:E5:14:DE:89:DE:AE:79:9B:ED
```



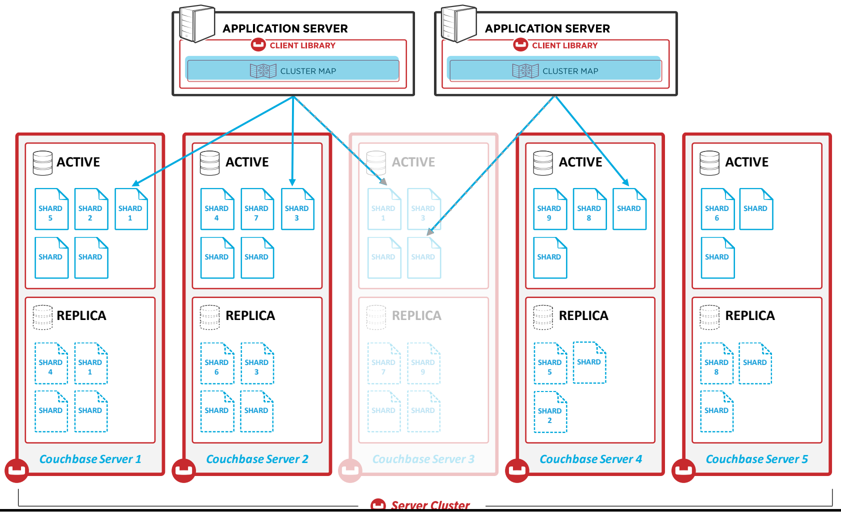
```
$ keytool -importcert -file cluster.cert

Enter keystore password:
Owner: CN=*
Issuer: CN=*
Serial number: 1381528ec7379f22
Valid from: Tue Jan 01 01:00:00 CET 2013 until: Sat Jan 01 00:59:59 CET 2050
Certificate fingerprints:
     MD5:  4A:5E:DB:4F:F6:7E:FD:C3:0E:0C:56:C4:05:34:C1:4A
     SHA1: 3A:BC:48:3C:0F:36:99:EB:35:76:7C:E5:14:DE:89:DE:AE:79:9B:ED
     SHA256: 24:46:59:55:F2:65:23:85:E2:80:9F:CC:D1:EF:41:E9:4E:D8:ED:11:C8:CF:60:C7:C5:AD:63:56:D0:E6:7F:4D
     Signature algorithm name: SHA1withRSA
     Version: 3
Trust this certificate? [no]:  yes
Certificate was added to keystore
```



Handling Timeouts

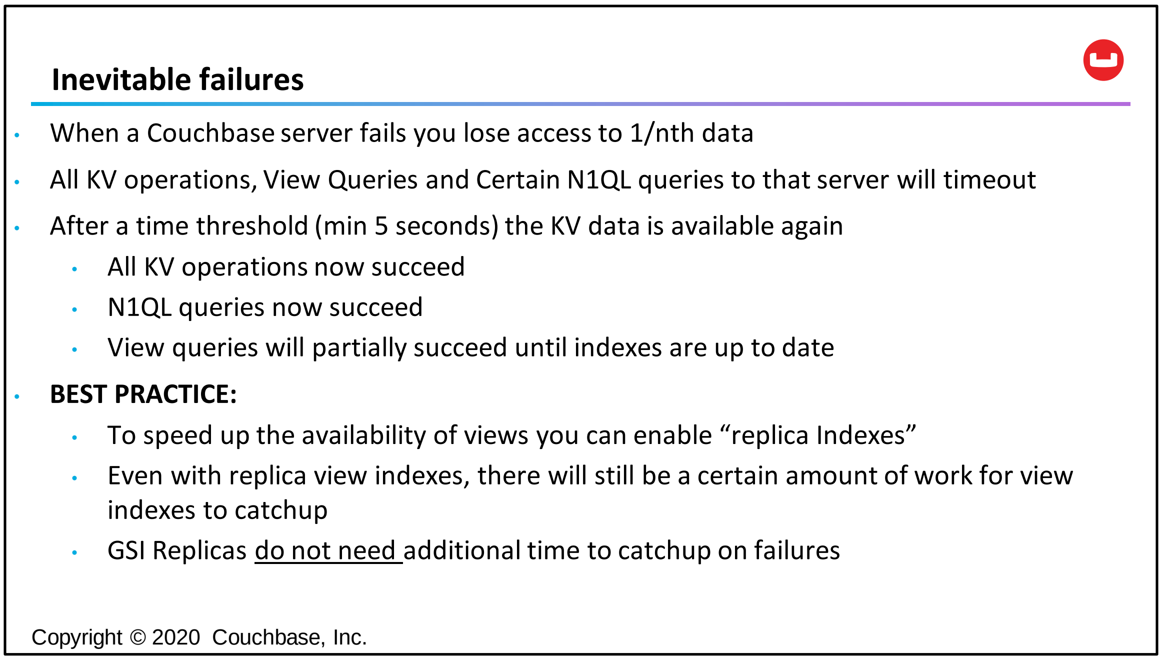
Inevitable Hardware Failures

Only thing certain in IT Infrastructure … H/W does fail



Application has single logical connection to cluster (client object)

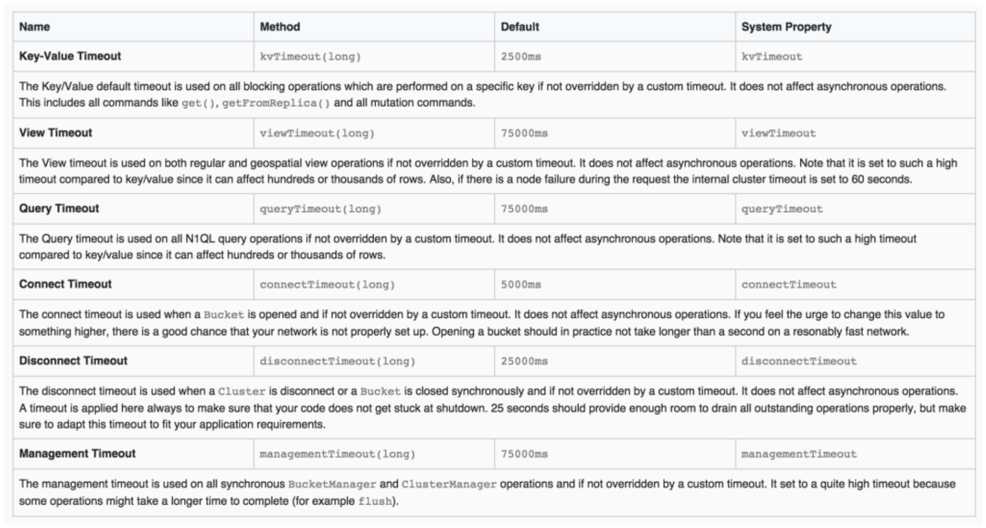
- When node goes down, some requests will fail
- Failover is either automatic or manual`
- Client library is automatically updated via cluster map
- Replicas not recreated to preserve stability
- Best practice to replace node and rebalance



Inevitable failures

- When a Couchbase server fails you lose access to 1/nth data
- All KV operations, View Queries and Certain N1QL queries to that server will timeout
- After a time threshold (min 5 seconds) the KV data is available again
  - All KV operations now succeed
  - N1QL queries now succeed
  - View queries will partially succeed until indexes are up to date
- **BEST PRACTICE:**
  - To speed up the availability of views you can enable “replica Indexes”
  - Even with replica view indexes, there will still be a certain amount of work for view indexes to catchup
  - GSI Replicas do not need additional time to catchup on failures

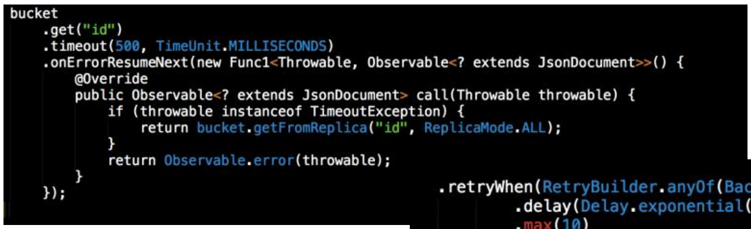
Handling Timeouts (Java)



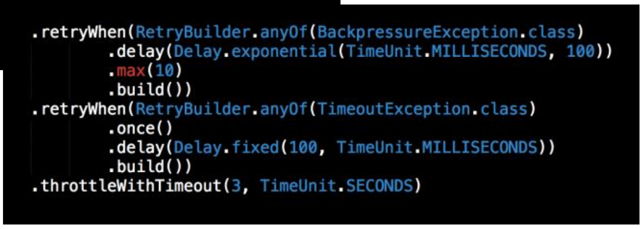
Name	Method	Default	System Property
Key-Value Timeout	`kvTimeout(long)`	`2500ms`	`kvTimeout`
The Key/Value default timeout is used on all blocking operations which are performed on a specific key if not overridden by a custom timeout. It does not affect asynchronous operations. This includes all commands like `get()`, `getFromReplica()` and all mutation commands.			
View Timeout	`viewTimeout(long)`	`75000ms`	`viewTimeout`
The View timeout is used on both regular and geospatial view operations if not overridden by a custom timeout. It does not affect asynchronous operations. Note that it is set to such a high timeout compared to key/value since it can affect hundreds or thousands of rows. Also, if there is a node failure during the request the internal cluster timeout is set to 60 seconds.			
Query Timeout	`queryTimeout(long)`	`75000ms`	`queryTimeout`
The Query timeout is used on all N1QL query operations if not overridden by a custom timeout. It does not affect asynchronous operations. Note that it is set to such a high timeout compared to key/value since it can affect hundreds or thousands of rows.			
Connect Timeout	`connectTimeout(long)`	`5000ms`	`connectTimeout`
The connect timeout is used when a `Bucket` is opened and if not overridden by a custom timeout. It does not affect asynchronous operations. If you feel the urge to change this value to something higher, there is a good chance that your network is not properly set up. Opening a bucket should in practice not take longer than a second on a resonably fast network.			
Disconnect Timeout	`disconnectTimeout(long)`	`25000ms`	`disconnectTimeout`
The disconnect timeout is used when a `Cluster` is disconnect or a `Bucket` is closed synchronously and if not overridden by a custom timeout. It does not affect asynchronous operations. A timeout is applied here always to make sure that your code does not get stuck at shutdown. 25 seconds should provide enough room to drain all outstanding operations properly, but make sure to adapt this timeout to fit your application requirements.			
Management Timeout	`managementTimeout(long)`	`75000ms`	`managementTimeout`
The management timeout is used on all synchronous `BucketManager` and `ClusterManager` operations and if not overridden by a custom timeout. It set to a quite high timeout because some operations might take a longer time to complete (for example `flush`).			

Handling Timeouts (Java)

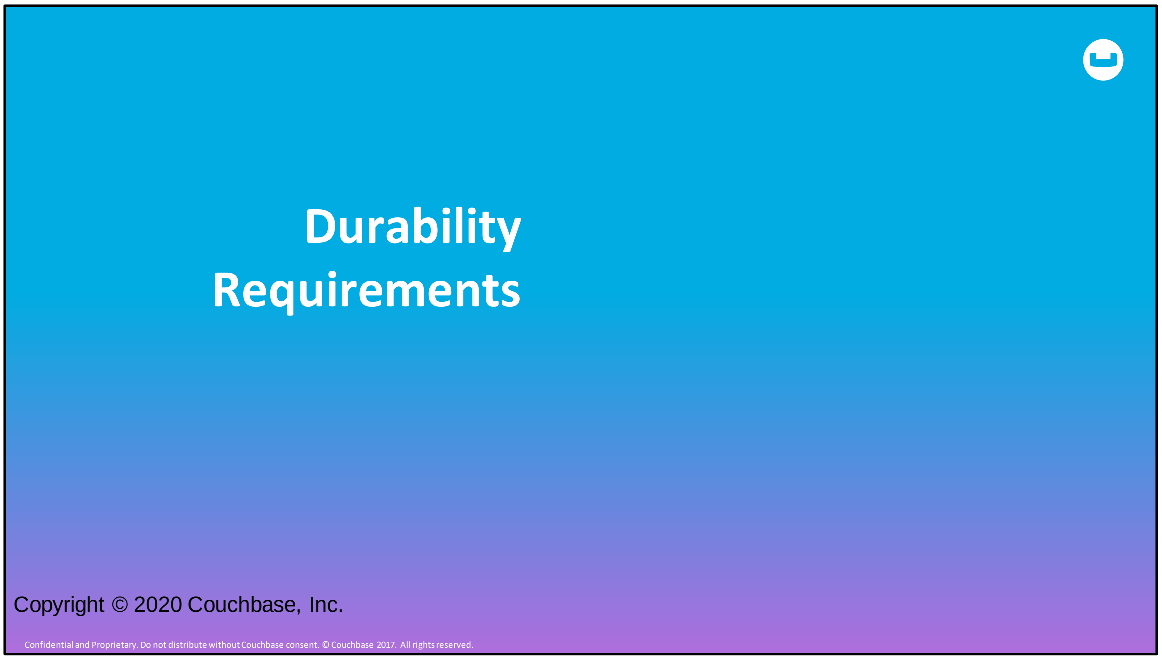
- Its easy to retry failures for TimeoutExceptions as well as other types of exceptions
- Note – reading from replica could mean you read a stale value.



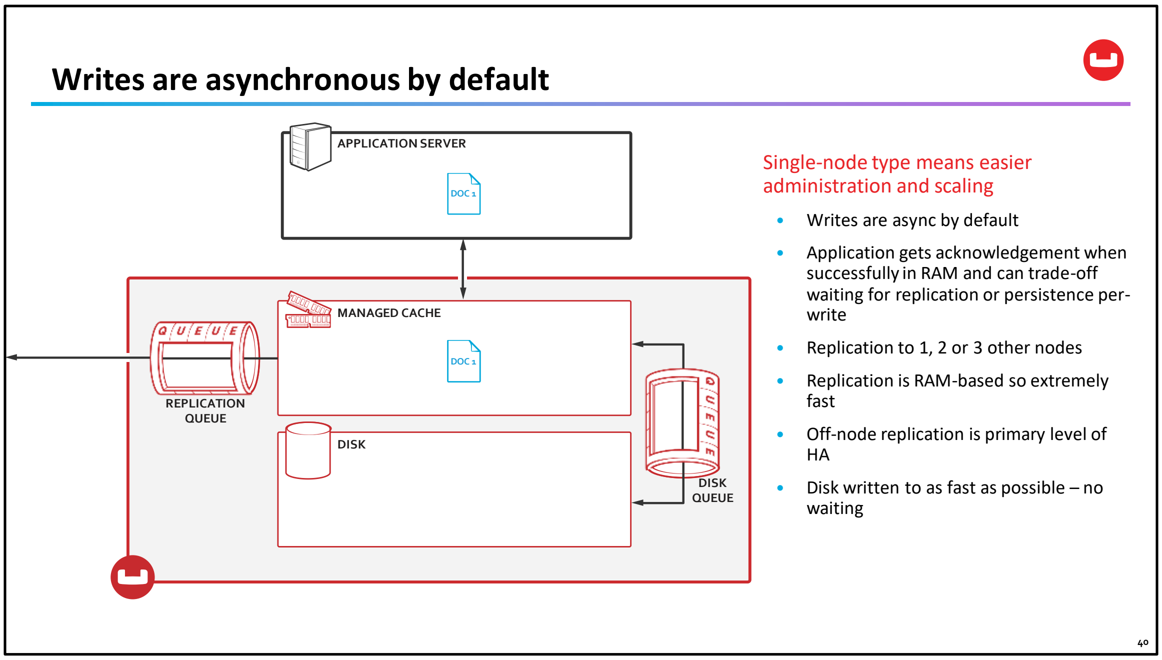
```java
bucket
    .get("id")
    .timeout(500, TimeUnit.MILLISECONDS)
    .onErrorResumeNext(new Func1<Throwable, Observable<? extends JsonDocument>>() {
        @Override
        public Observable<? extends JsonDocument> call(Throwable throwable) {
            if (throwable instanceof TimeoutException) {
                return bucket.getFromReplica("id", ReplicaMode.ALL);
            }
            return Observable.error(throwable);
        }
    });
```



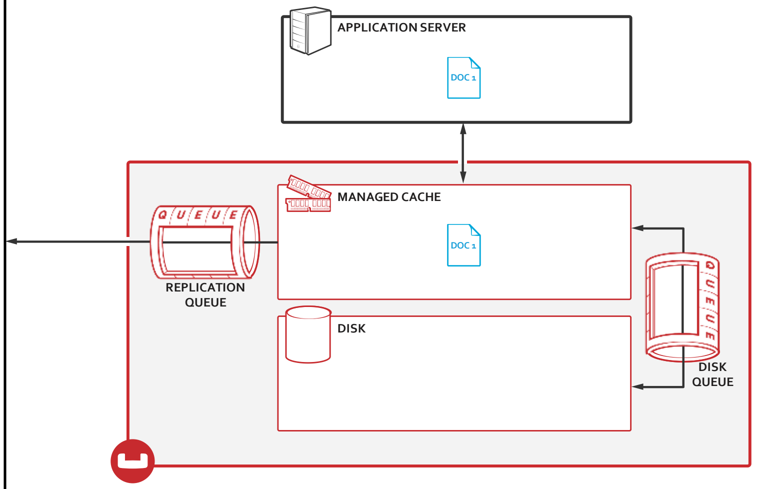
```java
.retryWhen(RetryBuilder.anyOf(BackpressureException.class)
        .delay(Delay.exponential(TimeUnit.MILLISECONDS, 100))
        .max(10)
        .build())
.retryWhen(RetryBuilder.anyOf(TimeoutException.class)
        .once()
        .delay(Delay.fixed(100, TimeUnit.MILLISECONDS))
        .build())
.throttleWithTimeout(3, TimeUnit.SECONDS)
```



Durability Requirements



Writes are asynchronous by default



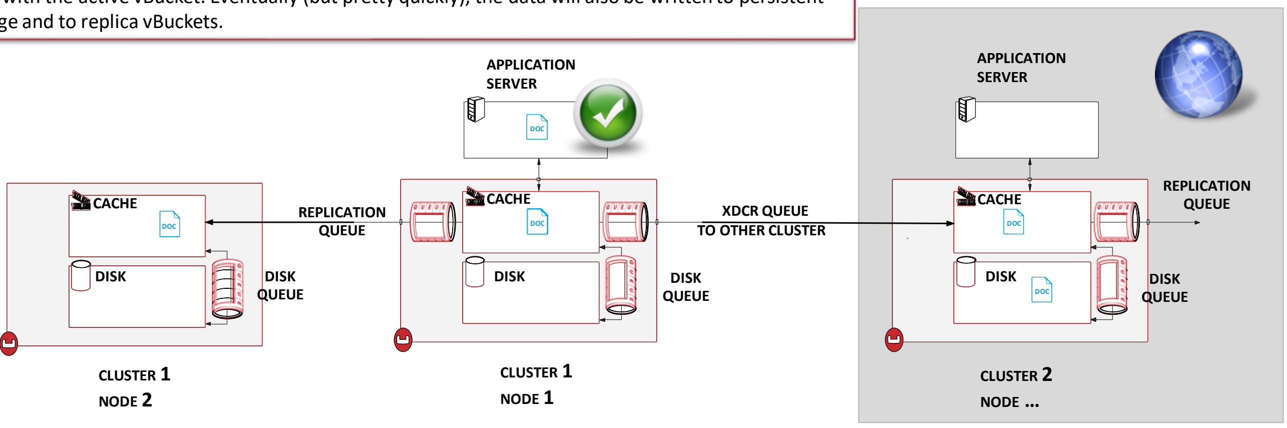
Single-node type means easier administration and scaling

- Writes are async by default
- Application gets acknowledgement when successfully in RAM and can trade-off waiting for replication or persistence per-write
- Replication to 1, 2 or 3 other nodes
- Replication is RAM-based so extremely fast
- Off-node replication is primary level of HA
- Disk written to as fast as possible – no waiting

40

Key/Value – Durability Methods

1 (**PUT**) By default, a write operation is successful when the data has been written to the memory of the node with the active vBucket. Eventually (but pretty quickly), the data will also be written to persistent storage and to replica vBuckets.

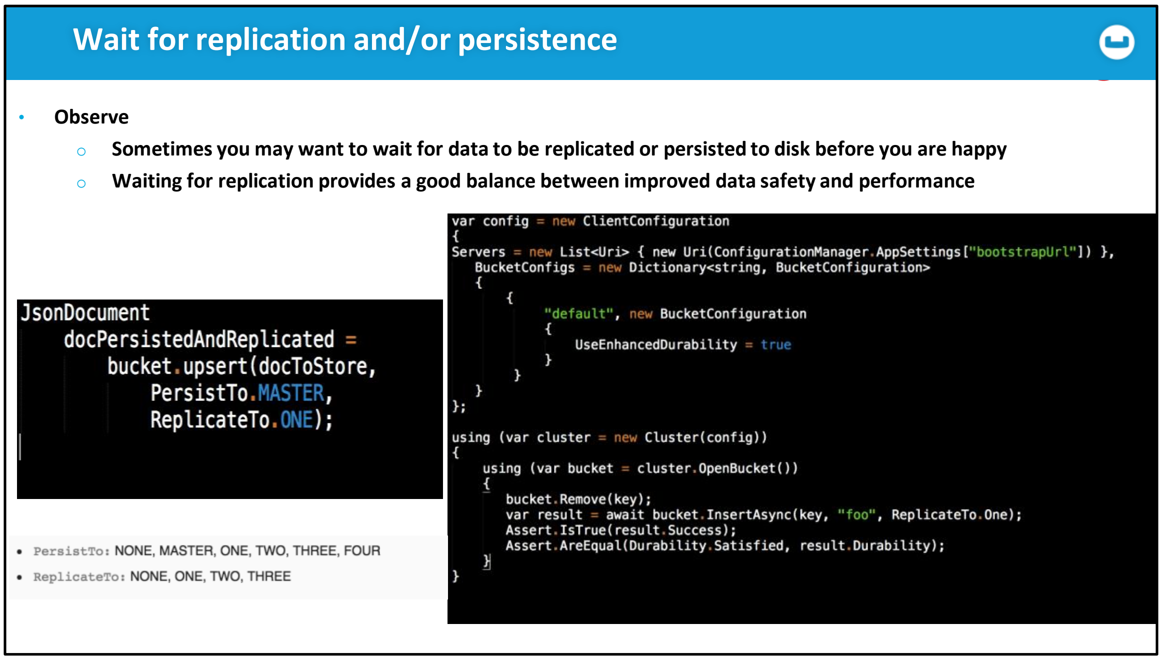


2 (**replicateTo**) write operation is successful when data has been written to both the active vBucket and replica vBuckets.

3 (**persistTo**) write operation is successful when data has been written to persistent storage

- Can be set to require writes to persistent storage for replicas.
- Using persistTo will slow performance.

Replicate to majority
Persist to majority
Persist to master,& replicate to majority



Wait for replication and/or persistence

- **Observe**
  - **Sometimes you may want to wait for data to be replicated or persisted to disk before you are happy**
  - **Waiting for replication provides a good balance between improved data safety and performance**

```
JsonDocument
    docPersistedAndReplicated =
        bucket.upsert(docToStore,
            PersistTo.MASTER,
            ReplicateTo.ONE);
```

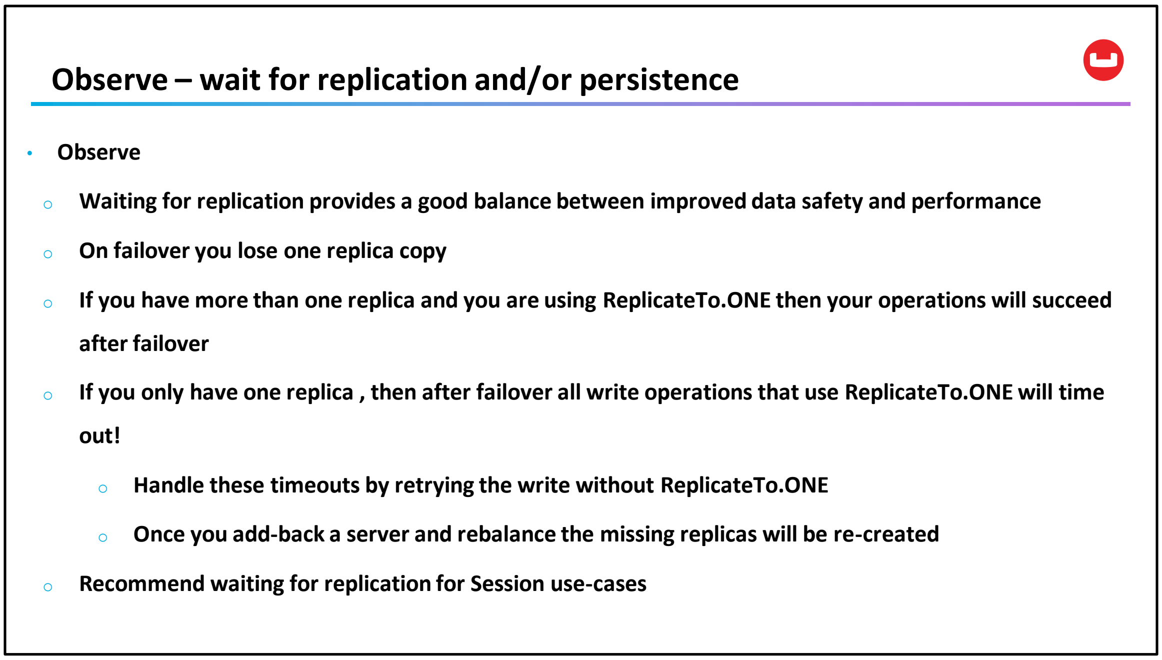
- `PersistTo:` NONE, MASTER, ONE, TWO, THREE, FOUR
- `ReplicateTo:` NONE, ONE, TWO, THREE

```
var config = new ClientConfiguration
{
Servers = new List<Uri> { new Uri(ConfigurationManager.AppSettings["bootstrapUrl"]) },
   BucketConfigs = new Dictionary<string, BucketConfiguration>
   {
       {
            "default", new BucketConfiguration
            {
                UseEnhancedDurability = true
            }
        }
   }
};

using (var cluster = new Cluster(config))
{
    using (var bucket = cluster.OpenBucket())
    {
       bucket.Remove(key);
       var result = await bucket.InsertAsync(key, "foo", ReplicateTo.One);
       Assert.IsTrue(result.Success);
       Assert.AreEqual(Durability.Satisfied, result.Durability);
    }
}
```

persistedto ... will not return untill it writes to memory and to disk.

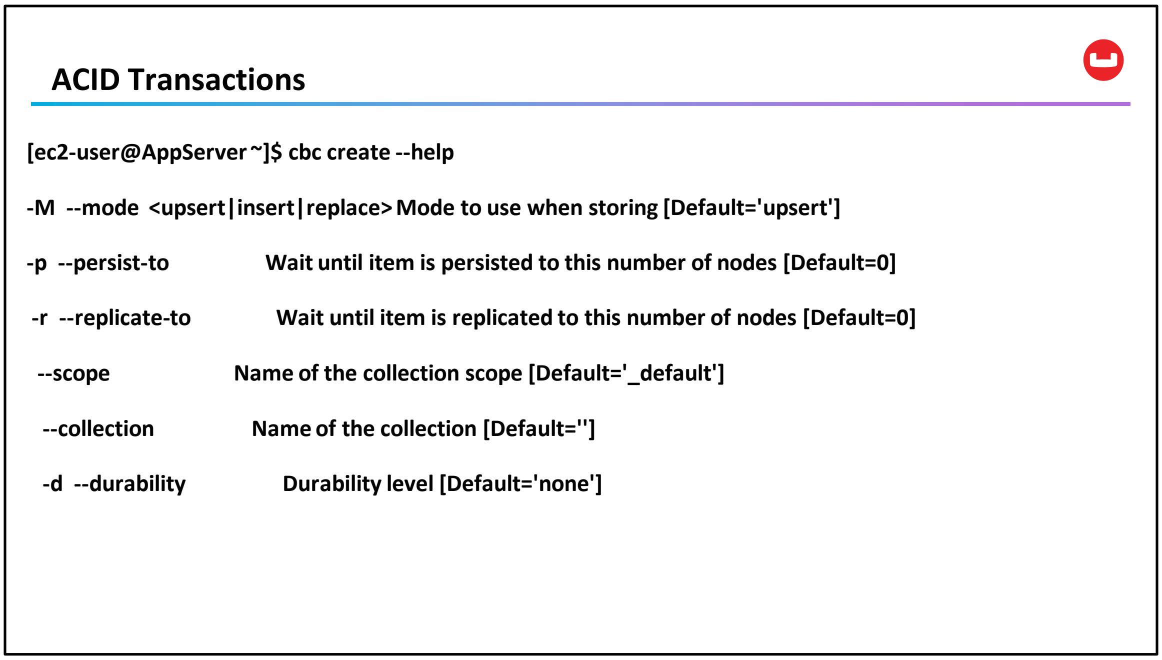
replicateto -- will return when the number of repilca have been created.

FOUR
Persist to at least four nodes including Master.
MASTER
Persist to the Master.
ONE
ONE implies MASTER.
THREE
Persist to at least three nodes including Master.
TWO
Persist to at least two nodes including Master.
ZERO
Don't wait for persistence on any nodes.



Observe – wait for replication and/or persistence

- Observe
  - Waiting for replication provides a good balance between improved data safety and performance
  - On failover you lose one replica copy
  - If you have more than one replica and you are using ReplicateTo.ONE then your operations will succeed after failover
  - If you only have one replica , then after failover all write operations that use ReplicateTo.ONE will time out!
    - Handle these timeouts by retrying the write without ReplicateTo.ONE
    - Once you add-back a server and rebalance the missing replicas will be re-created
  - Recommend waiting for replication for Session use-cases



ACID Transactions

```
[ec2-user@AppServer ~]$ cbc create --help

-M  --mode  <upsert|insert|replace> Mode to use when storing [Default='upsert']

-p  --persist-to            Wait until item is persisted to this number of nodes [Default=0]

 -r  --replicate-to          Wait until item is replicated to this number of nodes [Default=0]

  --scope                 Name of the collection scope [Default='_default']

   --collection            Name of the collection [Default='']

   -d  --durability            Durability level [Default='none']
```

A durable write is synchronous, and is supported by durability. Such a write may be appropriate when saving data whose loss could have a considerable, negative impact. For example, data corresponding to a financial transaction.

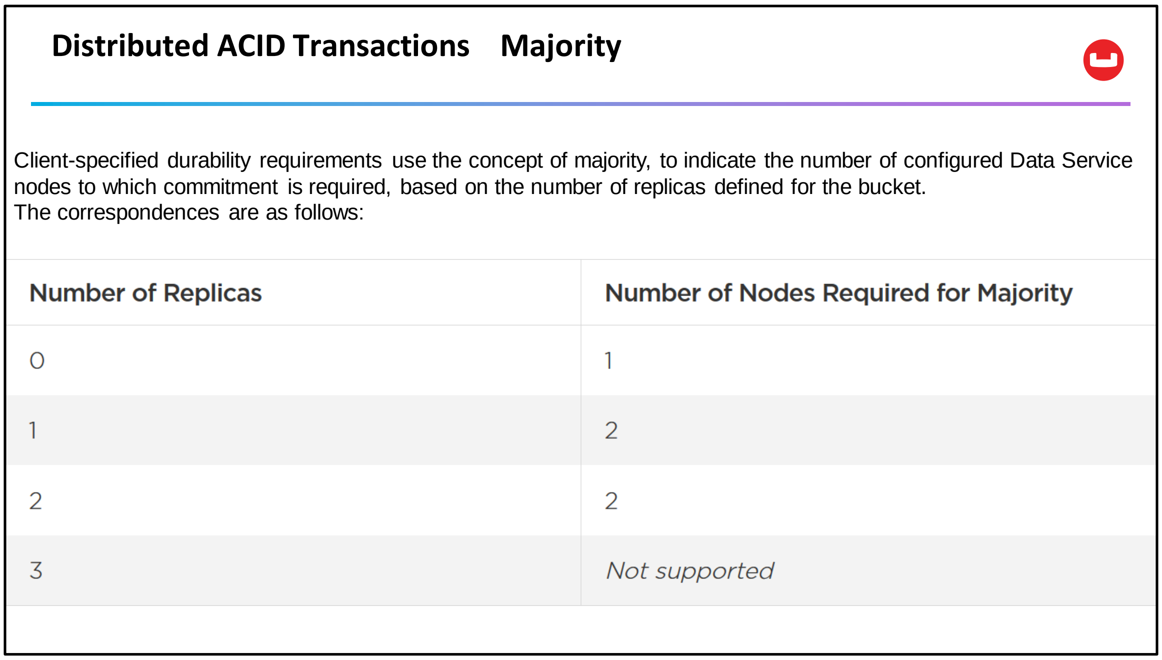
The durability requirements specified by a client are:

Level. The level of durability required. The possible values are:

majority. The mutation must be replicated to (that is, held in the memory allocated to the bucket on) a majority of the Data Service nodes.

majorityAndPersistActive. The mutation must be replicated to a majority of the Data Service nodes. Additionally, it must be persisted (that is, written and synchronised to disk) on the node hosting the active vBucket for the data.

persistToMajority. The mutation must be persisted to a majority of the Data Service nodes. Accordingly, it will be written to disk on those nodes.



Distributed ACID Transactions Majority

Client-specified durability requirements use the concept of majority, to indicate the number of configured Data Service nodes to which commitment is required, based on the number of replicas defined for the bucket.
The correspondences are as follows:

Number of Replicas	Number of Nodes Required for Majority
0	1
1	2
2	2
3	*Not supported*

A durable write is synchronous, and is supported by durability. Such a write may be appropriate when saving data whose loss could have a considerable, negative impact. For example, data corresponding to a financial transaction.

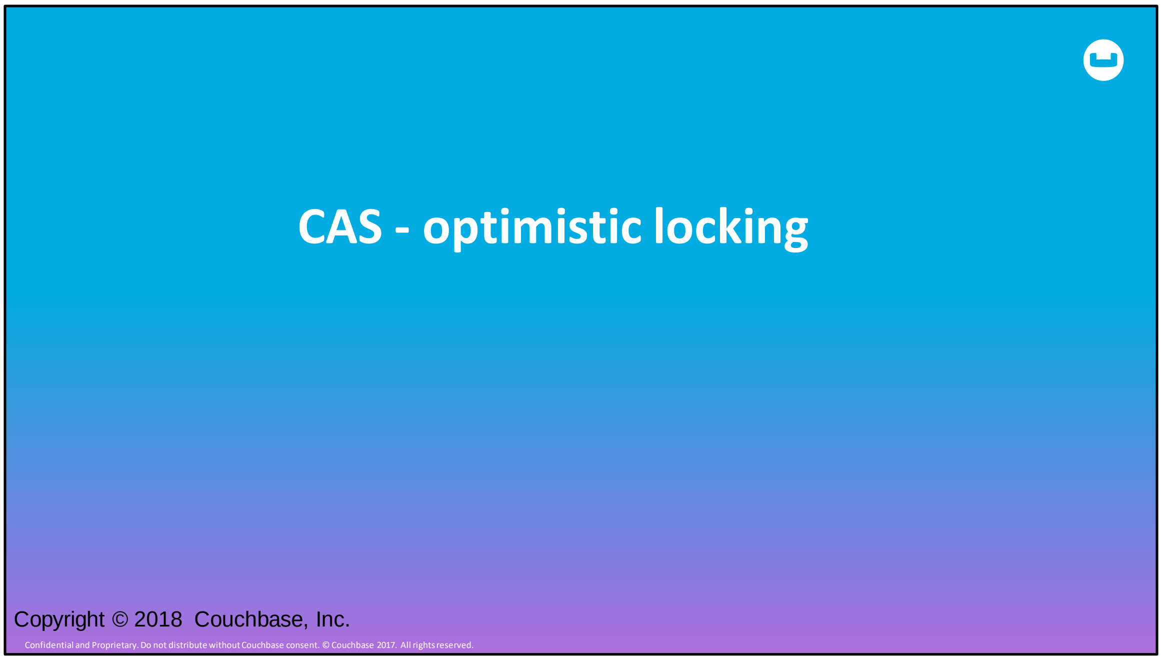
The durability requirements specified by a client are:

Level. The level of durability required. The possible values are:

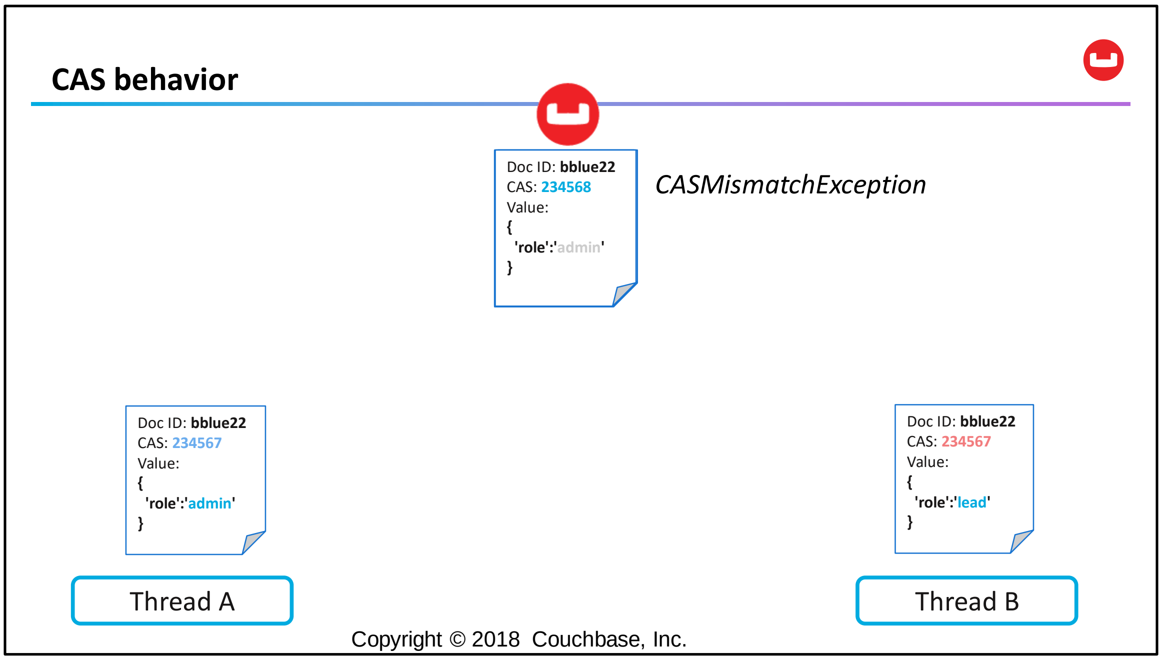
majority. The mutation must be replicated to (that is, held in the memory allocated to the bucket on) a majority of the Data Service nodes.

majorityAndPersistActive. The mutation must be replicated to a majority of the Data Service nodes. Additionally, it must be persisted (that is, written and synchronised to disk) on the node hosting the active vBucket for the data.

persistToMajority. The mutation must be persisted to a majority of the Data Service nodes. Accordingly, it will be written to disk on those nodes.



CAS - optimistic locking



CAS behavior

CAS is acronym for "Check and Set" and is useful for ensuring that a mutation of a document by one user or thread does not override another near simultaneous mutation by another user or thread. The CAS value is returned by the server with the result when you perform a read on a document using Get or when you perform a mutation on a document using Insert, Upsert, Replace or Remove.

In certain situations you may want to ensure that you are not overwriting another users changes when you perform a mutation operation on a document. For this you have two choices: optimistic concurrency or pessimistic concurrency (locking). Optimistic concurrency tends to provide better performance than pessimistic concurrency, because no actual locks are help on the data.The pattern for optimistic concurrency is simple:

Perform a Get on a document to retrieve it
Edit the document as desired
Perform a mutation on the document and pass the CAS as an argument
If the mutation fails and the key exists, get the latest revision and apply mutation and then submit it again
Repeat until success

CAS implementation

- static **create** (*ID*, [expiry], content, [CAS])

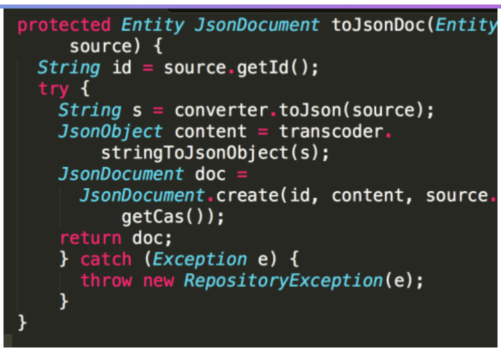
✓ factory for *JsonDocument* creation

✓ ID assigned to document as metadata

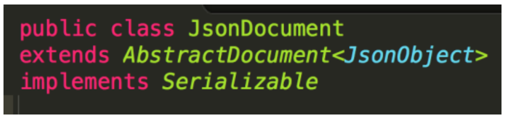
✓ CAS may also be assigned (persisted)

- To implement CAS:

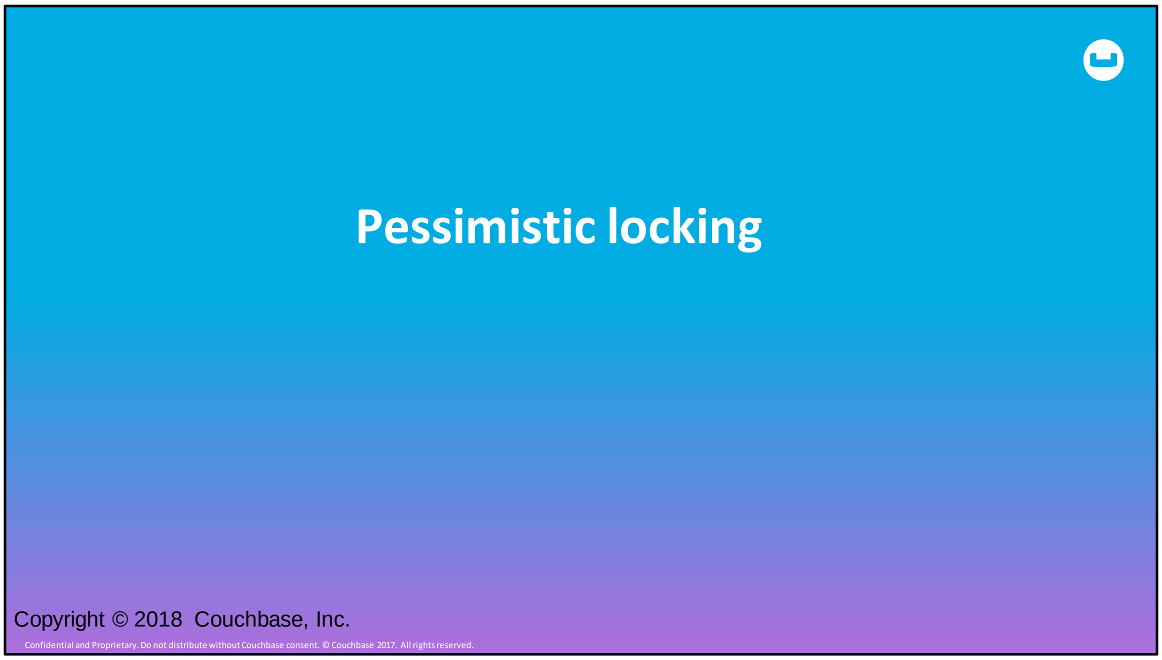
1. Persist CAS values in *Document* objects
2. Use *replace()* and *remove()* operations, not *upsert()*



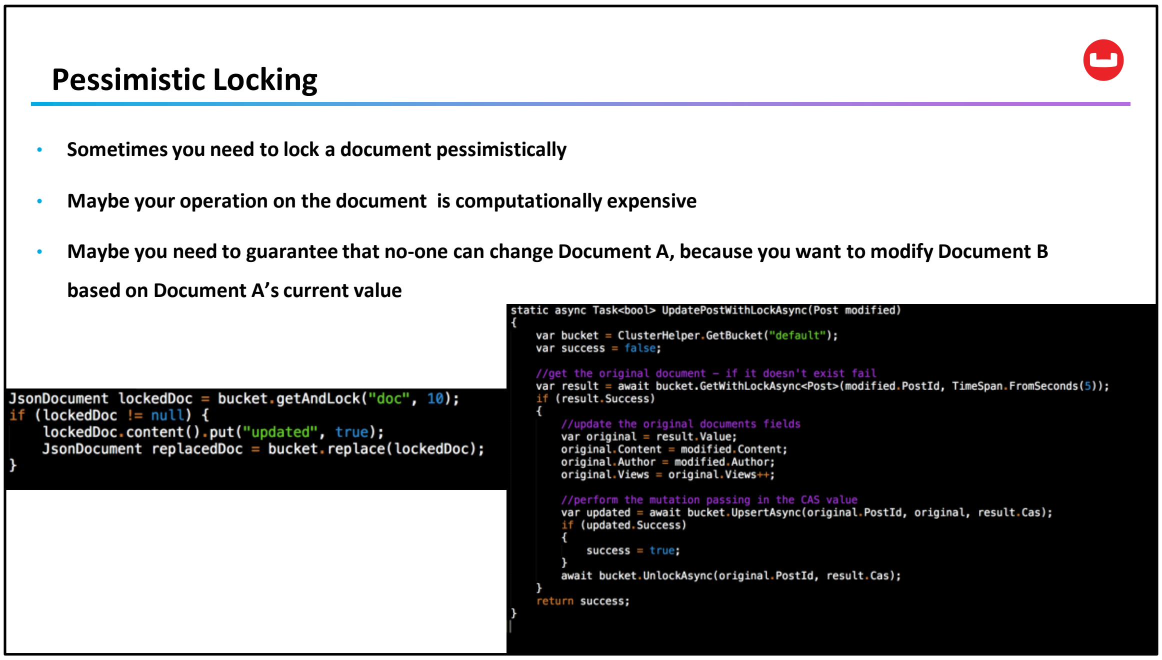
```
protected Entity JsonDocument toJsonDoc(Entity
     source) {
  String id = source.getId();
  try {
    String s = converter.toJson(source);
    JsonObject content = transcoder.
        stringToJsonObject(s);
    JsonDocument doc =
      JsonDocument.create(id, content, source.
        getCas());
    return doc;
    } catch (Exception e) {
      throw new RepositoryException(e);
    }
}
```



```
public class JsonDocument
extends AbstractDocument<JsonObject>
implements Serializable
```



Pessimistic locking



Pessimistic Locking

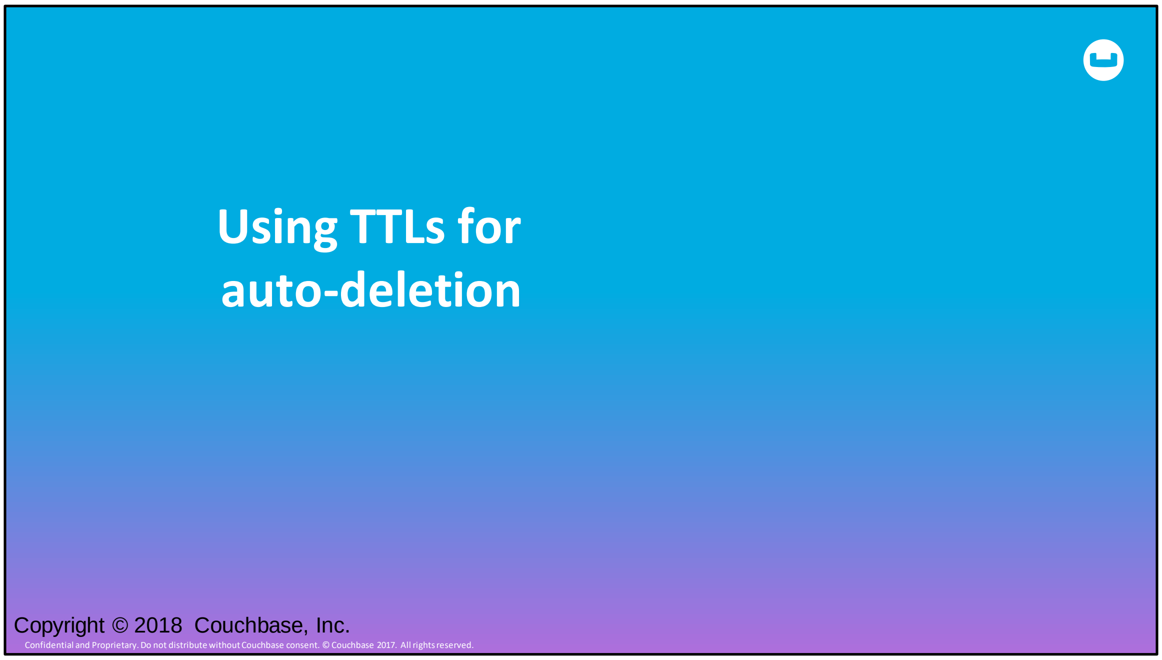
- Sometimes you need to lock a document pessimistically
- Maybe your operation on the document is computationally expensive
- Maybe you need to guarantee that no-one can change Document A, because you want to modify Document B based on Document A’s current value

```
JsonDocument lockedDoc = bucket.getAndLock("doc", 10);
if (lockedDoc != null) {
    lockedDoc.content().put("updated", true);
    JsonDocument replacedDoc = bucket.replace(lockedDoc);
}
```

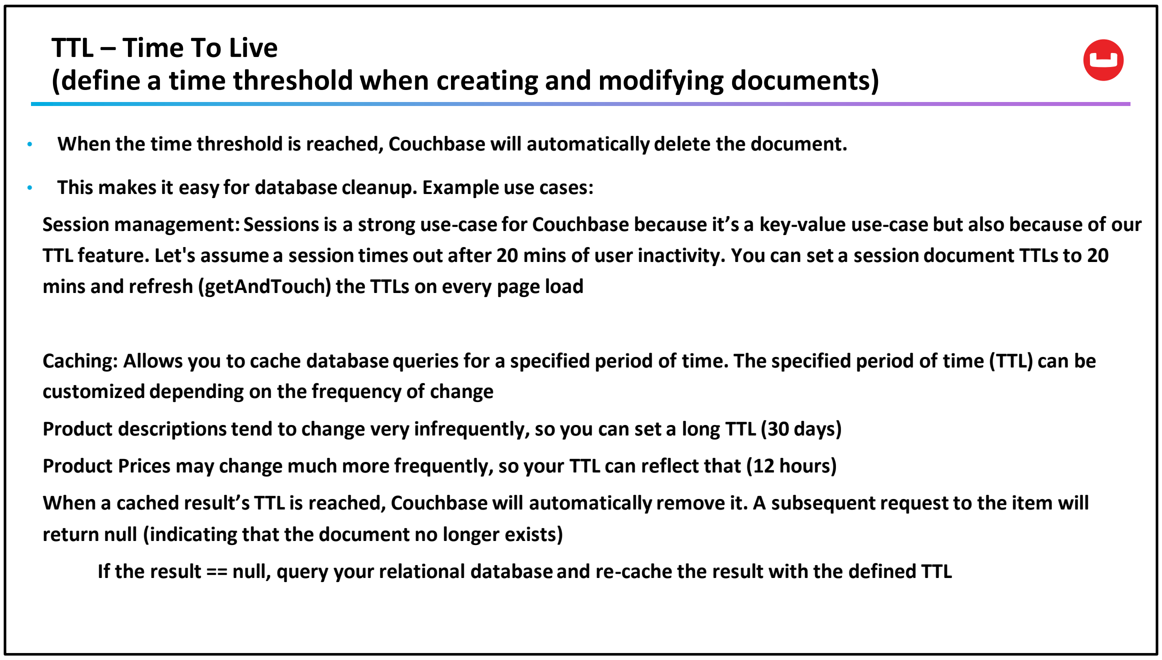
```
static async Task<bool> UpdatePostWithLockAsync(Post modified)
{
    var bucket = ClusterHelper.GetBucket("default");
    var success = false;

    //get the original document - if it doesn't exist fail
    var result = await bucket.GetWithLockAsync<Post>(modified.PostId, TimeSpan.FromSeconds(5));
    if (result.Success)
    {
        //update the original documents fields
        var original = result.Value;
        original.Content = modified.Content;
        original.Author = modified.Author;
        original.Views = original.Views++;

        //perform the mutation passing in the CAS value
        var updated = await bucket.UpsertAsync(original.PostId, original, result.Cas);
        if (updated.Success)
        {
            success = true;
        }
        await bucket.UnlockAsync(original.PostId, result.Cas);
    }
    return success;
}
```

While CAS on an operation handles optimistic concurrency without requiring explicit locks, pessimistic concurrency can be achieved by using Lock and Unlock. Using Lock ensures that no other users can update the key or value if it is locked. Unlock removes the lock held on key so that it can be mutated by other users.

This is roughly equivalent to the optimistic concurrency example. First, we get the document and lock it, giving us a 5 second lock to complete the operation. Then we update the original document and replace it. Finally, we unlock the document using the original CAS value from the GetWithLockAsync call.



Using TTLs for auto-deletion



TTL – Time To Live
(define a time threshold when creating and modifying documents)

- When the time threshold is reached, Couchbase will automatically delete the document.
- This makes it easy for database cleanup. Example use cases:

Session management: Sessions is a strong use-case for Couchbase because it's a key-value use-case but also because of our TTL feature. Let's assume a session times out after 20 mins of user inactivity. You can set a session document TTLs to 20 mins and refresh (getAndTouch) the TTLs on every page load

Caching: Allows you to cache database queries for a specified period of time. The specified period of time (TTL) can be customized depending on the frequency of change

Product descriptions tend to change very infrequently, so you can set a long TTL (30 days)

Product Prices may change much more frequently, so your TTL can reflect that (12 hours)

When a cached result's TTL is reached, Couchbase will automatically remove it. A subsequent request to the item will return null (indicating that the document no longer exists)

If the result == null, query your relational database and re-cache the result with the defined TTL

In some situations you may want to limit the lifetime of a document so that is automatically evicted after some set duration. TTL or "time to live" is supported by both Memcached and Couchbase buckets and allows you to control the lifespan of a document.

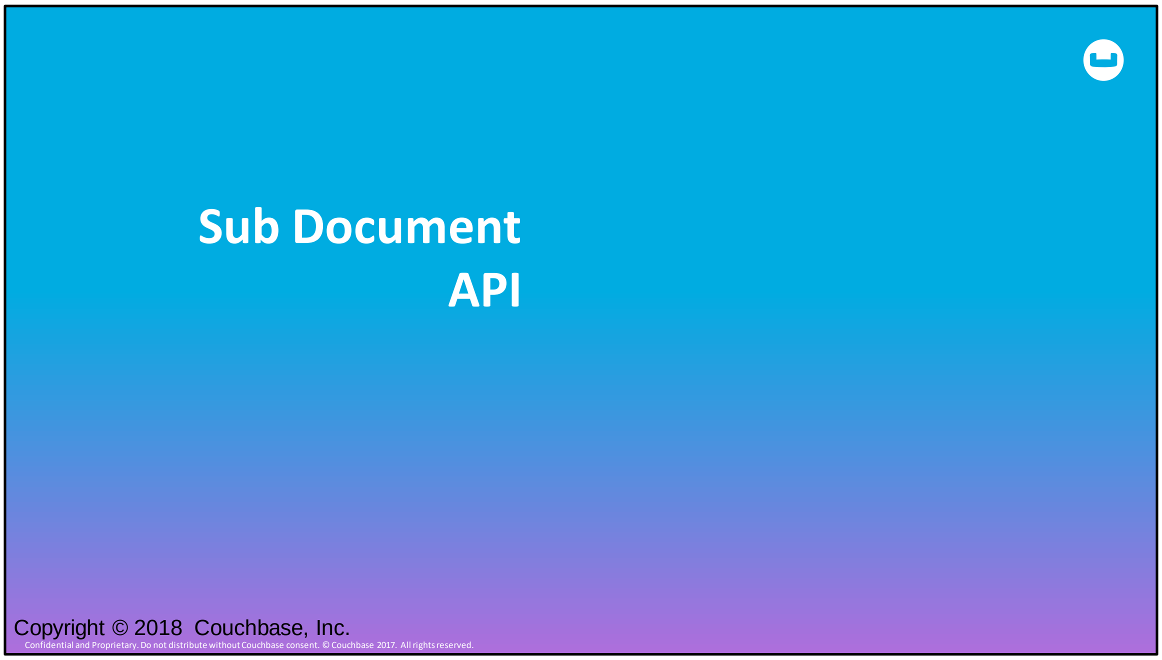
For example, perhaps you have a scenario where the most recent documents are read quite often and older docs, less often. To keep your memory footprint small, you want to expire older documents if there not accessed after some arbitrary time-span. In this case you could use TTL to provide a sliding expiration where if the document is accessed within a set amount of time, the expiry will be updated. If it's not read within the duration, the document will be evicted from the cache.

TTL – Time To Live examples (Java)

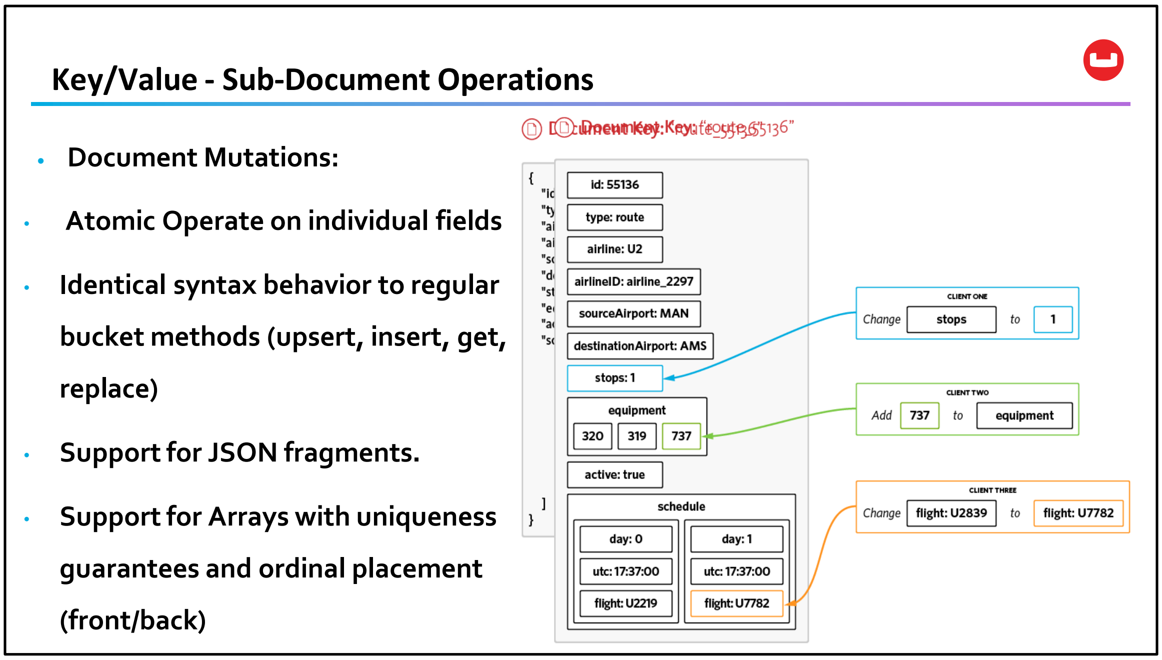
- **Here I create a session object with a TTL of 60 seconds**
- **For every page load I will update the TTL using getAndTouch**

```
//Session creation – initial TTL is 60 second
JsonDocument sesionObject = JsonDocument.create(GUUID, 60, session);
bucket.upsert(sesionObject);

//Update TTL for every page load
bucket.getAndTouch(GUUID, 60); //refresh the TTL by 60 seconds
```

- Instead using the relative number of seconds (since the creation) you could also use a Unix time stamp. For example, the value 1421454149 represents Saturday, 17 January 2017 at 00:22:29 UTC.

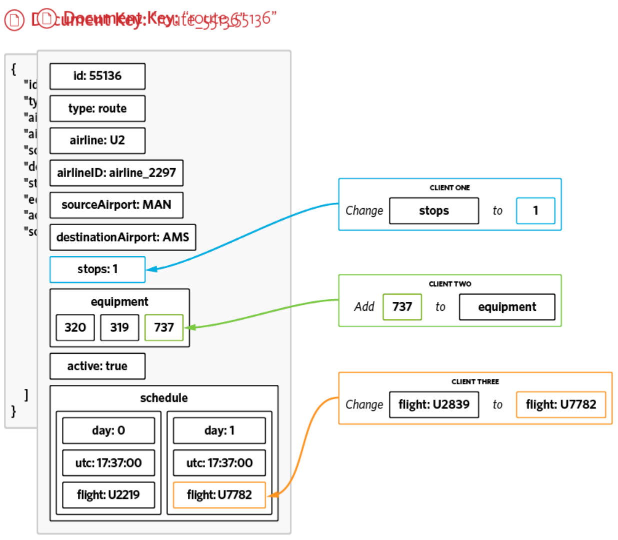


Sub Document API



Key/Value - Sub-Document Operations

- Document Mutations:
- Atomic Operate on individual fields
- Identical syntax behavior to regular bucket methods (upsert, insert, get, replace)
- Support for JSON fragments.
- Support for Arrays with uniqueness guarantees and ordinal placement (front/back)



Sub-Document Operations

Edit on GitHub

Sub-document operations can be used to efficiently access parts of documents. Sub-document operations may be quicker and more network-efficient than full-document operations such as upsert, update and get because they only transmit the accessed sections of the document over the network. Sub-document operations are also atomic, allowing safe modifications to documents with built-in concurrency control.
Sub-documents
Sub-documents

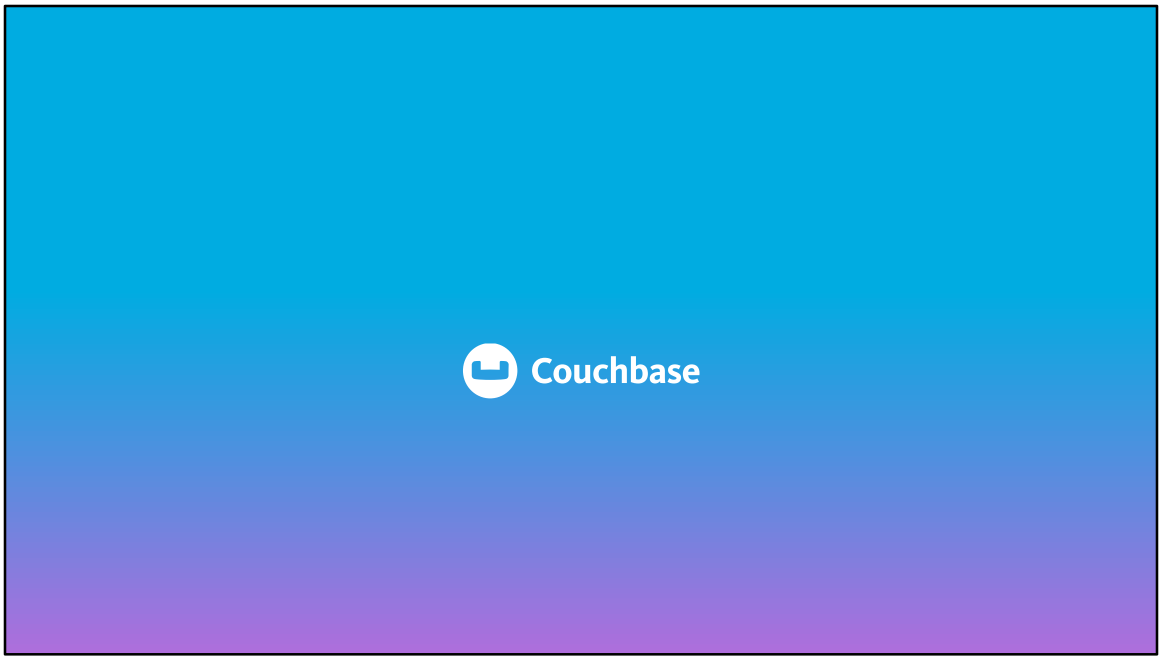
Starting with Couchbase Server 4.5 you can atomically and efficiently update and retrieve parts of a document. These parts are called sub-documents. While full-document retrievals retrieve the entire document and full document updates require sending the entire document, sub-document retrievals only retrieve relevant parts of a document and sub-document updates only require sending the updated portions of a document. You should use sub-document operations when you are modifying only portions of a document, and full-document operations when the contents of a document is to change significantly.

Pattern – Sub Document

```
DocumentFragment<Lookup> lookupRslt = bucket
      .lookupIn(”keyname")
      .get("parent.name")
      .execute();
```

```
MutateInBuilder builder = bucket.mutateIn(”keyname");
DocumentFragment<Mutation> result = builder.replace("parent.name",
40).execute();
```

- **Lab 2.0**



Couchbase