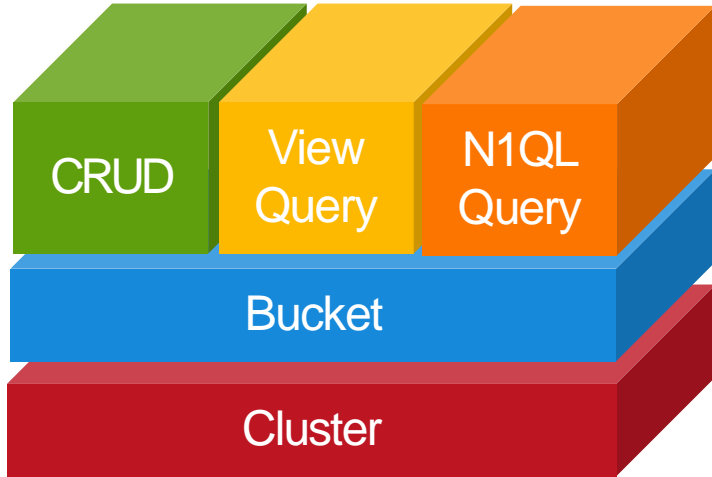


# Couchbase – Java SDK

# Couchbase SDKs

---

- What does it mean to be a Couchbase SDK?



# Languages and Interfaces for Couchbase

- Official SDKs
  - Java – Version 2.5
  - .NET – Version 2.1
  - Node.js – Version 2.1
  - Python – Version 2.0
  - PHP – Version 2.0
  - C – Version 2.5
  - Go – Version 1.0
  - Ruby – Version 2.0 DP
- For each of these we have
  - Full Document support
  - Interoperability
  - Common yet idiomatic Programming Model

Others: Erlang, Perl, TCL, Clojure, Scala



JDBC and ODBC

# The Document

---

- Documents are integral to the SDKs.
- There are many implementations, depending on the content type.
- A Document contains:

Property	Description
<b>ID</b>	The bucket-unique identifier
<b>Content</b>	The value that is stored
<b>Expiry</b>	An expiration time
<b>CAS</b>	The Compare-And-Swap identifier

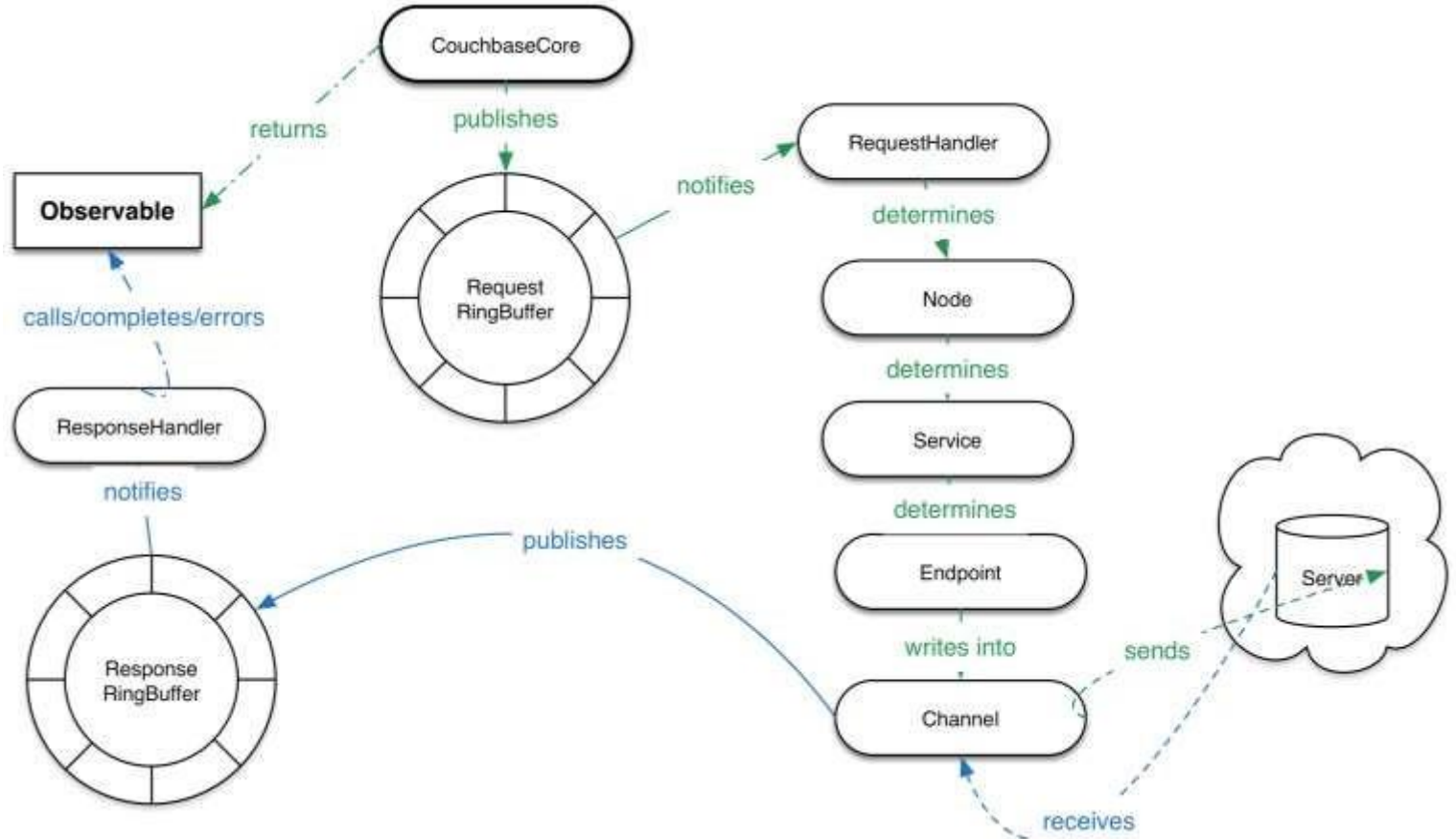
---

# Couchbase Core IO

---

- Common infrastructure & feature set for all language bindings
  - Message oriented
  - Asynchronous only
  - Low overhead and performance focused
  - Supports Java 6+ (including 8!)
- 
- Disruptor RingBuffer for implicit batching and backpressure
  - Netty for high performance IO

# Detour: Java Client Architecture



Couchbase

## Installing the SDK

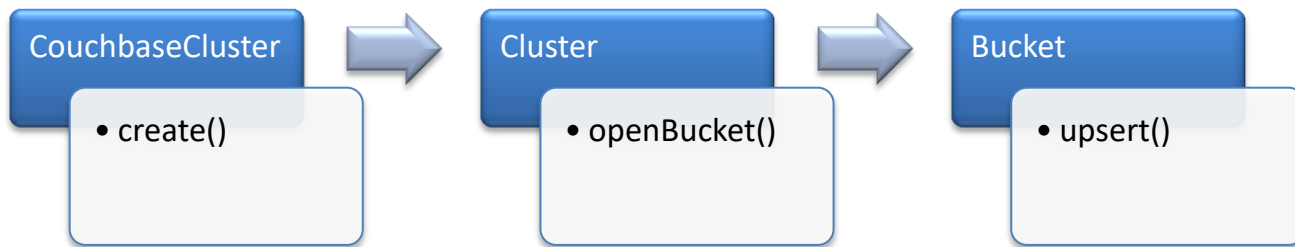
---

```

❏ <dependencies>
❏   <dependency>
      <groupId>com.couchbase.client</groupId>
      <artifactId>java-client</artifactId>
      <version>2.5.4</version>
    </dependency>
  </dependencies>
```

# Couchbase Connection

---



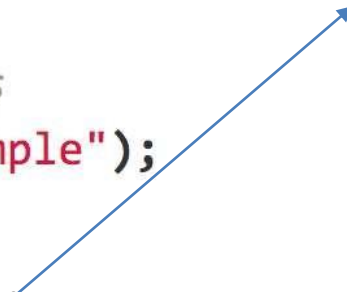
The actual process of connecting to a bucket (that is, opening sockets and everything related) happens when you call the openBucket method:



# Connecting

---

```
1 // Create a cluster reference
2 CouchbaseCluster cluster = CouchbaseCluster.create("192.168.56.101");
3
4 // Open a bucket and establish all resources
5 Bucket bucket = cluster.openBucket("beer-sample");
```



a few seed nodes so that the client is able to establish initial contact.

# From Sync to Async

---

```
1  // Create a cluster reference
2  CouchbaseCluster cluster = CouchbaseCluster.create("192.168.56.101");
3
4  // Open a bucket and establish all resources
5  Bucket bucket = cluster.openBucket("beer-sample");
6
7  // Get a reference to the underlying async bucket
8  AsyncBucket asyncBucket = bucket.async();
```

# Cluster

---

When your application shuts down, you need to make sure to properly disconnect from the cluster to free all resources (sockets, threads, and so on).

This disconnects all buckets and frees associated resources.

```
cluster.disconnect();
```

# Connection Manager

---

ConnectionManager class has been created as a singleton

```
public class ConnectionManager {

    private static final ConnectionManager connectionManager = new ConnectionManager();

    public static ConnectionManager getInstance() {
        return connectionManager;
    }

    static Cluster cluster = CouchbaseCluster.create();
    static Bucket bucket = cluster.openBucket("beer-sample");

    public static void disconnect() {
        cluster.disconnect().toBlocking().single();
    }
}
```

it is important to reuse the Couchbase connections so that the underlying resources are not duplicated for each connection.

# Document Operations

---

Access documents in Couchbase using methods of the `couchbase.couchbase.client.java.Bucket`

- `get()` and `lookupIn()`
- `upsert()`, `insert()`, `replace()` and `mutateIn()`.

# JsonDocument

---

Couchbase Server uses the JSON format as a first-class citizen. It is used for querying (via both views and N1QL) and represents the main storage format that should be used.

The **JsonDocument** class has factory methods named **create()** that you use to create documents.

```
JsonDocument doc = JsonDocument.create("id", content);
```



The content needs to be of type **JsonObject**, which ships with the Java SDK.

# CAS and Expiry

---

Every Document also contains the expiry and **cas** properties.

They are considered meta information and are optional.

An expiration time of 0 means that no expiration is set at all, and a **CAS** value 0 means it won't be used.

You can set the expiry to control when the document should be deleted:

```
// Expire in 10 seconds.  
JsonDocument.create("id", 10, content);  
  
// Expire in 1 day.  
JsonDocument.create("id", TimeUnit.DAYS.toSeconds(1), content);
```

The CAS value can either be set by you directly or is populated by the SDK when the Document is loaded from the server (which is the recommended way to use it).

# JsonObject

---

An empty JSON document can be created like this:

JsonObject content = JsonObject.empty();

```
JSONArray friends = JSONArray.empty()
    .add(JsonObject.empty().put("name", "Mike Ehrmantraut"))
    .add(JsonObject.empty().put("name", "Jesse Pinkman"));

JsonObject content = JsonObject.empty()
    .put("firstname", "Walter")
    .put("lastname", "White")
    .put("age", 52)
    .put("aliases", JSONArray.from("Walt Jackson", "Mr. Mayhew", "David Lynn"))
    .put("friends", friends);
```



# JsonObject

---

```
"firstname": "Walter",  
"aliases": [  
    "Walt Jackson",  
    "Mr. Mayhew",  
    "David Lynn"  
],  
"age": 52,  
"friends": [  
    {  
        "name": "Mike Ehrmantraut"  
    },  
    {  
        "name": "Jesse Pinkman"  
    }  
],  
"lastname": "White"
```

generates a JSON document like

# JsonObject

---

```
JsonObject content = JsonObject.empty()
.put("firstname", "Walter");

JsonDocument walter = JsonDocument.create("user:walter", content);
JsonDocument inserted = bucket.insert(walter);
int age = content.getInt("age");
String name = content.getString("firstname") + content.getString("lastname");
```

the **JsonObject** and **JsonArray** classes provide convenience methods to generate and modify them.

If you want to read values out of the **JsonDocument**, you can use either the typed or untyped getter methods

# JsonArrayDocument

---

```
JsonArray content = JsonArray.from("Hello", "World", 1234);  
bucket.upsert(JsonArrayDocument.create("docWithArray", content));
```

The **JsonArrayDocument** class works exactly like the **JsonDocument** class, with the main difference that you can have a JSON array at the top level content (instead of an object).



```
docWithArray  
1  [  
2    "Hello",  
3    "World",  
4    1234  
5  ]
```

# RawJsonDocument

---

```
// write the raw data
String content = "{\"hello\": \"couchbase\", \"active\": true}";
bucket.upsert(RawJsonDocument.create("rawJsonDoc", content));
```

The **JsonObject** and **JsonArray** convenience.  
In a lot of places though, custom JSON handling is already in place through libraries like Jackson or Google GSON

```
// prints RawJsonDocument{id='rawJsonDoc', cas=..., expiry=0, content={"hello":
"couchbase", "active": true}}
System.out.println(bucket.get("rawJsonDoc", RawJsonDocument.class));

// read it parsed
// prints true
System.out.println(bucket.get("rawJsonDoc").content().getBoolean("active"));
```

# SerializableDocument

---

Any object that implements **Serializable** can be safely encoded and decoded using the built-in Java serialization mechanism.

While it is very convenient, it can be slow in cases where the POJOs are very complex and deeply nested.

```
public class User implements Serializable {
    private final String username;
    public User(String username) {
        this.username = username;
    }
    public String getUsername() {
        return username;
    }
}

// Create the User and store it
bucket.upsert(SerializableDocument.create("user::michael", new User("Michael")));
// Read it back
SerializableDocument found = bucket.get("user::michael", SerializableDocument.class);
// Print a property to verify
System.out.println(((User) found.content()).getUsername());
```

# StringDocument

---

```
// Create the document
bucket.upsert(StringDocument.create("stringDoc", "Hello World"));

// Prints:
// StringDocument{id='stringDoc', cas=1424054670330, expiry=0, content=Hello World}
System.out.println(bucket.get("stringDoc", StringDocument.class));
```

It should not be mistaken with the **JsonStringDocument** which automatically quotes it and also flags it as JSON.  
If a String is stored through it, it is explicitly flagged as a non-JSON string.

# Creating and Updating Full Documents

---

```
JsonDocument doc = JsonDocument.create("document_id", JsonObject.create().put("some", "value"));
System.out.println(bucket.insert(doc));
```

Output: JsonDocument{id='document\_id', cas=216109389250560, expiry=0, content={"some":"value"}, mutationToken=null}

```
JsonDocument doc = JsonDocument.create("document_id", JsonObject.empty().put("some", "other value"));
System.out.println(bucket.upsert(doc));
```

Output: JsonDocument{id='document\_id', cas=216109392920576, expiry=0, content={"some":"other value"}, mutationToken=null}

```
JsonDocument doc = JsonDocument.create("document_id", JsonObject.empty().put("more", "content"));
System.out.println(bucket.replace(doc));
```

Output: JsonDocument{id='document\_id', cas=216109395083264, expiry=0, content={"more":"content"}, mutationToken=null}

## Retrieving full documents

---

```
System.out.println(bucket.get("document_id"));
```

Output: JsonDocument{id='document\_id', cas=216109395083264, expiry=0, content={"more":"content"}, mutationToken=null}

```
// Use a Document where ID is extracted  
JsonDocument someDoc = JsonDocument.create("document_id");  
System.out.println(bucket.get(someDoc));
```

Output: JsonDocument{id='document\_id', cas=216109395083264, expiry=0, content={"more":"content"}, mutationToken=null}



## Retrieving full documents ..

---

```
// Wait only 1 second instead of the default timeout  
JsonDocument doc = bucket.get("document_id", 1, TimeUnit.SECONDS);
```

```
Iterator<JsonDocument> docIter = bucket.getFromReplica("document_id");  
while(docIter.hasNext()) {  
    JsonDocument replicaDoc = docIter.next();  
    System.out.println(replicaDoc);  
}
```

## Retrieving full documents ..

---

```
// Get and Lock for max of 10 seconds
JsonDocument ownedDoc = bucket.getAndLock("document_id", 10);

// Do something with your document
JsonDocument modifiedDoc = modifyDocument(ownedDoc);

// Write it back with the correct CAS
bucket.replace(modifiedDoc);
```

## Removing full documents

---

```
// Remove the document  
JsonDocument removed = bucket.remove("document_id");
```

```
JsonDocument loaded = bucket.get("document_id");  
  
// Remove and take the CAS into account  
JsonDocument removed = bucket.remove(loaded);
```

# Deleting documents

---

can remove a document by utilizing the `remove()` method

```
// Remove the document by its ID.  
Observable<JsonDocument> doc = bucket.remove("id");
```

The returned **Document** only has the **id** populated, all other fields are set to their default values.

```
// Remove the document and make sure the delete is replicated.  
Observable<JsonDocument> doc = bucket.remove("id", ReplicateTo.ONE);  
  
// Remove the document and make sure the delete is persisted and replicated.  
Observable<JsonDocument> doc = bucket.remove("id", PersistTo.MASTER,  
ReplicateTo.ONE);
```

If no durability requirements are set on the remove method, the operation will succeed when the server acknowledges the document delete in its managed cache layer.

# Modifying expiration

---

```
int expiry = 2; // seconds
JsonDocument stored = bucket.upsert(
    JsonDocument.create("expires", expiry, JsonObject.create().put("some", "value"))
);

Thread.sleep(3000);

System.out.println(bucket.get("expires"));
```

null

```
bucket.touch("expires", 2);
```

## Reading and touching

---

Reading and touching works very similar to a regular read, but it also refreshes the expiration time of the document to the specified value.

```
// Get and set the new expiration time to 4 seconds  
Observable<JsonDocument> doc = bucket.getAndTouch("id", 4);
```

You can also use the **touch()** command if you do not want to read the document and just refresh its expiration time.

# Retrieving documents

---

```
// Read from all available replicas and the master node and return all responding
bucket.getFromReplica("id", ReplicaMode.ALL);

// Read only from the first replica
bucket.getFromReplica("id", ReplicaMode.FIRST);
```

## Reading from replica

A regular read always reads the document from its master node. If this node is down or not available, the document cannot be loaded.

Reading from replica allows you to load the document from one or more replica nodes instead.

Note that if `ReplicaMode.ALL` is used, requests are sent to the master node and all configured replicas.

# Retrieving documents

---

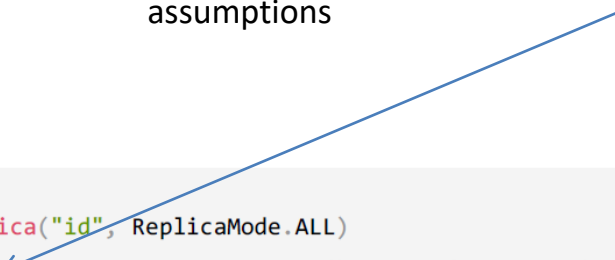
The main goal is to get responses back as fast as possible, but because more requests are sent, more responses can arrive.

You can use this to either compare all of the responding documents and draw conclusions from that, or just pick the first one arriving:

bucket

```
.getFromReplica("id", ReplicaMode.ALL)
.first()
.subscribe();
```

add operations to filter based on some assumptions



bucket

```
.getFromReplica("id", ReplicaMode.ALL)
.filter(document -> document.content().getInt("version") > 5)
.first()
.subscribe();
```



# Operating with Sub-Documents

```
bucket.lookupIn("docid")
    .get("path.to.get")
    .exists("check.path.exists")
    .execute();

boolean createParents = true;
bucket.mutateIn("docid")
    .upsert("path.to.upsert", value, createParents)
    .remove("path.to.del")
    .execute();
```

Allows to check that a response for a particular spec

```
DocumentFragment<Lookup> res =
bucket.lookupIn("docid")
    .get("foo")
    .exists("bar")
    .exists("baz")
    .execute();

// First result
res.content("foo");
// or
res.content(0);
```

## 2.x API

---

```
1  JsonDocument      get(String id);
2  Iterator<JsonDocument> getFromReplica(String id, ReplicaMode type);
3  JsonDocument      getAndLock(String id, int lockTime);
4  Boolean            exists(String id);
5  JsonDocument      getAndTouch(String id, int expiry);
6
7  Document           insert(Document document);
8  Document           upsert(Document document);
9  Document           replace(Document document);
10 JsonDocument       remove(String id);
11
12 ViewResult         query(ViewQuery query);
13 SpatialViewResult  query(SpatialViewQuery query);
14 N1qlQueryResult    query(N1qlQuery query);
15
16 Boolean            unlock(String id, long cas);
17 Boolean            touch(String id, int expiry);
18 JsonLongDocument   counter(String id, long delta, long initial, int expiry);
19 Document           append(Document document);
20 Document           prepend(Document document);
```

## Documents with JSON content

Document Name	Description
<code>JsonDocument</code>	The default, which has a <code>JsonObject</code> at the top level content.
<code>RawJsonDocument</code>	Stores any JSON value and should be used if custom JSON serializers such as Jackson or GSON are already in use.
<code>JsonArrayDocument</code>	Similar to <code>JsonDocument</code> , but has a <code>JsonArray</code> at the top level content.
<code>JsonStringDocument</code>	Stores JSON compatible <code>String</code> values. Input is automatically wrapped with quotes when stored.
<code>EntityDocument</code>	Used with the <code>Repository</code> implementation to write and read POJOs into JSON and back.

## Additional options

---

- Expiry (or *TTL*) value which will instruct the server to delete the document after a given amount of time.
- CAS value to protect against concurrent updates to the same document.
- Durability Requirements

# Data Structures

---

```
bucket.mapAdd("map_id", "name", "Mark Nunberg",  
    MutationOptionBuilder.builder().createDocument(true));
```

```
bucket.listAppend("list_id", "hello",  
    MutationOptionBuilder.builder().createDocument(true));
```

Data structures are stored on the Couchbase server as ordinary JSON documents.

# Accessing Data Structures

---

```
bucket.listGet("list_id", 0, String.class); // "hello"  
bucket.mapGet("map_id", "name", String.class); // "mark nunberg"
```

Most data access methods will return a generic type  $V$  which is provided as a generic parameter such as `Class<V>`.

```
Map<String, String> favorites = new CouchbaseMap<String>("mapDocId", bucket);  
favorites.put("color", "Blue");  
favorites.put("flavor", "Chocolate");  
  
System.out.println(favorites); //=> {flavor=Chocolate, color=Blue}  
  
// What does the JSON document look like?  
System.out.println(bucket.get("mapDocId").content());  
//=> {"flavor":"Chocolate","color":"Blue"}
```

## Durability Requirements

---

If no durability requirements are set on the insert or upsert methods, the operation will succeed when the server acknowledges the document in its managed cache layer.

While this is a performant operation, there might be situations where you want to make sure that your document has been persisted and/or replicated so that it survives power outages and other node failures.

# Durability Requirements

---

```
Observable<D> insert(D document, PersistTo persistTo);  
Observable<D> insert(D document, ReplicateTo replicateTo);  
Observable<D> insert(D document, PersistTo persistTo, ReplicateTo replicateTo);  
  
Observable<D> upsert(D document, PersistTo persistTo);  
Observable<D> upsert(D document, ReplicateTo replicateTo);  
Observable<D> upsert(D document, PersistTo persistTo, ReplicateTo replicateTo);
```

Insert the document and make sure it is persisted to the master node



```
bucket.insert(document, PersistTo.MASTER);  
  
// Insert the document and make sure it is replicate to one replica node  
bucket.insert(document, ReplicateTo.ONE);  
  
// Insert the document and make sure it is persisted to one node and replicated to two  
bucket.insert(document, PersistTo.ONE, ReplicateTo.TWO);
```



**Labs - Development**

**Couchbase - Java API Configuration - Web Application (Basic)**