Couchbase Client Library: Java 1.2

Couchbase

Couchbase Client Library: Java 1.2

Abstract

This is the manual for 1.2 of the Couchbase Java client library, which is compatible with Couchbase Server 2.0.

This manual provides a reference to the key features and best practice for using the Java Couchbase Client library (`couchbase-client`).

Table 1. Product Compatibility for Couchbase SDK Java

Product	All Features
Couchbase Server 2.0	✓

External Community Resources.

Download Client Library
JavaDoc
Couchbase Developer Guide 2.0
Couchbase Server Manual 2.0
Java Client Library
SDK Forum
Wiki: Java Client Library

Last document update: 06 Sep 2013 00:27; *Document built*: 06 Sep 2013 00:27.

Documentation Availability and Formats. This documentation is available *online: HTML Online* . For other documentation from Couchbase, see Couchbase Documentation Library

Contact: editors@couchbase.com or couchbase.com

Table of Contents

List of Figures

List of Tables

Chapter 1. Getting Started

This chapter will teach you the basics of Couchbase Server and how to interact with it through the Java Client SDK. Here's a quick outline of what you'll learn in this chapter:

1. Create a project in your favorite IDE and set up the dependencies.
2. Write a simple program to demonstrate connecting to Couchbase and saving some documents.
3. Write a program to demonstrate using Create, Read, Update, Delete (CRUD) operations on documents in combination with JSON serialization and deserialization.
4. Explore some of the API methods that will provide more specialized functions.

At this point we assume that you have a Couchbase Server 2.0 release running and you have the "beer-sample" bucket configured. If you need any help on setting up everything, see the following documents:

- Using the Couchbase Web Console, for information on using the Couchbase Administrative Console,
- Couchbase CLI, for the command line interface,
- Couchbase REST API, for creating and managing Couchbase resources.

Warning

The TCP/IP port allocation on Windows by default includes a restricted number of ports available for client communication. For more information on this issue, including information on how to adjust the configuration and increase the available ports, see MSDN: Avoiding TCP/IP Port Exhaustion.

1.1. Preparation

Installing Couchbase Server

You will need the latest version of Couchbase Server. You can get the latest Couchbase Server 2.0 release and install it.

As you follow the download instructions and setup wizard, make sure install the `beer-sample` default bucket. It contains sample data of beers and breweries, which that you will use with the examples here.

If you already have Couchbase Server 2.0 and but do not have the `beer-sample` bucket or you deleted it, open the Couchbase Web Console and navigate to `Settings/Sample Buckets`. Activate the `beer-sample` checkbox and click `Create`. In the right hand corner you will see a notification box that will disappear once the bucket is ready to be used.

Downloading the Couchbase Client Libraries

There are two options to include the Client SDK in your project. You can either manually include all dependencies in your `CLASSPATH` or if you want it to be easier, you can use Maven.

To include the libraries directly in your project, download the archive and add all `jar` files to your `CLASSPATH` of the system/project. Most IDEs also enable you add specific `jar` files to your project. Make sure you add the following dependencies in your `CLASSPATH`:

- couchbase-client-1.1.5.jar, or latest version available

- spymemcached-2.8.12.jar
- commons-codec-1.5.jar
- httpcore-4.1.1.jar
- netty-3.5.5.Final.jar
- httpcore-nio-4.1.1.jar
- jettison-1.1.jar

Alternatively, you can use a build manager to handle them for you. Couchbase provides a Maven repository that you can use which automatically includes the SDK dependencies. The root URL of the repository is located under http://files.couchbase.com/maven2/. Depending on your build manager, the exact syntax you use to include it may vary. Here is an example on how to do it in Maven by updating your `pom.xml`:

```
<repositories>
  <repository>
    <id>couchbase</id>
    <name>Couchbase Maven Repository</name>
    <layout>default</layout>
    <url>http://files.couchbase.com/maven2/</url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>
</repositories>

<dependency>
    <groupId>couchbase</groupId>
    <artifactId>couchbase-client</artifactId>
    <version>1.1.4</version>
    <scope>provided</scope>
</dependency>
```

If you have a background in Scala and want to manage your dependencies through sbt, then you can do it with these additions to your `build.sbt`:

```
resolvers += "Couchbase Maven Repository" at "http://files.couchbase.com/maven2"

libraryDependencies += "couchbase" % "couchbase-client" % "1.1.4"
```

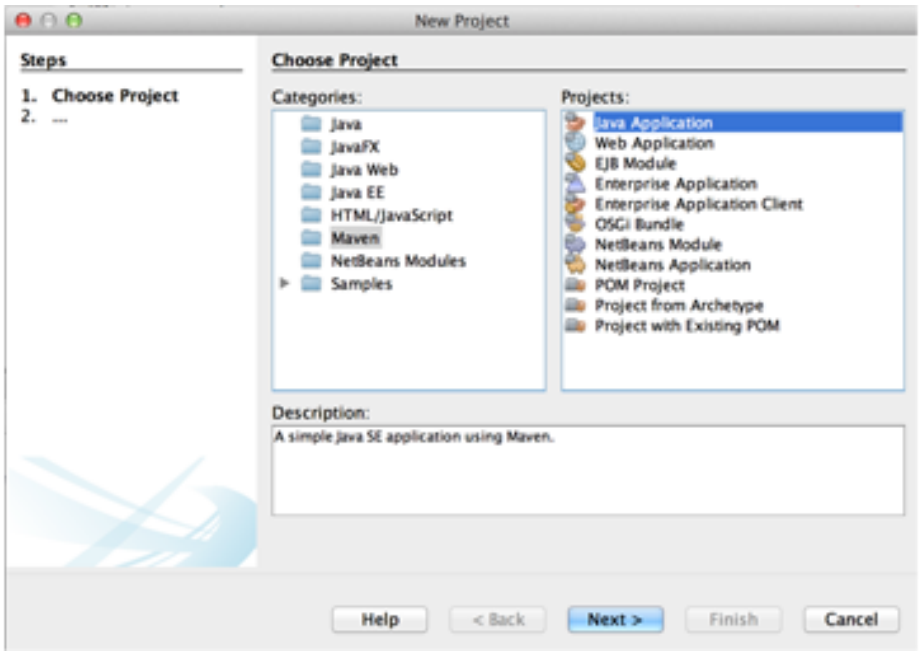
Now that you have the Java SDK third party dependencies set up in your classpath or via a build manager, you can set up your IDE to use the SDK.

Setting up your IDE

In this example we use the NetBeans IDE, but you can use any other Java-compatible IDE with the Java SDK as well. After you install the IDE and open it:

1. Select `File -> New Project` -> `Maven` -> `Java Application`.

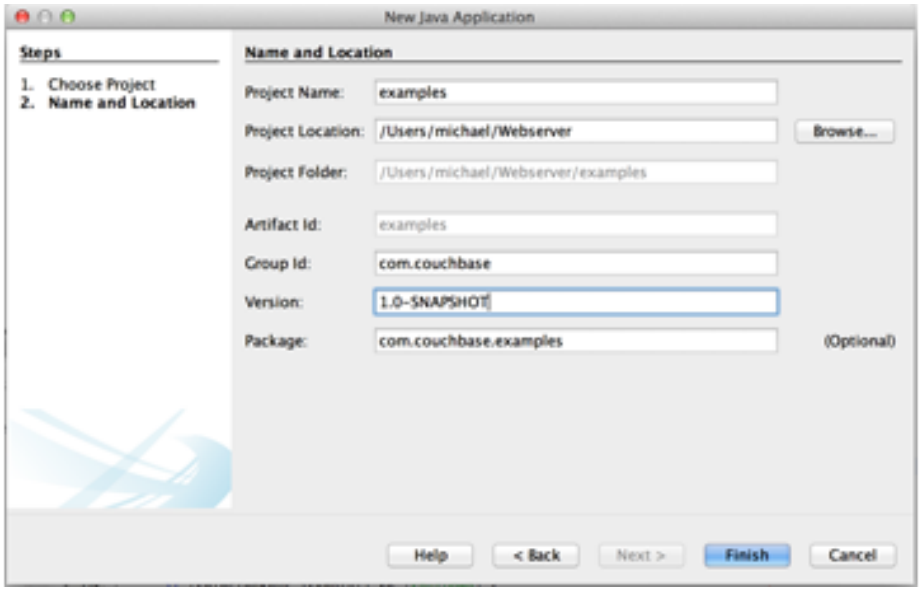
Figure 1.1. Set up Maven Project



2. Provide a name for your new project "examples" and change the location to the directory you want.

3. Provide a name for your new project "examples" and change the location to the directory you want.

Figure 1.2. Set up Maven Project



4. Provide a namespace for the project. We use the `com.couchbase` namespace for this example, but you can use your own if you like. If you do so, just make sure you change the namespace later in the source files when you copy them from our examples.

   Now that your project, you can add the Couchbase Maven repository to use the Java SDK.

5. Select `Window -> Services` to open a list of available services.
6. Under the `Maven Repositories` tree, right click on the Couchbase Java SDK and click `Add Repository`.
7. Use the following settings for the repository:
   - ID: couchbase
   - Name: Couchbase Maven Repository
   - URL: http://files.couchbase.com/maven2/
8. Go back to your new project and right click on `Dependencies`, and then `Add Dependency`. For now, we only need to add the Couchbase SDK itself, because the transitive dependencies will be fetched automatically. Use the following settings:
   - Group ID: couchbase
   - Artifact ID: couchbase-client
   - Version: 1.1.4

Now all the dependencies are in place and we can move forward to our first application with Couchbase.

1.2. Hello Couchbase

To follow the tradition of first programming tutorials, we start with a "Hello Couchbase" example. In this example, we connect to the a Couchbase node, set a simple document, retrieve the document, and then print the value out. This first example contains the full source code, but in later examples we omit the import statements and also assume an existing connection to the cluster.

Listing 1: Hello Couchbase!

```
package com.couchbase.examples;

import com.couchbase.client.CouchbaseClient;
import java.net.URI;
import java.util.ArrayList;

public class App {
  public static void main(String[] args) {
    ArrayList<URI> nodes = new ArrayList<URI>();

    // Add one or more nodes of your cluster (exchange the IP with yours)
    nodes.add(URI.create("http://127.0.0.1:8091/pools"));

    // Try to connect to the client
    CouchbaseClient client = null;
    try {
      client = new CouchbaseClient(nodes, "default", "");
    } catch (Exception e) {
      System.err.println("Error connecting to Couchbase: " + e.getMessage());
      System.exit(1);
    }

    // Set your first document with a key of "hello" and a value of "couchbase!"
    int timeout = 0; // 0 means store forever
    client.set("hello", timeout, "couchbase!");

    // Return the result and cast it to string
    String result = (String)client.get("hello");
    System.out.println(result);

    // Shutdown the client
    client.shutdown();
  }
```

```
}
```

While this code should be very easy to grasp, there is a lot going on worth a little more discussion:

1. **Connect:** the CouchbaseClient accepts a List of `URIs` that point to nodes in the Cluster. You can provide only one `URI` however we strongly recommend that you add two or three if your cluster has more than one node. Be aware that this list does not have to contain all nodes in the cluster; you need to provide a few so that during the initial connection phase your client can connect to the cluster even if one or more nodes fail.

   After initial connection, the Client automatically fetches cluster configuration and keeps it up to date, even when the cluster topology changes. This means that you do not need to change your application configuration at all when you add nodes to your cluster or when nodes fail. Also keep in mind to use a URI in this format: `http://[YOUR-NODE]:8091/pools`. If you only provide the IP address, your client will fail to connect. We call this initial URI the *bootstrap URI*.

   The next two arguments are for the `bucket` and the `password`. The bucket is the container for all your documents. Inside a bucket, a key - the identifier for a document - must be unique. In production environments, it is recommended to use a password on a bucket (this can be configured during bucket creation), but when you are just starting out using the `default` bucket without a password is fine. Note that the `beer-sample` bucket also doesn't have a password, so just change the bucket name and you're set.

2. **`Set` and `get`:** these two operations are the most important ones you will use from a Couchbase SDK. You use `set` to create or overwrite a document and you use `get` to read it from the server. There are lots of arguments and variations for these two methods, but if you use them as shown in the previous example it will get you pretty fair in your application development.

   Note that the `get` operation will read all types of information, including binary, from the server, so you need to cast it into the data format you want. In our case we knew we stored a string, so it makes sense to convert it back to a string when we get it later.

3. **Disconnect** when you shutdown your server instance, such as at the end of your application, you should use the `shutdown` method to prevent loss of data. If you use this method without arguments, it will wait until all outstanding operations complete, but will not accept any new operations. You can also call this method with a maximum waiting time which makes sense if you do not want your application to wait indefinitely for a response from the server.

Be aware that by default, the logger for the Java SDK will log from `INFO` upwards by default. This means the Java SDK will log a good amount of information about server communications. From our Hello Couchbase example the log look likes this:

```
2012-12-03 18:57:45.777 INFO com.couchbase.client.CouchbaseConnection:  Added {QA sa=/127.0.0.1:11210, #Rops=0, #Wops=
2012-12-03 18:57:45.788 INFO com.couchbase.client.CouchbaseConnection:  Connection state changed for sun.nio.ch.Select
2012-12-03 18:57:45.807 INFO com.couchbase.client.ViewConnection:  Added localhost to connect queue
2012-12-03 18:57:45.808 INFO com.couchbase.client.CouchbaseClient:  viewmode property isn't defined. Setting viewmode
couchbase!
2012-12-03 18:57:45.925 INFO com.couchbase.client.CouchbaseConnection:  Shut down Couchbase client
2012-12-03 18:57:45.929 INFO com.couchbase.client.ViewConnection:  Node localhost has no ops in the queue
2012-12-03 18:57:45.929 INFO com.couchbase.client.ViewNode:  I/O reactor terminated for localhost
```

You can determine which nodes the client is connected to, see whether views on the server are in development or production mode, and other helpful output. These logs provide vital information when you need to debug any issues on Couchbase community forums or through Couchbase Customer Support.

1.3. Reading Documents

With Couchbase Server 2.0, you have two ways of fetching your documents: either by the unique key through the `get` method, or through Views. Since Views are more complex we will discuss them later in this guide. In the meantime, we show `get` first:

```
Object get = client.get("mykey");
```

Since Couchbase Server will store all types of datatypes, including binary, you get a `Object` back. If you store JSON documents the actual document will be a String, so you can safely convert it to a string:

```
String json = (String) client.get("mykey");
```

If the server finds no document for that key it will return a `null`. It is important that you check for `null` in your code, to prevent `NullPointerExceptions` later down the stack.

With Couchbase Server 2.0, you can also query for documents with secondary indexes, which we collectively call Views. This feature enables you to provide *map functions* to extract information and you can optionally provide *reduce functions* to perform calculations on information. This guide gets you started on how to use them through the Java SDK, if you want to learn more, including how to set up views with Couchbase Web Console please see Couchbase Server Manual, Views.

This next example assumes you already have a views function set up with Couchbase Web Console. Once you create your View in the Web Console, you can query it from the Java SDK in three steps. First, you get the View definition from the Couchbase cluster, second you create a `Query` object and third, you query the cluster with both the `View` and the `Query` objects. In its simplest form, it looks like this:

```
// 1: Get the View definition from the cluster
View view = client.getView("beer", "brewery_beers");

// 2: Create the query object
Query query = new Query();

// 3: Query the cluster with the view and query information
ViewResponse result = client.query(view, query);
```

The `getView()` method needs both the name of the `Design Document` and the name of the `View` to load the proper definition from the cluster. The SDK needs this to determine if there is a view with the given map functions and also if it contains a reduce function or is even a spatial view.

You can query views with several different options. All options are available as setter methods on the `Query` object. Here are some of them:

- `setIncludeDocs(boolean)`: Use to define if the complete documents should be included in the result.
- `setReduce(boolean)`: Used to enable/disable the reduce function (if there is one defined on the server).
- `setLimit(int)`: Limit the number of results that should be returned.
- `setDescending(boolean)`: Revert the sorting order of the result set.
- `setStale(Stale)`: Can be used to define the tradeoff between performance and freshness of the data.
- `setDebug(boolean)`: Prints out debugging information in the logs.

Now that we have our View information and the Query object in place, we can issue the `query` command, which actually triggers indexing on a Couchbase cluster. The server returns the results to the Java SDK in the `ViewResponse` object. We can use it to iterate over the results and print out some details (here is a more complete example which also includes the full documents and only fetches the first five results):

```
View view = client.getView("beer", "brewery_beers");
Query query = new Query();
query.setIncludeDocs(true).setLimit(5); // include all docs and limit to 5
ViewResponse result = client.query(view, query);

// Iterate over the result and print the key of each document:
for(ViewRow row : result) {
  System.out.println(row.getId());
  // The full document (as String) is available through row.getDocument();
}
```

In the logs, you can see the corresponding document keys automatically sorted in ascending order:

```
21st_amendment_brewery_cafe
21st_amendment_brewery_cafe-21a_ipa
21st_amendment_brewery_cafe-563_stout
21st_amendment_brewery_cafe-amendment_pale_ale
21st_amendment_brewery_cafe-bitter_american
```

1.4. Deleting Documents

If you want to get delete documents, you can use the `delete` operation:

```
OperationFuture<Boolean> delete = client.delete("key");
```

Again, `delete` is an asynchronous operation and therefore returns a `OperationFuture` on which you can block through the `get()` method. If you try to delete a document that is not there, the result of the `OperationFuture` will be `false`. Be aware that when you delete a document, the server does not immediately remove a copy of that document from disk, instead it performs lazy deletion for items that expired or deleted items. For more information about how the server handles lazy expiration, see Couchbase Developer Guide, About Document Expiration.

1.5. Next Steps

You are now ready start exploring Couchbase Server and the Java SDK on your own. If you want to learn more and see a full-fledged application on top of Couchbase Server 2.0, go to to the Web Application Tutorial. Also, the server documentation and the developer documentation provide useful information for your day-to-day work with Couchbase. Finally, the API docs of the Java SDK can be found here. And JavaDoc is also available.

Chapter 2. Tutorial

In this chapter we build on the foundations introduced in the Getting Started section and build a complete web application. Make sure to have the `beer-sample` bucket installed, because the application will allow you to display and manage beers and breweries. If you still need to get the sample database, see Section 1.1, “Preparation”.

The full sourcecode for the example we show you is available at couchbaselabs on GitHub. Note that the sample application you can download actually provides more content than we describe in this tutorial; but it should be easy for you to look around and understand how it functions if you first start reading this tutorial here.

2.1. Preview the Application

If you want to get up and running really quickly, here is how to do it with Jetty. Note that this guide assumes you have MacOS or Linux. If you use Windows, you will need to modify the paths accordingly. Also, make sure to have at least Maven installed on your machine.

1. Download Couchbase Server 2.0 and install it. Make sure you install the beer-sample dataset when you run the wizard, because this tutorial will use it.

2. Add the following views and design documents to the `beer-sample` bucket. Views and design documents enable us to index and query data from our database. Later we will publish the views as production views. For more information about using views from an SDK, see Couchbase Developer Guide, Finding Data with Views.

   The first design document name is `beer` and view name is `by_name`:

   ```
   function (doc, meta) {
     if(doc.type && doc.type == "beer") {
       emit(doc.name, null);
     }
   }
   ```

   The other design document name is `brewery` and view name is `by_name`:

   ```
   function (doc, meta) {
     if(doc.type && doc.type == "brewery") {
       emit(doc.name, null);
     }
   }
   ```

3. Clone the Java SDK beer repository from GitHub and `cd` into the directory:

   ```
   $ git clone git://github.com/couchbaselabs/beersample-java.git
   Cloning into 'beersample-java'...
   remote: Counting objects: 153, done.
   remote: Compressing objects: 100% (92/92), done.
   remote: Total 153 (delta 51), reused 124 (delta 22)
   Receiving objects: 100% (153/153), 81.97 KiB | 120 KiB/s, done.
   Resolving deltas: 100% (51/51), done.
   $ cd beersample-java
   ```

4. In Maven, run the application inside the Jetty container:

   ```
   $ mvn jetty:run
   .... snip ....
   Dec 17, 2012 1:50:16 PM com.couchbase.beersample.ConnectionManager contextInitialized
   INFO: Connecting to Couchbase Cluster
   2012-12-17 13:50:16.621 INFO com.couchbase.client.CouchbaseConnection:  Added {QA sa=/127.0.0.1:11210, #Rops=0, #Wo
   2012-12-17 13:50:16.624 INFO com.couchbase.client.CouchbaseConnection:  Connection state changed for sun.nio.ch.Se
   2012-12-17 13:50:16.635 WARN net.spy.memcached.auth.AuthThreadMonitor:  Incomplete authentication interrupted for r
   2012-12-17 13:50:16.662 WARN net.spy.memcached.auth.AuthThread:  Authentication failed to localhost/127.0.0.1:11210
   2012-12-17 13:50:16.662 INFO net.spy.memcached.protocol.binary.BinaryMemcachedNodeImpl:  Removing cancelled operati
   2012-12-17 13:50:16.664 INFO net.spy.memcached.auth.AuthThread:  Authenticated to localhost/127.0.0.1:11210
   2012-12-17 13:50:16.666 INFO com.couchbase.client.ViewConnection:  Added localhost to connect queue
   2012-12-17 13:50:16.667 INFO com.couchbase.client.CouchbaseClient:  viewmode property isn't defined. Setting viewmo
   2012-12-17 13:50:16.866:INFO::Started SelectChannelConnector@0.0.0.0:8080
   ```

```
[INFO] Started Jetty Server
```

5. Now, navigate to http://localhost:8080/welcome and enjoy the application.

2.2. Preparing Your Project

This tutorial uses Servlets and JSPs in combination with Couchbase Server 2.0 to display and manage beers and breweries found in the `beer-sample` dataset. The easiest way is to develop it with your IDE such as Eclipse or NetBeans. Then you can use your IDE to publish it automatically to an application server such as Apache Tomcat or GlassFish as a `war` archive. We designed the code here to be as portable as possible, but it may be the case that you need to change one or two things if you have a slightly different version or a customized setup in your environment.

2.2.1. Project Setup

In your IDE, create a new `Web Project`, either with or without Maven support. If you have not already gone through the Getting Started for the Java SDK, you should review the information on how to include the Couchbase SDK and all the required dependencies in your project. For more information, see Section 1.1, “Preparation”.

Also make sure to include Google GSON or your favorite JSON library as well.

This tutorial uses a directory structure that appears as follows:

```
|-target
|-src
|---main
|-----java
|-------com
|---------couchbase
|-----------beersample
|-----resources
|-----webapp
|-------WEB-INF
|---------beers
|---------breweries
|---------maps
|---------tags
|---------welcome
|-------css
|-------js
```

If you use Maven, you should also have a `pom.xml` in the root directory. Here is a sample `pom.xml` so you can see the general structure and dependencies. The full source is at the repository we mentioned earlier. See couchbaselabs on GitHub for the full `pom.xml`:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.couchbase</groupId>
    <artifactId>beersample-java</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>

    <name>beersample-java</name>

    <repositories>
        <repository>
            <id>couchbase</id>
            <name>Couchbase Maven Repository</name>
            <layout>default</layout>
            <url>http://files.couchbase.com/maven2/</url>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </repository>
    </repositories>

    <dependencies>
```

```
        <dependency>
            <groupId>couchbase</groupId>
            <artifactId>couchbase-client</artifactId>
            <version>1.1.4</version>
        </dependency>
        <dependency>
            <groupId>com.google.code.gson</groupId>
            <artifactId>gson</artifactId>
            <version>2.2.2</version>
        </dependency>
        <dependency>
            <groupId>javax</groupId>
            <artifactId>javaee-web-api</artifactId>
            <version>6.0</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
</project>
```

To make the application more interactive, we use jQuery and Twitter Bootstrap. You can either download the libraries and put them in their appropriate css and js directories under `webapp`, or clone the project repository and use it from there. Either way, make sure you have the following files in place:

- css/beersample.css
- css/bootstrap.min.css (the minified twitter bootstrap library)
- css/bootstrap-responsive.min.css (the minified responsive layout classes from bootstrap)
- js/beersample.js
- js/jquery.min.js (the jQuery javascript library)

From here, you should have a basic web application configured that has all the dependencies included. We now move on and configure the `beer-sample` bucket so we can use it in our application.

2.2.2. Creating Your Views

Views enable you to index and query data from your database. The `beer-sample` bucket comes with a small set of predefined view functions, but to have our application function correctly we need some more views. This is also a very good chance for you to see how you can manage views inside Couchbase Web Console. For more information on the topics, see Couchbase Developer Guide, Finding Data with Views and Couchbase Manual, Using the Views Editor.

Since we want to list beers and breweries by their name, we need to define one view function for each type of result that we want.

1. In Couchbase Web Console and click on the `Views` menu.
2. Select `beer-sample` from the dropdown list to switch to the correct bucket.
3. Now click on `Development Views` and then `Create Development View` to define your first view.
4. You need to give it the name of both the design document and the actual view. Insert the following names:

   Design Document Name: _design/dev_beer

   View Name: by_name

   The next step is to define the `map` and function and optionally at this phase you could define a `reduce` function to perform information on the index results. In our example, we do not use the `reduce` functions at all but you can play around with reduce functions and see how they work. See, Couchbase Developer Guide, Using Built-in Reduce Functions and Creating Custom Reduces.

5. Insert the following `map` function (that's JavaScript) and click `Save`.

```
function (doc, meta) {
  if(doc.type && doc.type == "beer") {
    emit(doc.name, null);
  }
}
```

Every `map` function takes the full document (`doc`) and its associated metadata (`meta`) as the arguments. Your map function can then inspect this data and `emit` the item to a result set when you want to have it in your index. In our case we emit the name of the beer (`doc.name`) when the document has a type field and the type is `beer`. For our application we do not need to emit a value; therefore we emit a `null` here.

In general you should try to keep the index as small as possible. You should resist the urge to include the full document with `emit(meta.id, doc)`, because it will increase the size of your view indexes and potentially impact application performance. If you need to access the full document or large parts of it, use the `setIncludeDocs(true)` directive which will do a `get()` call with the document ID in the background. Couchbase Server may return a version of the document that may be slightly out of sync with your view, but it will be a fast and efficient operation.

Now we need to provide a similar map function for our breweries. Since you already know how to do this, here is all the information you need to create it:

- Design Document Name: _design/dev_brewery
- View Name: by_name
- Map Function:

```
function (doc, meta) {
  if(doc.type && doc.type == "brewery") {
    emit(doc.name, null);
  }
}
```

The final step that you need to do is to push the design documents in production mode for Couchbase Server. While the design documents are in `development`, the index is only applied on the local node. See, Couchbase Manual, Development and Production Views. Since we want to have the index on the whole dataset:

1. In Couchbase Web Console, click the Views Tab.
2. click the `Publish` button on both design documents.
3. Accept any dialog that warns you from overriding the old view function.

For more information about using views for indexing and querying from Couchbase Server, here are some useful resources:

- General Information: Couchbase Server Manual: Views and Indexes.
- Sample Patterns: to see examples and patterns you can use for views, see Couchbase Views, Sample Patterns.
- Timestamp Pattern: many developers frequently ask about extracting information based on date or time. To find out more, see Couchbase Views, Sample Patterns.

2.2.3. Bootstrapping Our Servlets

To tell the application server where and how the incoming HTTP requests should be routed, we need to define a `web.xml` inside the `WEB-INF` directory of our project:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
    <listener>
        <listener-class>com.couchbase.beersample.ConnectionManager</listener-class>
    </listener>
    <servlet>
        <servlet-name>WelcomeServlet</servlet-name>
        <servlet-class>com.couchbase.beersample.WelcomeServlet</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>BreweryServlet</servlet-name>
        <servlet-class>com.couchbase.beersample.BreweryServlet</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>BeerServlet</servlet-name>
        <servlet-class>com.couchbase.beersample.BeerServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>WelcomeServlet</servlet-name>
        <url-pattern>/welcome</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>BreweryServlet</servlet-name>
        <url-pattern>/breweries/*</url-pattern>
        <url-pattern>/breweries</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>BeerServlet</servlet-name>
        <url-pattern>/beers/*</url-pattern>
        <url-pattern>/beers</url-pattern>
    </servlet-mapping>
  <welcome-file-list>
    <welcome-file>welcome</welcome-file>
  </welcome-file-list>
</web-app>
```

This is not ready to run yet, because you have not implemented any of these classes yet, but we will do that soon. The `listener` directive references the `ConnectionMananger` class, which we implement to manage the connection instance to our Couchbase cluster. The `servlet` directives define our servlet classes that we use and the following `servlet-mapping` directives map HTTP URLs to them. The final `welcome-file-list` directive tells the application server where to route the root URL (`"/"`).

For now, comment out all `servlet`, `servlet-mapping` and `welcome-file-list` directives with the `<!--` and `-->` characters, because the application server will complain that they are not implemented. When you implement the appropriate servlets, remove the comments accordingly. If you plan to add your own servlets, remember to add and map them inside the `web.xml` properly!

2.3. Managing Connections

The first class we implement is the `ConnectionManager` in the `src/main/java/com/couchbase/beersample` directory. This is a `ServletContextListener` which starts the `CouchbaseClient` on application startup and closes the connection when the application shuts down. Here is the full class:.

```
package com.couchbase.beersample;

public class ConnectionManager implements ServletContextListener {

  private static CouchbaseClient client;

  private static final Logger logger = Logger.getLogger(
    ConnectionManager.class.getName());

  @Override
  public void contextInitialized(ServletContextEvent sce) {
    logger.log(Level.INFO, "Connecting to Couchbase Cluster");
    ArrayList<URI> nodes = new ArrayList<URI>();
    nodes.add(URI.create("http://127.0.0.1:8091/pools"));
    try {
      client = new CouchbaseClient(nodes, "beer-sample", "");
    } catch (IOException ex) {
      logger.log(Level.SEVERE, ex.getMessage());
    }
```

```
  }

  @Override
  public void contextDestroyed(ServletContextEvent sce) {
    logger.log(Level.INFO, "Disconnecting from Couchbase Cluster");
    client.shutdown();
  }

  public static CouchbaseClient getInstance() {
    return client;
  }

}
```

Note in this example, we removed the comments and imports have been removed to shorten the listing a bit. The `contextInitialized` and the `contextDestroyed` method are called on startup and shutdown. When the application starts, we initialize the `CouchbaseClient` with the list of nodes, the bucket name and an empty password. Note that in a production deployment, you want to fetch these environment-dependent settings from a config file. We will call the `getInstance()` method from the servlets to obtain the `CouchbaseClient` instance.

When you publish your application, you should see in the server logs that the Java SDK correctly connects to the bucket. If you see an exception at this phase, it means that your settings are wrong or you have no Couchbase Server running at the given nodes. Here is an example server log during successfull connection:

```
INFO: Connecting to Couchbase Cluster
SEVERE: 2012-12-05 14:39:00.419 INFO com.couchbase.client.CouchbaseConnection:  Added {QA sa=/127.0.0.1:11210, #Rops=0
SEVERE: 2012-12-05 14:39:00.426 INFO com.couchbase.client.CouchbaseConnection:  Connection state changed for sun.nio.c
SEVERE: 2012-12-05 14:39:00.458 INFO net.spy.memcached.auth.AuthThread:  Authenticated to localhost/127.0.0.1:11210
SEVERE: 2012-12-05 14:39:00.487 INFO com.couchbase.client.ViewConnection:  Added localhost to connect queue
SEVERE: 2012-12-05 14:39:00.489 INFO com.couchbase.client.CouchbaseClient:  viewmode property isn't defined. Setting v
INFO: WEB0671: Loading application [com.couchbase_beersample-java_war_1.0-SNAPSHOT] at [/]
INFO: com.couchbase_beersample-java_war_1.0-SNAPSHOT was successfully deployed in 760 milliseconds.
```

2.4. The Welcome Page

The first servlet that we implement is the `WelcomeServlet`, so go ahead and remove the appropriate comments inside the `web.xml` file. You also want to enable the `welcome-file-list` as well at this point. When a user visits the application, we show him a nice greeting and give him all available options to choose.

Since there is no Couchbase Server interaction involved, we just tell it to render the JSP template:

```
package com.couchbase.beersample;

public class WelcomeServlet extends HttpServlet {

  @Override
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    request.getRequestDispatcher("/WEB-INF/welcome/index.jsp")
      .forward(request, response);
  }

}
```

The index.jsp uses styling from Twitter bootstrap to look provide clean layout. Aside from that, it shows a nice greeting and links to the servlets that provide the actual functionality:

```
<%@taglib prefix="t" tagdir="/WEB-INF/tags" %>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<t:layout>
    <jsp:body>
        <div class="span6">
          <div class="span12">
            <h4>Browse all Beers</h4>
            <a href="/beers" class="btn btn-warning">Show me all beers</a>
            <hr />
          </div>
```

```
          <div class="span12">
            <h4>Browse all Breweries</h4>
            <a href="/breweries" class="btn btn-info">Take me to the breweries</a>
          </div>
        </div>
        <div class="span6">
          <div class="span12">
            <h4>About this App</h4>
            <p>Welcome to Couchbase!</p>
            <p>This application helps you to get started on application
                development with Couchbase. It shows how to create, update and
                delete documents and how to work with JSON documents.</p>
            <p>The official tutorial can be found
                <a href="http://www.couchbase.com/docs/couchbase-sdk-java-1.1/tutorial.html">here</a>!</p>
          </div>
        </div>
    </jsp:body>
</t:layout>
```

There is one more interesting note to make here: it uses taglibs, which enables us to use the same layout for all pages. Since we have not created this layout, we do so now. Create a `layout.tag` file in the `/WEB-INF/tags` directory that looks like this:

```
<%@tag description="Page Layout" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Couchbase Java Beer-Sample</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta name="description" content="The Couchbase Java Beer-Sample App">
    <meta name="author" content="Couchbase, Inc. 2012">

    <link href="/css/bootstrap.min.css" rel="stylesheet">
    <link href="/css/beersample.css" rel="stylesheet">
    <link href="/css/bootstrap-responsive.min.css" rel="stylesheet">

    <!-- HTML5 shim, for IE6-8 support of HTML5 elements -->
    <!--[if lt IE 9]>
      <script src="http://html5shim.googlecode.com/svn/trunk/html5.js"></script>
    <![endif]-->
  </head>
  <body>
    <div class="container-narrow">
      <div class="masthead">
        <ul class="nav nav-pills pull-right">
          <li><a href="/welcome">Home</a></li>
          <li><a href="/beers">Beers</a></li>
          <li><a href="/breweries">Breweries</a></li>
        </ul>
        <h2 class="muted">Couchbase Beer-Sample</h2>
      </div>
      <hr>
      <div class="row-fluid">
        <div class="span12">
            <jsp:doBody/>
        </div>
      </div>
      <hr>
      <div class="footer">
        <p>&copy; Couchbase, Inc. 2012</p>
      </div>
    </div>
    <script src="/js/jquery.min.js"></script>
    <script src="/js/bootstrap.min.js"></script>
    <script src="/js/beersample.js"></script>
  </body>
</html>
```

Again, nothing fancy here. We just need it in place to make everything look clean afterwards. When you deploy your application, you should see in the logs that it is connects to the Couchbase cluster and when you view it in the browser you see a nice web page greeting.

2.5. Managing Beers

Now we reach the main portion of the tutorial where we actually interact with Couchbase Server. First, we uncomment the `BeerServlet` and its corresponding tags inside the `web.xml`. We make use of tje view to list all beers and make them easily searchable. We also provide a form to create and/or edit beers and finally delete them.

Here is the bare structure of our `BeerServlet`, which will be filled with live data soon. Once again, we removed comments and imports for the sake of brevity:

```
package com.couchbase.beersample;

public class BeerServlet extends HttpServlet {

  final CouchbaseClient client = ConnectionManager.getInstance();

  final Gson gson = new Gson();

  @Override
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    try {
      if(request.getPathInfo() == null) {
        handleIndex(request, response);
      } else if(request.getPathInfo().startsWith("/show")) {
        handleShow(request, response);
      } else if(request.getPathInfo().startsWith("/delete")) {
        handleDelete(request, response);
      } else if(request.getPathInfo().startsWith("/edit")) {
        handleEdit(request, response);
      } else if(request.getPathInfo().startsWith("/search")) {
        handleSearch(request, response);
      }
    } catch (InterruptedException ex) {
      Logger.getLogger(BeerServlet.class.getName()).log(
        Level.SEVERE, null, ex);
    } catch (ExecutionException ex) {
      Logger.getLogger(BeerServlet.class.getName()).log(
        Level.SEVERE, null, ex);
    }
  }

  @Override
  protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
  }

  private void handleIndex(HttpServletRequest request,
    HttpServletResponse response) throws IOException, ServletException {
  }

  private void handleShow(HttpServletRequest request,
    HttpServletResponse response) throws IOException, ServletException {
  }

  private void handleDelete(HttpServletRequest request,
    HttpServletResponse response) throws IOException, ServletException,
    InterruptedException,
    ExecutionException {
  }

  private void handleEdit(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
  }

  private void handleSearch(HttpServletRequest request,
    HttpServletResponse response) throws IOException, ServletException {
  }

}
```

Since our `web.xml` uses wildcards (`*`) to route every `/beer` that is related to this servlet, we need to inspect the path through `getPathInfo()` and dispatch the request to a helper method that does the actual work. We use the `doPost()`

method to analyze and store the results of the web-form. We also use this method to edit and create new beers since we sent the form through a POST request.

The first functionality we implement is a list of the top 20 beers in a table. We can use the `beer/by_name` view we created at earlier to get a sorted list of all beers. The following code belongs to the `handleIndex` method and will build the list:

```
// Fetch the View
View view = client.getView("beer", "by_name");

// Set up the Query object
Query query = new Query();

// We the full documents and only the top 20
query.setIncludeDocs(true).setLimit(20);

// Query the Cluster
ViewResponse result = client.query(view, query);

// This ArrayList will contain all found beers
ArrayList<HashMap<String, String>> beers = new ArrayList<HashMap<String, String>>();

// Iterate over the found documents
for(ViewRow row : result) {
  // Use Google GSON to parse the JSON into a HashMap
  HashMap<String, String> parsedDoc = gson.fromJson((String)row.getDocument(), HashMap.class);

  // Create a HashMap which will be stored in the beers list.
  HashMap<String, String> beer = new HashMap<String, String>();
  beer.put("id", row.getId());
  beer.put("name", parsedDoc.get("name"));
  beer.put("brewery", parsedDoc.get("brewery_id"));
  beers.add(beer);
}

// Pass all found beers to the JSP layer
request.setAttribute("beers", beers);

// Render the index.jsp template
request.getRequestDispatcher("/WEB-INF/beers/index.jsp")
  .forward(request, response);
```

The index action in the code above queries the view, parses the results with GSON into a `HashMap` and eventually forwards the `ArrayList` to the JSP layer. At this point we can implement the `index.jsp` template which will iterate over the `ArrayList` and print the beers out in a nicely-formatted table:

```
<%@taglib prefix="t" tagdir="/WEB-INF/tags" %>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<t:layout>
  <jsp:body>
    <h3>Browse Beers</h3>

    <form class="navbar-search pull-left">
      <input id="beer-search" type="text" class="search-query" placeholder="Search for Beers">
    </form>

    <table id="beer-table" class="table table-striped">
      <thead>
        <tr>
          <th>Name</th>
          <th>Brewery</th>
          <th></th>
        </tr>
      </thead>
      <tbody>
        <c:forEach items="${beers}" var="beer">
            <tr>
              <td><a href="/beers/show/${beer.id}">${beer.name}</a></td>
              <td><a href="/breweries/show/${beer.brewery}">To Brewery</a></td>
              <td>
                <a class="btn btn-small btn-warning" href="/beers/edit/${beer.id}">Edit</a>
                <a class="btn btn-small btn-danger" href="/beers/delete/${beer.id}">Delete</a>
              </td>
```

```
            </tr>
          </c:forEach>
        </tbody>
      </table>
    </jsp:body>
</t:layout>
```

Here we useJSP tags to iterate over the beers and use their properties,`name` and `id`, and fill the rows in the table with this information. In a browser you should now see a table with a list of beers with `Edit` and `Delete` buttons on the right. You can also see a link to the associated brewery that you can click on. Now we implement the delete action for each beer, since its very easy to do with Couchbase:

```
private void handleDelete(HttpServletRequest request,
  HttpServletResponse response) throws IOException, ServletException, InterruptedException, ExecutionException {

  // Split the Request-Path and get the Beer ID out of it
  String beerId = request.getPathInfo().split("/")[2];

  // Try to delete the document and store the OperationFuture
  OperationFuture<Boolean> delete = client.delete(beerId);

  // If the Future succeeded (returned true), redirect to /beers
  if(delete.get()) {
    response.sendRedirect("/beers");
  }
}
```

The delete method deletes a document from the cluster based on the given document key. Here, we wait on the `OperationFuture` to return from the `get()` method and if the server successfully deletes the item we get `true` and can redirect to the index action.

Now that we can delete a document, we want to enable users to also edits beerst. The edit action is very similar to the delete action, but it reads and updates the document based on the given `ID` instead of deleting it. Before we can edit a beer, we also need to parse the String representation of the JSON document into a Java structure, so we can use it in the template. We again make use of the excellent Google GSON library to handle this for us:

```
private void handleEdit(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {

    // Extract the Beer ID from the URL
    String[] beerId = request.getPathInfo().split("/");

    // If there is a Beer ID
    if(beerId.length > 2) {

      // Read the Document (as a JSON string)
      String document = (String) client.get(beerId[2]);

      HashMap<String, String> beer = null;
      if(document != null) {
        // Convert the String into a HashMap
        beer = gson.fromJson(document, HashMap.class);
        beer.put("id", beerId[2]);

        // Forward the beer to the view
        request.setAttribute("beer", beer);
      }
      request.setAttribute("title", "Modify Beer \"" + beer.get("name") + "\"");
    } else {
      request.setAttribute("title", "Create a new beer");
    }

    request.getRequestDispatcher("/WEB-INF/beers/edit.jsp").forward(request, response);
  }
```

If we get this beer document back from Couchbase Server and parse it into JSON, we convert it into a `HashMap` and then forwarded to the edit.jsp template. Also, we define a title variable that we use inside the template to determine if we want to edit a document or create a new one. We can enables users to create new beers as opposed to editing an existing beer anytime we pass no Beer ID to the edit method. Here is the corresponding edit.jsp template:

```
<%@taglib prefix="t" tagdir="/WEB-INF/tags" %>
```

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<t:layout>
  <jsp:body>
    <h3>${title}</h3>

    <form method="post" action="/beers/edit/${beer.id}">
      <fieldset>
        <legend>General Info</legend>
        <div class="span12">
          <div class="span6">
            <label>Name</label>
            <input type="text" name="beer_name" placeholder="The name of the beer." value="${beer.name}">

            <label>Description</label>
            <input type="text" name="beer_description" placeholder="A short description." value="${beer.description}">
          </div>
          <div class="span6">
            <label>Style</label>
            <input type="text" name="beer_style" placeholder="Bitter? Sweet? Hoppy?" value="${beer.style}">

            <label>Category</label>
            <input type="text" name="beer_category" placeholder="Ale? Stout? Lager?" value="${beer.category}">
          </div>
        </div>
      </fieldset>
      <fieldset>
        <legend>Details</legend>
        <div class="span12">
          <div class="span6">
            <label>Alcohol (ABV)</label>
            <input type="text" name="beer_abv" placeholder="The beer's ABV" value="${beer.abv}">

            <label>Biterness (IBU)</label>
            <input type="text" name="beer_ibu" placeholder="The beer's IBU" value="${beer.ibu}">
          </div>
          <div class="span6">
            <label>Beer Color (SRM)</label>
            <input type="text" name="beer_srm" placeholder="The beer's SRM" value="${beer.srm}">

            <label>Universal Product Code (UPC)</label>
            <input type="text" name="beer_upc" placeholder="The beer's UPC" value="${beer.upc}">
          </div>
        </div>
      </fieldset>
      <fieldset>
        <legend>Brewery</legend>
        <div class="span12">
          <div class="span6">
            <label>Brewery</label>
            <input type="text" name="beer_brewery_id" placeholder="The brewery" value="${beer.brewery_id}">
          </div>
        </div>
      </fieldset>
      <div class="form-actions">
          <button type="submit" class="btn btn-primary">Save changes</button>
      </div>
    </form>
  </jsp:body>
</t:layout>
```

This template is a little bit longer, but this is mainly because we have lots of fields on our beer documents. Note how we use the beer attributes inside the value attributes of the HTML input fields. We also use the unique ID in the form method to dispatch it to the correct URL on submit.

The last thing we need to do for form submission to work is the actual form parsing and storing itself. Since we do form submission through a POST request, we need to implement the `doPost()` method on our servlet:

```
@Override
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

  // Parse the Beer ID
  String beerId = request.getPathInfo().split("/")[2];
  HashMap<String, String> beer = beer = new HashMap<String, String>();
  Enumeration<String> params = request.getParameterNames();
```

```
  // Iterate over all POST params
  while(params.hasMoreElements()) {
    String key = params.nextElement();
    if(!key.startsWith("beer_")) {
      continue;
    }
    String value = request.getParameter(key);

    // Store them in a HashMap with key and value
    beer.put(key.substring(5), value);
  }

  // Add two more fields
  beer.put("type", "beer");
  beer.put("updated", new Date().toString());

  // Set (add or override) the document (converted to JSON with GSON)
  client.set(beerId, 0, gson.toJson(beer));

  // Redirect to the show page
  response.sendRedirect("/beers/show/" + beerId);
}
```

The code iterates over all POST fields and stores them in a `HashMap`. We then use the set command to store the document to Couchbase Server and use Google GSON to translate information out of `HashMap` into a JSON string. In this case, we could also wait for a `OperationFuture` response and return an error if we determine the set failed.

The last line redirects to a show method, which just shows all fields of the document. Since the patterns are the same as before, here is the show method:

```
private void handleShow(HttpServletRequest request,
  HttpServletResponse response) throws IOException, ServletException {

  // Extract the Beer ID
  String beerId = request.getPathInfo().split("/")[2];
  String document = (String) client.get(beerId);
  if(document != null) {
    // Parse the JSON and set it for the template if a document was found
    HashMap<String, String> beer = gson.fromJson(document, HashMap.class);
    request.setAttribute("beer", beer);
  }

  // render the show.jsp template
  request.getRequestDispatcher("/WEB-INF/beers/show.jsp")
    .forward(request, response);
}
```

Again we extract the ID and if Couchbase Server finds the document it gets parsed into a HashMap and forwarded to the show.jsp template. If the server finds no document, we get a return of null in the Java SDK. The template then just prints out all keys and values in a table:

```
<%@taglib prefix="t" tagdir="/WEB-INF/tags" %>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<t:layout>
  <jsp:body>
    <h3>Show Details for Beer "${beer.name}"</h3>
    <table class="table table-striped">
      <tbody>
        <c:forEach items="${beer}" var="item">
          <tr>
              <td><strong>${item.key}</strong></td>
              <td>${item.value}</td>
          </tr>
        </c:forEach>
      </tbody>
    </table>
  </jsp:body>
</t:layout>
```

In the index.jsp template, you may notice the search box at the top. We can use it to dynamically filter our table results based on the user input. We will use nearly the same code for the filter as in the index method; except this time we make

use of range queries to define a beginning and end to search for. For more information about performing range queries, see Ordering.

Before we implement the actual Java method, we need to put the following snippet in the `js/beersample.js` file. You may have already done this at the beginning of the tutorial, and if so skip this step. This code takes any searchbox changes from the UI and updates the table with the JSON returned from the search method:

```
$("#beer-search").keyup(function() {
   var content = $("#beer-search").val();
   if(content.length >= 0) {
       $.getJSON("/beers/search", {"value": content}, function(data) {
           $("#beer-table tbody tr").remove();
           for(var i=0;i<data.length;i++) {
               var html = "<tr>";
               html += "<td><a href=\"/beers/show/"+data[i].id+"\">"+data[i].name+"</a></td>";
               html += "<td><a href=\"/breweries/show/"+data[i].brewery+"\">To Brewery</a></td>";
               html += "<td>";
               html += "<a class=\"btn btn-small btn-warning\" href=\"/beers/edit/"+data[i].id+"\">Edit</a>\n";
               html += "<a class=\"btn btn-small btn-danger\" href=\"/beers/delete/"+data[i].id+"\">Delete</a>";
               html += "</td>";
               html += "</tr>";
               $("#beer-table tbody").append(html);
           }
       });
   }
});
```

The code waits for keyup events on the search field and then does a AJAX query to the search method on the servlet. The servlet computes the result and sends it back as JSON. The JavaScript then clears the table, iterates over the result and creates new rows with the new JSON results. The search method looks like this:

```
private void handleSearch(HttpServletRequest request, HttpServletResponse response) throws IOException, ServletExcept

  // Exctract the searched value
  String startKey = request.getParameter("value").toLowerCase();

  // Prepare a query against the by_name view
  View view = client.getView("beer", "by_name");
  Query query = new Query();

  // Define the query params
  query.setIncludeDocs(true) // include the full documents
    .setLimit(20) // only show 20 results
    .setRangeStart(ComplexKey.of(startKey)) // Start the search at the given search value
    .setRangeEnd(ComplexKey.of(startKey + "\uefff")); // End the search at the given search plus the unicode "end"

  // Query the view
  ViewResponse result = client.query(view, query);

  ArrayList<HashMap<String, String>> beers = new ArrayList<HashMap<String, String>>();
  // Iterate over the results
  for(ViewRow row : result) {
    // Parse the Document to a HashMap
    HashMap<String, String> parsedDoc = gson.fromJson((String)row.getDocument(), HashMap.class);

      // Create a new Beer out of it
      HashMap<String, String> beer = new HashMap<String, String>();
      beer.put("id", row.getId());
      beer.put("name", parsedDoc.get("name"));
      beer.put("brewery", parsedDoc.get("brewery_id"));
      beers.add(beer);
  }

  // Return a JSON representation of all Beers
  response.setContentType("application/json");
  PrintWriter out = response.getWriter();
  out.print(gson.toJson(beers));
  out.flush();
}
```

You can use the `setRangeStart()` and `setRangeEnd()` methods to define the key range Couchbase Server returns. If we just provide the start range key, then we get all documents starting from our search value. Since we want only

those beginning with the search value, we can use the special `"\uefff"` UTF-8 character at the end which means "end here". You will need to get used to it this convention, but its very fast and efficient when accessing the view.

2.6. Wrapping Up

The tutorial presents an easy approach to start a web application with Couchbase Server 2.0 as the underlying data source. If you want to dig a little bit deeper, see the full source code on couchbaselabs on GitHub. This contains more servlets and code to learn from. This may be extended and updated from time to time so you may want to watch the repo.

Of course this is only the starting point for Couchbase, but together with the Getting Started Guide and other community resources you are well equipped to start exploring Couchbase Server on your own. Have fun working with Couchbase!

Chapter 3. Using the APIs

The Client libraries provides an interface to both Couchbase and Memcached clients using a consistent interface. The interface between your Java application and your Couchbase or Memcached servers is provided through the instantiation of a single object class, `CouchbaseClient`.

Creating a new object based on this class opens the connection to each configured server and handles all the communication with the server(s) when setting, retrieving and updating values. A number of different methods are available for creating the object specifying the connection address and methods.

3.1. Connecting to a Couchbase Bucket

You can connect to specific Couchbase buckets (in place of using the default bucket, or a hostname/port combination configured on the Couchbase cluster) by using the Couchbase `URI` for one or more Couchbase nodes, and specifying the bucket name and password (if required) when creating the new `CouchbaseClient` object.

For example, to connect to the local host and the `default` bucket:

```
List<URI> uris = new LinkedList<URI>();

    uris.add(URI.create("http://127.0.0.1:8091/pools"));
    try {
      client = new CouchbaseClient(uris, "default", "");
    } catch (Exception e) {
      System.err.println("Error connecting to Couchbase: " + e.getMessage());
      System.exit(0);
    }
```

The format of this constructor is:

```
CouchbaseClient(URIs,BUCKETNAME,BUCKETPASSWORD)
```

Where:

- `URIS` is a `List` of URIs to the Couchbase nodes. The format of the URI is the hostname, port and path `/pools`.
- `BUCKETNAME` is the name of the bucket on the cluster that you want to use. Specified as a `String`.
- `BUCKETPASSWORD` is the password for this bucket. Specified as a `String`.

The returned `CouchbaseClient` object can be used as with any other `CouchbaseClient` object.

3.2. Connecting using Hostname and Port with SASL

If you want to use SASL to provide secure connectivity to your Couchbase server then you could create a `CouchbaseConnectionFactory` that defines the SASL connection type, userbucket and password.

The connection to Couchbase uses the underlying protocol for SASL. This is similar to the earlier example except that we use the `CouchbaseConnectionFactory`.

```
List<URI> baseURIs = new ArrayList<URI>();
        baseURIs.add(base);
        CouchbaseConnectionFactory cf = new
                CouchbaseConnectionFactory(baseURIs,
                    "userbucket", "password");

        client = new CouchbaseClient((CouchbaseConnectionFactory) cf);
```

3.3. Setting runtime Parameters for the CouchbaseConnectionFactory-Builder

A final approach to creating the connection is using the `CouchbaseConnectionFactoryBuilder` and `CouchbaseConnectionFactory` classes.

It's possible to ovverride some of the default paramters that are defined in the `CouchbaseConnectionFactoryBuilder` for a variety of reasons and customize the connection for the session depending on expected load on the server and potential network traffic.

For example, in the following program snippet, we instatiate a new `CouchbaseConnectionFactoryBuilder` and use the `setOpTimeout` method to change the default value to 10000ms (or 10 secs).

We subsequently use the `buildCouchbaseConnection` specifying the bucket name, password and an username (which is not being used any more) to get a `CouchbaseConnectionFactory` object. We then create a `CouchbaseClient` object.

```
List<URI> baseURIs = new ArrayList<URI>();
        baseURIs.add(base);
        CouchbaseConnectionFactoryBuilder cfb = new
            CouchbaseConnectionFactoryBuilder();

        // Ovveride default values on CouchbaseConnectionFactoryBuilder

        // For example - wait up to 10 seconds for an operation to succeed
        cfb.setOpTimeout(10000);

        CouchbaseConnectionFactory cf =
            cfb.buildCouchbaseConnection(baseURIs, "default", "", "");

        client = new CouchbaseClient((CouchbaseConnectionFactory) cf);
```

For example, the following code snippet will set the OpTimeOut value to 10000 secs. before creating the connection as we saw in the code above.

```
cfb.setOpTimeout(10000);
```

These parameters can be set at runtime by setting a property on the command line (such as *-DopTimeout=1000*) or via properties in a file *cbclient.properties* in that order of precedence.

The following parameters can be set as summarized in the table below. We provide the parameter name, a brief description, the default value and why the particular parameter might need to be modified.

Table 3.1. Parameters that can be set via CouchbaseConnectionFactoryBuilder

Parameter	Description	Default value	When to Override the default value
`opTimeout`	Time in millisecs for an operation to Timeout	2500 millisecs.	You can set this value higher when there is heavy network traffic and timeouts happen frequently.
`timeoutExceptionThreshold`	Number of operations to timeout before the node is deemed down	998	You can set this value lower to deem a node is down earlier.
`readBufSize`	Read Buffer Size	16384	You can set this value higher or lower to optimize the reads.

Parameter	Description	Default value	When to Override the default value
`opQueueMaxBlockTime`	The maximum time to block waiting for op queue operations to complete, in milliseconds.	10000 millisecs.	The default has been set with the expectation that most requests are interactive and waiting for more than a few seconds is thus more undesirable than failing the request. However, this value could be lowered for operations not to block for this time.
`shouldOptimize`	Optimize behavior for the network	False	You can set this value to be true if the performance should be optimized for the network as in cases where there are some known issues with the network that may be causing adverse effects on applications.
`maxReconnectDelay`	Maximum number of milliseconds to wait between reconnect attempts.	30000 millisecs.	You can set this value lower when there is intermittent and frequent connection failures.
`MinReconnectInterval`	A default minimum reconnect interval in millisecs.	1100	This means that if a reconnect is needed, it won't try to reconnect more frequently than default value. The internal connections take up to 500ms per request. You can set this to higher to try reconnecting less frequently.
`obsPollInterval`	Wait for the specified interval before the Observe operation polls the nodes.	400	Set this higher or lower depending on whether the polling needs to happen less or more frequently depending on the tolerance limits for the Observe operation as compared to other operations.
`obsPollMax`	The maximum times to poll the master and replica(s) to meet the desired durability requirements.	10	You could set this value higher if the Observe operations do not complete after the normal polling.

3.4. Shutting down the Connection

The preferred method for closing a connection is to cleanly shutdown the active connection with a timeout using the `shutdown()` method with an optional timeout period and unit specification. The following will shutdown the active connection to all the configured servers after 60 seconds:

```
client.shutdown(60, TimeUnit.SECONDS);
```

The unit specification relies on the `TimeUnit` object enumerator, which supports the following values:

Constant	Description
`TimeUnit.NANOSECONDS`	Nanoseconds (10^{-9} s).
`TimeUnit.MICROSECONDS`	Microseconds (10^{-6} s).
`TimeUnit.MILLISECONDS`	Milliseconds (10^{-3} s).
`TimeUnit.SECONDS`	Seconds.

The method returns a `boolean` value indicating whether the shutdown request completed successfully.

You also can shutdown an active connection immediately by using the `shutdown()` method to your Couchbase object instance. For example:

```
client.shutdown();
```

In this form the `shutdown()` method returns no value.

Chapter 4. Java Method Summary

The `couchbase-client` and `spymemcached` libraries support the full suite of API calls to Couchbase. A summary of the supported methods are listed in Table 4.1, “Java Client Library Method Summary”.

Table 4.1. Java Client Library Method Summary

Method	Title
`client.add(key, expiry, value)`	Add a value with the specified key that does not already exist
`client.add(key, expiry, value, persistto)`	Add a value using the specified key and observe it being persisted on master and more node(s).
`client.add(key, expiry, value, persistto, replicateto)`	Add a value using the specified key and observe it being persisted on master and more node(s) and being replicated to one or more node(s).
`client.add(key, expiry, value, replicateto)`	Add a value using the specified key and observe it being replicated to one or more node(s).
`client.add(key, expiry, value, transcoder)`	Add a value that does not already exist using custom transcoder
`client.append(casunique, key, value)`	Append a value to an existing key
`client.append(casunique, key, value, transcoder)`	Append a value to an existing key
`client.asyncCAS(key, casunique, value)`	Asynchronously compare and set a value
`client.asyncCAS(key, casunique, expiry, value, transcoder)`	Asynchronously compare and set a value with custom transcoder and expiry
`client.asyncCAS(key, casunique, value, transcoder)`	Asynchronously compare and set a value with custom transcoder
`client.asyncDecr(key, offset)`	Asynchronously decrement the value of an existing key
`client.asyncGetAndTouch(key, expiry)`	Asynchronously get a value and update the expiration time for a given key
`client.asyncGetAndTouch(key, expiry, transcoder)`	Asynchronously get a value and update the expiration time for a given key using a custom transcoder
`client.asyncGet(key)`	Asynchronously get a single key
`client.asyncGetBulk(keycollection)`	Asynchronously get multiple keys
`client.asyncGetBulk(keyn)`	Asynchronously get multiple keys
`client.asyncGetBulk(transcoder, keyn)`	Asynchronously get multiple keys using a custom transcoder
`client.asyncGetBulk(keycollection, transcoder)`	Asynchronously get multiple keys using a custom transcoder
`client.asyncGet(key, transcoder)`	Asynchronously get a single key using a custom transcoder
`client.asyncGetLock(key [, getl-expiry ])`	Asynchronously get a lock.
`client.asyncGetLock(key [, getl-expiry ], transcoder)`	Asynchronously get a lock with transcoder.
`client.asyncGets(key)`	Asynchronously get single key value with CAS value

Method	Title
client.asyncGets(key, transcoder)	Asynchronously get single key value with CAS value using custom transcoder
client.asyncIncr(key, offset)	Asynchronously increment the value of an existing key
client.cas(key, casunique, value)	Compare and set
client.cas(key, casunique, expiry, value, transcoder)	Compare and set with a custom transcoder and expiry
client.cas(key, casunique, value, transcoder)	Compare and set with a custom transcoder
client.decr(key, offset)	Decrement the value of an existing numeric key
client.decr(key, offset, default)	Decrement the value of a key, setting the initial value if the key didn't already exist
client.decr(key, offset, default, expiry)	Decrement the value of a key, setting the initial value if the key didn't already exist, with an expiry
client.delete(key)	Delete the specified key
client.getAndTouch(key, expiry)	Get a value and update the expiration time for a given key
client.getAndTouch(key, expiry, transcoder)	Get a value and update the expiration time for a given key using a custom transcoder
client.get(key)	Get a single key
client.getAndLock(key [, getl-expiry])	Get and lock Asynchronously
client.getAndLock(key [, getl-expiry], transcoder)	Get and lock
client.getBulk(keycollection)	Get multiple keys
client.getBulk(keyn)	Get multiple keys
client.getBulk(transcoder, keyn)	Get multiple keys using a custom transcoder
client.getBulk(keycollection, transcoder)	Get multiple keys using a custom transcoder
client.get(key, transcoder)	Get a single key using a custom transcoder
client.gets(key)	Get single key value with CAS value
client.gets(key, transcoder)	Get single key value with CAS value using custom transcoder
client.getStats()	Get the statistics from all connections
client.getStats(statname)	Get the statistics from all connections
client.getView(ddocname, viewname)	Create a view object
client.incr(key, offset)	Increment the value of an existing numeric key
client.incr(key, offset, default)	Increment the value of an existing numeric key
client.incr(key, offset, default, expiry)	Increment the value of an existing numeric key
client.new CouchbaseClient([url] [, urls] [, username] [, password])	Create connection to Couchbase Server
client.prepend(casunique, key, value)	Prepend a value to an existing key using the default transcoder

Method	Title
`client.prepend(casunique, key, value, transcoder)`	Prepend a value to an existing key using a custom transcoder
`client.query(view, query)`	Query a view
`Query.new()`	Create a query object
`client.replace(key, expiry, value)`	Update an existing key with a new value
`client.replace(key, expiry, value, persistto)`	Replace a value using the specified key and observe it being persisted on master and more node(s).
`client.replace(key, expiry, value, persistto, replicateto)`	Replace a value using the specified key and observe it being persisted on master and more node(s) and being replicated to one or more node(s).
`client.replace(key, expiry, value, replicateto)`	Replace a value using the specified key and observe it being replicated to one or more node(s).
`client.replace(key, expiry, value, transcoder)`	Update an existing key with a new value using a custom transcoder
`client.set(key, expiry, value)`	Store a value using the specified key
`client.set(key, expiry, value, persistto)`	Store a value using the specified key and observe it being persisted on master and more node(s).
`client.set(key, expiry, value, persistto, replicateto)`	Store a value using the specified key and observe it being persisted on master and more node(s) and being replicated to one or more node(s).
`client.set(key, expiry, value, replicateto)`	Store a value using the specified key and observe it being replicated to one or more node(s).
`client.set(key, expiry, value, transcoder)`	Store a value using the specified key and a custom transcoder.
`client.touch(key, expiry)`	Update the expiry time of an item
`client.unlock(key, casunique)`	Unlock

4.1. Synchronous Method Calls

The Java Client Libraries support the core Couchbase API methods as direct calls to the Couchbase server through the API call. These direct methods can be used to provide instant storage, retrieval and updating of Couchbase key/value pairs.

For example, the `get()` is a synchronous operation:

```
Object myObject = client.get("someKey");
```

In the example code above, the client `get()` call will wait until a response has been received from the appropriately configured Couchbase servers before returning the required value or an exception.

A list of the synchronous methods are shown in Table 4.2, “Java Client Library Synchronous Method Summary”.

Table 4.2. Java Client Library Synchronous Method Summary

Method	Title
`client.append(casunique, key, value)`	Append a value to an existing key
`client.append(casunique, key, value, transcoder)`	Append a value to an existing key

Method	Title
`client.cas(key, casunique, value)`	Compare and set
`client.cas(key, casunique, expiry, value, transcoder)`	Compare and set with a custom transcoder and expiry
`client.cas(key, casunique, value, transcoder)`	Compare and set with a custom transcoder
`client.decr(key, offset)`	Decrement the value of an existing numeric key
`client.decr(key, offset, default)`	Decrement the value of a key, setting the initial value if the key didn't already exist
`client.decr(key, offset, default, expiry)`	Decrement the value of a key, setting the initial value if the key didn't already exist, with an expiry
`client.getAndTouch(key, expiry)`	Get a value and update the expiration time for a given key
`client.getAndTouch(key, expiry, transcoder)`	Get a value and update the expiration time for a given key using a custom transcoder
`client.get(key)`	Get a single key
`client.getAndLock(key [, getl-expiry ], transcoder)`	Get and lock
`client.getBulk(keycollection)`	Get multiple keys
`client.getBulk(keyn)`	Get multiple keys
`client.getBulk(transcoder, keyn)`	Get multiple keys using a custom transcoder
`client.getBulk(keycollection, transcoder)`	Get multiple keys using a custom transcoder
`client.get(key, transcoder)`	Get a single key using a custom transcoder
`client.gets(key)`	Get single key value with CAS value
`client.gets(key, transcoder)`	Get single key value with CAS value using custom transcoder
`client.getStats()`	Get the statistics from all connections
`client.getStats(statname)`	Get the statistics from all connections
`client.getView(ddocname, viewname)`	Create a view object
`client.incr(key, offset)`	Increment the value of an existing numeric key
`client.incr(key, offset, default)`	Increment the value of an existing numeric key
`client.incr(key, offset, default, expiry)`	Increment the value of an existing numeric key
`client.new CouchbaseClient([ url ] [, urls ] [, username ] [, password ])`	Create connection to Couchbase Server
`client.query(view, query)`	Query a view
`Query.new()`	Create a query object
`client.unlock(key, casunique)`	Unlock

4.2. Asynchronous Method Calls

In addition, the librares also support a range of asynchronous methods that can be used to store, update and retrieve values without having to explicitly wait for a response.

The asynchronous methods use a *Future* object or its appropriate implementation which is returned by the initial method call for the operation. The communication with the Couchbase server will be handled by the client libraries in the background so that the main program loop can continue. You can recover the status of the operation by using a method to check the status on the returned Future object. For example, rather than synchronously getting a key, an asynchronous call might look like this:

```
GetFuture getOp = client.asyncGet("someKey");
```

A list of the asynchronous methods are shown in Table 4.3, “Java Client Library Asynchronous Method Summary”.

Table 4.3. Java Client Library Asynchronous Method Summary

Method	Title
`client.add(key, expiry, value)`	Add a value with the specified key that does not already exist
`client.add(key, expiry, value, persistto)`	Add a value using the specified key and observe it being persisted on master and more node(s).
`client.add(key, expiry, value, persistto, replicateto)`	Add a value using the specified key and observe it being persisted on master and more node(s) and being replicated to one or more node(s).
`client.add(key, expiry, value, replicateto)`	Add a value using the specified key and observe it being replicated to one or more node(s).
`client.add(key, expiry, value, transcoder)`	Add a value that does not already exist using custom transcoder
`client.asyncCAS(key, casunique, value)`	Asynchronously compare and set a value
`client.asyncCAS(key, casunique, expiry, value, transcoder)`	Asynchronously compare and set a value with custom transcoder and expiry
`client.asyncCAS(key, casunique, value, transcoder)`	Asynchronously compare and set a value with custom transcoder
`client.asyncDecr(key, offset)`	Asynchronously decrement the value of an existing key
`client.asyncGetAndTouch(key, expiry)`	Asynchronously get a value and update the expiration time for a given key
`client.asyncGetAndTouch(key, expiry, transcoder)`	Asynchronously get a value and update the expiration time for a given key using a custom transcoder
`client.asyncGet(key)`	Asynchronously get a single key
`client.asyncGetBulk(keycollection)`	Asynchronously get multiple keys
`client.asyncGetBulk(keyn)`	Asynchronously get multiple keys
`client.asyncGetBulk(transcoder, keyn)`	Asynchronously get multiple keys using a custom transcoder
`client.asyncGetBulk(keycollection, transcoder)`	Asynchronously get multiple keys using a custom transcoder
`client.asyncGet(key, transcoder)`	Asynchronously get a single key using a custom transcoder
`client.asyncGetLock(key [, getl-expiry ])`	Asynchronously get a lock.
`client.asyncGetLock(key [, getl-expiry ], transcoder)`	Asynchronously get a lock with transcoder.
`client.asyncGets(key)`	Asynchronously get single key value with CAS value

Method	Title
`client.asyncGets(key, transcoder)`	Asynchronously get single key value with CAS value using custom transcoder
`client.asyncIncr(key, offset)`	Asynchronously increment the value of an existing key
`client.delete(key)`	Delete the specified key
`client.getAndLock(key [, getl-expiry ])`	Get and lock Asynchronously
`client.prepend(casunique, key, value)`	Prepend a value to an existing key using the default transcoder
`client.prepend(casunique, key, value, transcoder)`	Prepend a value to an existing key using a custom transcoder
`client.replace(key, expiry, value)`	Update an existing key with a new value
`client.replace(key, expiry, value, persistto)`	Replace a value using the specified key and observe it being persisted on master and more node(s).
`client.replace(key, expiry, value, persistto, replicateto)`	Replace a value using the specified key and observe it being persisted on master and more node(s) and being replicated to one or more node(s).
`client.replace(key, expiry, value, replicateto)`	Replace a value using the specified key and observe it being replicated to one or more node(s).
`client.replace(key, expiry, value, transcoder)`	Update an existing key with a new value using a custom transcoder
`client.set(key, expiry, value)`	Store a value using the specified key
`client.set(key, expiry, value, persistto)`	Store a value using the specified key and observe it being persisted on master and more node(s).
`client.set(key, expiry, value, persistto, replicateto)`	Store a value using the specified key and observe it being persisted on master and more node(s) and being replicated to one or more node(s).
`client.set(key, expiry, value, replicateto)`	Store a value using the specified key and observe it being replicated to one or more node(s).
`client.set(key, expiry, value, transcoder)`	Store a value using the specified key and a custom transcoder.
`client.touch(key, expiry)`	Update the expiry time of an item

This will populate the Future object `GetFuture` with the response from the server. The Future object class is defined here. The primary methods are:

- `cancel()`

  Attempts to Cancel the operation if the operation has not already been completed.

- `get()`

  Waits for the operation to complete. Gets the object returned by the operation as if the method was synchronous rather than asynchronous.

- `get(timeout, TimeUnit)`

  Gets the object waiting for a maximum time specified by `timeout` and the corresponding `TimeUnit`.

- `isDone()`

The operation has been completed successfully.

For example, you can use the timeout method to obtain the value or cancel the operation:

```
GetFuture getOp = client.asyncGet("someKey");

Object myObj;

try {
    myObj = getOp.get(5, TimeUnit.SECONDS);
} catch(TimeoutException e) {
    getOp.cancel(false);
}
```

Alternatively, you can do a blocking wait for the response by using the `get()` method:

```
Object myObj;

myObj = getOp.get();
```

4.3. Object Serialization (Transcoding)

All of the Java client library methods use the default Whalin transcoder that provides compatibility with memcached clients for the serialization of objects from the object type into a byte array used for storage within Couchbase.

You can also use a custom transcoder for the serialization of objects. This can be used to serialize objects in a format that is compatible with other languages or environments.

You can customize the transcoder by implementing a new Transcoder interface and then using this when storing and retrieving values. The Transcoder will be used to encode and decode objects into binary strings. All of the methods that store, retrieve or update information have a version that supports a custom transcoder.

4.4. Expiry Values

All values in Couchbase and Memcached can be set with an expiry value. The expiry value indicates when the item should be expired from the database and can be set when an item is added or updated.

Within `spymemcached` the expiry value is expressed in the native form of an integer as per the Memcached protocol specification. The integer value is expressed as the number of seconds, but the interpretation of the value is different based on the value itself:

- Expiry is less than `30*24*60*60` (30 days)

  The value is interpreted as the number of seconds from the point of storage or update.

- Expiry is greater than `30*24*60*60`

  The value is interpreted as the number of seconds from the epoch (January 1st, 1970).

- Expiry is 0

  This disables expiry for the item.

For example:

```
client.set("someKey", 3600, someObject);
```

The value will have an expiry time of 3600 seconds (one hour) from the time the item was stored.

The statement:

```
client.set("someKey", 1307458800, someObject);
```

Will set the expiry time as June 7th 2011, 15:00 (UTC).

Chapter 5. Connection Operations

Table 5.1. Java Client Library Connection Methods

Method	Title
`client.new CouchbaseClient([ url ] [, urls ] [, username ] [, password ])`	Create connection to Couchbase Server

API Call	`client.new CouchbaseClient([ url ] [, urls ] [, username ] [, password ])`
Asynchronous	no
Description	Create a connection to Couchbase Server with given parameters, such as node URL. The connection obtains the cluster configuration from the first host to which it has connected. Further communication operates directly with each node in the cluster as required.
Returns	(none)
Arguments	
`String url`	URL for Couchbase Server Instance, or node.
`String urls`	Linked list containing one or more URLs as strings.
`String username`	Username for Couchbase bucket.
`String password`	Password for Couchbase bucket.

Chapter 6. Store Operations

The Couchbase Java Client Library store operations set information within the Couchbase database. These are distinct from the update operations in that the key does not have to exist within the Couchbase database before being stored.

Table 6.1. Java Client Library Store Methods

Method	Title
`client.add(key, expiry, value)`	Add a value with the specified key that does not already exist
`client.add(key, expiry, value, persistto)`	Add a value using the specified key and observe it being persisted on master and more node(s).
`client.add(key, expiry, value, persistto, replicateto)`	Add a value using the specified key and observe it being persisted on master and more node(s) and being replicated to one or more node(s).
`client.add(key, expiry, value, replicateto)`	Add a value using the specified key and observe it being replicated to one or more node(s).
`client.add(key, expiry, value, transcoder)`	Add a value that does not already exist using custom transcoder
`client.replace(key, expiry, value)`	Update an existing key with a new value
`client.replace(key, expiry, value, persistto)`	Replace a value using the specified key and observe it being persisted on master and more node(s).
`client.replace(key, expiry, value, persistto, replicateto)`	Replace a value using the specified key and observe it being persisted on master and more node(s) and being replicated to one or more node(s).
`client.replace(key, expiry, value, replicateto)`	Replace a value using the specified key and observe it being replicated to one or more node(s).
`client.replace(key, expiry, value, transcoder)`	Update an existing key with a new value using a custom transcoder
`client.set(key, expiry, value)`	Store a value using the specified key
`client.set(key, expiry, value, persistto)`	Store a value using the specified key and observe it being persisted on master and more node(s).
`client.set(key, expiry, value, persistto, replicateto)`	Store a value using the specified key and observe it being persisted on master and more node(s) and being replicated to one or more node(s).
`client.set(key, expiry, value, replicateto)`	Store a value using the specified key and observe it being replicated to one or more node(s).
`client.set(key, expiry, value, transcoder)`	Store a value using the specified key and a custom transcoder.

6.1. Add Operations

The `add()` method adds a value to the database with the specified key, but will fail if the key already exists in the database.

API Call	`client.add(key, expiry, value)`
Asynchronous	yes

Description	Add a value with the specified key that does not already exist. Will fail if the key/value pair already exist.
Returns	`Future<Boolean>` (Asynchronous request value, as Boolean)
Arguments	
`String key`	Document ID used to identify the value
`int expiry`	Expiry time for key. Values larger than 30*24*60*60 seconds (30 days) are interpreted as absolute times (from the epoch).
`Object value`	Value to be stored

The `add()` method adds a value to the database using the specified key.

```
client.add("someKey", 0, someObject);
```

Unlike Section 6.2, “Set Operations” the operation can fail (and return false) if the specified key already exists.

For example, the first operation in the example below may complete if the key does not already exist, but the second operation will always fail as the first operation will have set the key:

```
OperationFuture<Boolean> addOp = client.add("someKey",0,"astring");
System.out.printf("Result was %b",addOp.get());
addOp =  client.add("someKey",0,"anotherstring");
System.out.printf("Result was %b",addOp.get());
```

API Call	`client.add(key, expiry, value, transcoder)`
Asynchronous	yes
Description	Add a value with the specified key that does not already exist. Will fail if the key/value pair already exist.
Returns	`Future<Boolean>` (Asynchronous request value, as Boolean)
Arguments	
`String key`	Document ID used to identify the value
`int expiry`	Expiry time for key. Values larger than 30*24*60*60 seconds (30 days) are interpreted as absolute times (from the epoch).
`Object value`	Value to be stored
`Transcoder<T> transcoder`	Transcoder class to be used to serialize value

This method is identical to the `add()` method, but supports the use of a custom transcoder for serialization of the object value. For more information on transcoding, see Section 4.3, “Object Serialization (Transcoding)”.

6.2. Set Operations

The `set()` operations store a value into Couchbase or Memcached using the specified key and value. The value is stored against the specified key, even if the key already exists and has data. This operation overwrites the existing with the new data. The store operation in this case is asynchronous.

API Call	`client.set(key, expiry, value)`
Asynchronous	yes
Description	Store a value using the specified key, whether the key already exists or not. Will overwrite a value if the given key/value already exists.
Returns	`OperationFuture<Boolean>` (Asynchronous request value, as Boolean)

Arguments	
`String key`	Document ID used to identify the value
`int expiry`	Expiry time for key. Values larger than 30*24*60*60 seconds (30 days) are interpreted as absolute times (from the epoch).
`Object value`	Value to be stored

The first form of the `set()` method stores the key, sets the expiry (use 0 for no expiry), and the corresponding object value using the built in transcoder for serialization.

The simplest form of the command is:

```
client.set("someKey", 3600, someObject);
```

The `Boolean` return value from the asynchronous operation return value will be true if the value was stored. For example:

```
OperationFuture<Boolean> setOp = client.set("someKey",0,"astring");
System.out.printf("Result was %b",setOp.get());
```

API Call	`client.set(key, expiry, value, transcoder)`
Asynchronous	yes
Description	Store a value using the specified key and a custom transcoder.
Returns	`OperationFuture<Boolean>` (Asynchronous request value, as Boolean)
Arguments	
`String key`	Document ID used to identify the value
`int expiry`	Expiry time for key. Values larger than 30*24*60*60 seconds (30 days) are interpreted as absolute times (from the epoch).
`Object value`	Value to be stored
`Transcoder<T> transcoder`	Transcoder class to be used to serialize value

The second form of the `set()` method supports the use of a custom transcoder for serialization of the object value. For more information on transcoding, see Section 4.3, “Object Serialization (Transcoding)”.

6.3. Store Operations with Durability Requirements

API Call	`client.set(key, expiry, value, persistto)`
Asynchronous	yes
Description	Store a value using the specified key and observe it being persisted on master and more node(s).
Returns	`OperationFuture<Boolean>` (Asynchronous request value, as Boolean)
Arguments	
`String key`	Document ID used to identify the value
`int expiry`	Expiry time for key. Values larger than 30*24*60*60 seconds (30 days) are interpreted as absolute times (from the epoch).
`Object value`	Value to be stored
`enum persistto`	Specify the number of nodes on which the document must be persisted to before returning.

API Call	`client.set(key, expiry, value, replicateto)`
Asynchronous	yes
Description	Store a value using the specified key and observe it being replicated to one or more node(s).
Returns	`OperationFuture<Boolean>` (Asynchronous request value, as Boolean)
Arguments	
`String key`	Document ID used to identify the value
`int expiry`	Expiry time for key. Values larger than 30*24*60*60 seconds (30 days) are interpreted as absolute times (from the epoch).
`Object value`	Value to be stored
`enum replicateto`	Specify the number of nodes on which the document must be replicated to before returning

API Call	`client.set(key, expiry, value, persistto, replicateto)`
Asynchronous	yes
Description	Store a value using the specified key and observe it being persisted on master and more node(s) and being replicated to one or more node(s).
Returns	`OperationFuture<Boolean>` (Asynchronous request value, as Boolean)
Arguments	
`String key`	Document ID used to identify the value
`int expiry`	Expiry time for key. Values larger than 30*24*60*60 seconds (30 days) are interpreted as absolute times (from the epoch).
`Object value`	Value to be stored
`enum persistto`	Specify the number of nodes on which the document must be persisted to before returning.
`enum replicateto`	Specify the number of nodes on which the document must be replicated to before returning

This method is identical to the `set()` method, but supports the ability to observe the persistence on the master and replicas and the propagation to the replicas. Using these methods above, it's possible to set the persistence requirements for the data on the nodes.

The persistence requirements can be specified in terms of how the data should be persisted on the master and the replicas using `PeristTo` and how the data should be propagated to the replicas using `ReplicateTo` respectively.

The client library will poll the server until the persistence requirements are met. The method will return FALSE if the requirments are impossible to meet based on the configuration (inadequate number of replicas) or even after a set amount of retries the persistence requirments could not be met.

The program snippet below illustrates how to specify a requirement that the data should be persisted on 4 nodes (master and three replicas).

```
// Perist to all four nodes including master
OperationFuture<Boolean> setOp =
   c.set("key", 0, "value", PersistTo.FOUR);
System.out.printf("Result was %b", setOp.get());
```

The peristence requirements can be specified for both the master and replicas. In the case above, it's required that the key and value is persisted on all the 4 nodes (including replicas).

In the following, the requirement is specified as requiring persistence to the master and propagation of the data to the three replicas. This requirment is weaker than requring the data to be persisted on all four nodes including the three replicas.

```
// Perist to master and propagate the data to three replicas
OperationFuture<Boolean> setOp =
   c.set("key", 0, "value", PersistTo.MASTER, ReplicateTo.THREE);
System.out.printf("Result was %b", setOp.get());
```

Similar requirements can used with the *add()* and *replace()* mutation operations.

Chapter 7. Retrieve Operations

The retrieve operations get information from the Couchbase database. A summary of the available API calls is listed below.

Table 7.1. Java Client Library Retrieval Methods

Method	Title
`client.asyncGetAndTouch(key, expiry)`	Asynchronously get a value and update the expiration time for a given key
`client.asyncGetAndTouch(key, expiry, transcoder)`	Asynchronously get a value and update the expiration time for a given key using a custom transcoder
`client.asyncGet(key)`	Asynchronously get a single key
`client.asyncGetBulk(keycollection)`	Asynchronously get multiple keys
`client.asyncGetBulk(keyn)`	Asynchronously get multiple keys
`client.asyncGetBulk(transcoder, keyn)`	Asynchronously get multiple keys using a custom transcoder
`client.asyncGetBulk(keycollection, transcoder)`	Asynchronously get multiple keys using a custom transcoder
`client.asyncGet(key, transcoder)`	Asynchronously get a single key using a custom transcoder
`client.asyncGetLock(key [, getl-expiry ])`	Asynchronously get a lock.
`client.asyncGetLock(key [, getl-expiry ], transcoder)`	Asynchronously get a lock with transcoder.
`client.asyncGets(key)`	Asynchronously get single key value with CAS value
`client.asyncGets(key, transcoder)`	Asynchronously get single key value with CAS value using custom transcoder
`client.getAndTouch(key, expiry)`	Get a value and update the expiration time for a given key
`client.getAndTouch(key, expiry, transcoder)`	Get a value and update the expiration time for a given key using a custom transcoder
`client.get(key)`	Get a single key
`client.getAndLock(key [, getl-expiry ])`	Get and lock Asynchronously
`client.getAndLock(key [, getl-expiry ], transcoder)`	Get and lock
`client.getBulk(keycollection)`	Get multiple keys
`client.getBulk(keyn)`	Get multiple keys
`client.getBulk(transcoder, keyn)`	Get multiple keys using a custom transcoder
`client.getBulk(keycollection, transcoder)`	Get multiple keys using a custom transcoder
`client.get(key, transcoder)`	Get a single key using a custom transcoder
`client.gets(key)`	Get single key value with CAS value
`client.gets(key, transcoder)`	Get single key value with CAS value using custom transcoder
`client.unlock(key, casunique)`	Unlock

7.1. Synchronous `get` Methods

The synchronous `get()` methods allow for direct access to a given key/value pair.

API Call	`client.get(key)`
Asynchronous	no
Description	Get one or more key values
Returns	`Object` (Binary object)
Arguments	
`String key`	Document ID used to identify the value

The `get()` method obtains an object stored in Couchbase using the default transcoder for serialization of the object.

For example:

```
Object myObject = client.get("someKey");
```

Transcoding of the object assumes the default transcoder was used when the value was stored. The returned object can be of any type.

If the request key does no existing in the database then the returned value is null.

API Call	`client.get(key, transcoder)`
Asynchronous	no
Description	Get one or more key values
Returns	`T` (Transcoded object)
Arguments	
`String key`	Document ID used to identify the value
`Transcoder<T> transcoder`	Transcoder class to be used to serialize value

The second form of the `get()` retrieves a value from Couchbase using a custom transcoder.

For example to obtain an integer value using the IntegerTranscoder:

```
Transcoder<Integer> tc = new IntegerTranscoder();
Integer ic = client.get("someKey", tc);
```

7.2. Asynchronous `get` Methods

The asynchronous `asyncGet()` methods allow to retrieve a given value for a key without waiting actively for a response.

API Call	`client.asyncGet(key)`
Asynchronous	yes
Description	Get one or more key values
Returns	`Future<Object>` (Asynchronous request value, as Object)
Arguments	
`String key`	Document ID used to identify the value

Exceptions	
`TimeoutException`	Value could not be retrieved

The first form of the `asyncGet()` method obtains a value for a given key returning a Future object so that the value can be later retrieved. For example, to get a key with a stored String value:

```
GetFuture<Object> getOp =
    client.asyncGet("samplekey");

String username;

try {
    username = (String) getOp.get(5, TimeUnit.SECONDS);
} catch(Exception e) {
    getOp.cancel(false);
}
```

API Call	`client.asyncGet(key, transcoder)`
Asynchronous	yes
Description	Get one or more key values
Returns	`Future<T>` (Asynchronous request value, as Transcoded Object)
Arguments	
`String key`	Document ID used to identify the value
`Transcoder<T> transcoder`	Transcoder class to be used to serialize value

The second form is identical to the first, but includes the ability to use a custom transcoder on the stored value.

7.3. Get-and-Touch Methods

The Get-and-Touch (GAT) methods obtain a value for a given key and update the expiry time for the key. This can be useful for session values and other information where you want to set an expiry time, but don't want the value to expire while the value is still in use.

API Call	`client.getAndTouch(key, expiry)`
Asynchronous	no
Description	Get a value and update the expiration time for a given key
Introduced Version	1.0
Returns	`CASValue` (Check and set object)
Arguments	
`String key`	Document ID used to identify the value
`int expiry`	Expiry time for key. Values larger than 30*24*60*60 seconds (30 days) are interpreted as absolute times (from the epoch).

The first form of the `getAndTouch()` obtains a given value and updates the expiry time. For example, to get session data and renew the expiry time to five minutes:

```
session = client.getAndTouch("sessionid",300);
```

API Call	`client.getAndTouch(key, expiry, transcoder)`
Asynchronous	no

Description	Get a value and update the expiration time for a given key
Introduced Version	1.0
Returns	`CASValue` (Check and set object)
Arguments	
`String key`	Document ID used to identify the value
`int expiry`	Expiry time for key. Values larger than 30*24*60*60 seconds (30 days) are interpreted as absolute times (from the epoch).
`Transcoder<T> transcoder`	Transcoder class to be used to serialize value

The second form supports the use of a custom transcoder for the stored value information.

API Call	`client.asyncGetAndTouch(key, expiry)`
Asynchronous	yes
Description	Get a value and update the expiration time for a given key
Introduced Version	1.0
Returns	`Future<CASValue<Object>>` (Asynchronous request value, as CASValue, as Object)
Arguments	
`String key`	Document ID used to identify the value
`int expiry`	Expiry time for key. Values larger than 30*24*60*60 seconds (30 days) are interpreted as absolute times (from the epoch).

The asynchronous methods obtain the value and update the expiry time without requiring you to actively wait for the response. The returned value is a CAS object with the embedded value object.

```
Future<CASValue<Object>> future = client.asyncGetAndTouch("sessionid", 300);

CASValue casv;

try {
    casv = future.get(5, TimeUnit.SECONDS);
} catch(Exception e) {
    future.cancel(false);
}
```

API Call	`client.asyncGetAndTouch(key, expiry, transcoder)`
Asynchronous	yes
Description	Get a value and update the expiration time for a given key
Introduced Version	1.0
Returns	`Future<CASValue<T>>` (Asynchronous request value, as CASValue as Transcoded object)
Arguments	
`String key`	Document ID used to identify the value
`int expiry`	Expiry time for key. Values larger than 30*24*60*60 seconds (30 days) are interpreted as absolute times (from the epoch).
`Transcoder<T> transcoder`	Transcoder class to be used to serialize value

The second form of the asynchronous method supports the use of a custom transcoder for the stored object.

7.4. CAS `get` Methods

The `gets()` methods obtain a CAS value for a given key. The CAS value is used in combination with the corresponding Check-and-Set methods when updating a value. For example, you can use the CAS value with the `append()` operation to update a value only if the CAS value you supply matches. For more information see Section 8.1, “Append Methods” and Section 8.3, “Check-and-Set Methods”.

API Call	`client.gets(key)`
Asynchronous	no
Description	Get single key value with CAS value
Returns	`CASValue` (Check and set object)
Arguments	
`String key`	Document ID used to identify the value

The `gets()` method obtains a `CASValue` for a given key. The CASValue holds the CAS to be used when performing a Check-And-Set operation, and the corresponding value for the given key.

For example, to obtain the CAS and value for the key `someKey`:

```
CASValue status = client.gets("someKey");
System.out.printf("CAS is %s\n",status.getCas());
System.out.printf("Result was %s\n",status.getValue());
```

The CAS value can be used in a CAS call such as `append()`:

```
client.append(status.getCas(),"someKey", "appendedstring");
```

API Call	`client.gets(key, transcoder)`
Asynchronous	no
Description	Get single key value with CAS value
Returns	`CASValue` (Check and set object)
Arguments	
`String key`	Document ID used to identify the value
`Transcoder<T> transcoder`	Transcoder class to be used to serialize value

The second form of the `gets()` method supports the use of a custom transcoder.

API Call	`client.asyncGets(key)`
Asynchronous	yes
Description	Get single key value with CAS value
Returns	`Future<CASValue<Object>>` (Asynchronous request value, as CASValue, as Object)
Arguments	
`String key`	Document ID used to identify the value

The `asyncGets()` method obtains the `CASValue` object for a stored value against the specified key, without requiring an explicit wait for the returned value.

For example:

```
Future<CASValue<Object>> future = client.asyncGets("someKey");

System.out.printf("CAS is %s\n",future.get(5,TimeUnit.SECONDS).getCas());
```

Note that you have to extract the CASValue from the Future response, and then the CAS value from the corresponding object. This is performed here in a single statement.

API Call	`client.asyncGets(key, transcoder)`
Asynchronous	yes
Description	Get single key value with CAS value
Returns	`Future<CASValue<T>>` (Asynchronous request value, as CASValue as Transcoded object)
Arguments	
`String key`	Document ID used to identify the value
`Transcoder<T> transcoder`	Transcoder class to be used to serialize value

The final form of the `asyncGets()` method supports the use of a custom transcoder.

7.5. Bulk `get` Methods

The bulk `getBulk()` methods allow you to get one or more items from the database in a single request. Using the bulk methods is more efficient than multiple single requests as the operation can be conducted in a single network call.

API Call	`client.getBulk(keycollection)`
Asynchronous	no
Description	Get one or more key values
Returns	`Map<String,Object>` (Map of Strings/Objects)
Arguments	
`Collection<String> keycollection`	One or more keys used to reference a value

The first format accepts a `String` `Collection` as the request argument which is used to specify the list of keys that you want to retrieve. The return type is `Map` between the keys and object values.

For example:

```
Map<String,Object> keyvalues = client.getBulk(keylist);

System.out.printf("A is %s\n",keyvalues.get("keyA"));
System.out.printf("B is %s\n",keyvalues.get("keyB"));
System.out.printf("C is %s\n",keyvalues.get("keyC"));
```

Note

The returned map will only contain entries for keys that exist from the original request. For example, if you request the values for three keys, but only one exists, the resultant map will contain only that one key/value pair.

API Call	`client.getBulk(keycollection, transcoder)`

Asynchronous	no
Description	Get one or more key values
Returns	`Map<String,T>` (Map of Strings/Transcoded objects)
Arguments	
`Collection<String> keycollection`	One or more keys used to reference a value
`Transcoder<T> transcoder`	Transcoder class to be used to serialize value

The second form of the `getBulk()` method supports the same `Collection` argument, but also supports the use of a custom transcoder on the returned values.

Note

The specified transcoder will be used for every value requested from the database.

API Call	`client.getBulk(keyn)`
Asynchronous	no
Description	Get one or more key values
Returns	`Map<String,Object>` (Map of Strings/Objects)
Arguments	
`String... keyn`	One or more keys used to reference a value

The third form of the `getBulk()` method supports a variable list of arguments with each interpreted as the key to be retrieved from the database.

For example, the equivalent of the `Collection` method operation using this method would be:

```
Map<String,Object> keyvalues = client.getBulk("keyA","keyB","keyC");

System.out.printf("A is %s\n",keyvalues.get("keyA"));
System.out.printf("B is %s\n",keyvalues.get("keyB"));
System.out.printf("C is %s\n",keyvalues.get("keyC"));
```

The return `Map` will only contain entries for keys that exist. Non-existent keys will be silently ignored.

API Call	`client.getBulk(transcoder, keyn)`
Asynchronous	no
Description	Get one or more key values
Returns	`Map<String,T>` (Map of Strings/Transcoded objects)
Arguments	
`Transcoder<T> transcoder`	Transcoder class to be used to serialize value
`String... keyn`	One or more keys used to reference a value

The fourth form of the `getBulk()` method uses the variable list of arguments but supports a custom transcoder.

Warning

Note that unlike other formats of the methods used for supporting custom transcoders, the transcoder specification is at the start of the argument list, not the end.

API Call	`client.asyncGetBulk(keycollection)`
Asynchronous	yes
Description	Get one or more key values
Returns	`BulkFuture<Map<String,Object>>` (Map of Strings/Objects)
Arguments	
`Collection<String> keycollection`	One or more keys used to reference a value

The asynchronous `getBulk()` method supports a `Collection` of keys to be retrieved, returning a BulkFuture object (part of the `spymemcached` package) with the returned map of key/value information. As with other asynchronous methods, the benefit is that you do not need to actively wait for the response.

The `BulkFuture` object operates slightly different in context to the standard `Future` object. Whereas the `Future` object gets a returned single value, the `BulkFuture` object will return an object containing as many keys as have been returned. For very large queries requesting large numbers of keys this means that multiple requests may be required to obtain every key from the original list.

For example, the code below will obtain as many keys as possible from the supplied `Collection`.

```
BulkFuture<Map<String,Object>> keyvalues = client.asyncGetBulk(keylist);

Map<String,Object> keymap = keyvalues.getSome(5,TimeUnit.SECONDS);

System.out.printf("A is %s\n",keymap.get("keyA"));
System.out.printf("B is %s\n",keymap.get("keyB"));
System.out.printf("C is %s\n",keymap.get("keyC"));
```

API Call	`client.asyncGetBulk(keycollection, transcoder)`
Asynchronous	yes
Description	Get one or more key values
Returns	`BulkFuture<Map<String,T>>` (Map of Strings/Transcoded objects)
Arguments	
`Collection<String> keycollection`	One or more keys used to reference a value
`Transcoder<T> transcoder`	Transcoder class to be used to serialize value

The second form of the asynchronous `getBulk()` method supports the custom transcoder for the returned values.

API Call	`client.asyncGetBulk(keyn)`
Asynchronous	yes
Description	Get one or more key values
Returns	`BulkFuture<Map<String,Object>>` (Map of Strings/Objects)
Arguments	
`String... keyn`	One or more keys used to reference a value

The third form is identical to the multi-argument key request method (see collection based `asyncBulkGet()`), except that the operation occurs asynchronously.

API Call	`client.asyncGetBulk(transcoder, keyn)`

Asynchronous	yes
Description	Get one or more key values
Returns	`BulkFuture<Map<String,T>>` (Map of Strings/Transcoded objects)
Arguments	
`Transcoder<T> transcoder`	Transcoder class to be used to serialize value
`String... keyn`	One or more keys used to reference a value

The final form of the `asyncGetBulk()` method supports a custom transcoder with the variable list of keys supplied as arguments.

7.6. Get and Lock

API Call	`client.asyncGetLock(key [, getl-expiry ], transcoder)`		
Asynchronous	yes		
Description	Get the value for a key, lock the key from changes		
Returns	`Future<CASValue<T>>` (Asynchronous request value, as CASValue as Transcoded object)		
Arguments			
`String key`	Document ID used to identify the value		
`int getl-expiry`	Expiry time for lock		
	Default	15	
	Maximum	30	
`Transcoder<T> transcoder`	Transcoder class to be used to serialize value		

API Call	`client.asyncGetLock(key [, getl-expiry ])`		
Asynchronous	yes		
Description	Get the value for a key, lock the key from changes		
Returns	`Future<CASValue<Object>>` (Asynchronous request value, as CASValue, as Object)		
Arguments			
`String key`	Document ID used to identify the value		
`int getl-expiry`	Expiry time for lock		
	Default	15	
	Maximum	30	

API Call	`client.getAndLock(key [, getl-expiry ], transcoder)`
Asynchronous	no
Description	Get the value for a key, lock the key from changes
Returns	`CASValue<T>` (CASValue as Transcoded object)
Arguments	

`String key`	Document ID used to identify the value	
`int getl-expiry`	Expiry time for lock	
	Default	15
	Maximum	30
`Transcoder<T> transcoder`	Transcoder class to be used to serialize value	
Exceptions		
`OperationTimeoutException`	Exception timeout occured while waiting for value.	
`RuntimeException`	Exception object specifying interruption while waiting for value.	

The simplest form of the method is without the transcoder as below.

API Call	`client.getAndLock(key [, getl-expiry ])`	
Asynchronous	yes	
Description	Get the value for a key, lock the key from changes	
Returns	`CASValue<Object>` (CASValue as Object)	
Arguments		
`String key`	Document ID used to identify the value	
`int getl-expiry`	Expiry time for lock	
	Default	15
	Maximum	30
Exceptions		
`OperationTimeoutException`	Exception timeout occured while waiting for value.	
`RuntimeException`	Exception object specifying interruption while waiting for value.	

```
CASValue<Object> casv = client.getAndLock("keyA", 3);
```

Will lock keyA for 3 seconds or until an Unlock is issued.

7.7. Unlock

API Call	`client.unlock(key, casunique)`
Asynchronous	no
Description	Unlock a previously locked key by providing the corresponding CAS value that was returned during the lock
Returns	`Boolean` (Boolean (true/false))
Arguments	
`String key`	Document ID used to identify the value
`long casunique`	Unique value used to verify a key/value combination
Exceptions	
`InterruptedException`	Interrupted Exception while waiting for value.

`OperationTimeoutException`	Exception timeout occured while waiting for value.
`RuntimeException`	Exception object specifying interruption while waiting for value.

```
CASValue<Object> casv = client.getAndLock("keyA", 3);
//Use CAS Value to Unlock
client.unlock("getunltest", casv.getCas());
```

Chapter 8. Update Operations

The update methods support different methods of updating and changing existing information within Couchbase. A list of the available methods is listed below.

Table 8.1. Java Client Library Update Methods

Method	Title
`client.append(casunique, key, value)`	Append a value to an existing key
`client.append(casunique, key, value, transcoder)`	Append a value to an existing key
`client.asyncCAS(key, casunique, value)`	Asynchronously compare and set a value
`client.asyncCAS(key, casunique, expiry, value, transcoder)`	Asynchronously compare and set a value with custom transcoder and expiry
`client.asyncCAS(key, casunique, value, transcoder)`	Asynchronously compare and set a value with custom transcoder
`client.asyncDecr(key, offset)`	Asynchronously decrement the value of an existing key
`client.asyncIncr(key, offset)`	Asynchronously increment the value of an existing key
`client.cas(key, casunique, value)`	Compare and set
`client.cas(key, casunique, expiry, value, transcoder)`	Compare and set with a custom transcoder and expiry
`client.cas(key, casunique, value, transcoder)`	Compare and set with a custom transcoder
`client.decr(key, offset)`	Decrement the value of an existing numeric key
`client.decr(key, offset, default)`	Decrement the value of a key, setting the initial value if the key didn't already exist
`client.decr(key, offset, default, expiry)`	Decrement the value of a key, setting the initial value if the key didn't already exist, with an expiry
`client.delete(key)`	Delete the specified key
`client.incr(key, offset)`	Increment the value of an existing numeric key
`client.incr(key, offset, default)`	Increment the value of an existing numeric key
`client.incr(key, offset, default, expiry)`	Increment the value of an existing numeric key
`client.prepend(casunique, key, value)`	Prepend a value to an existing key using the default transcoder
`client.prepend(casunique, key, value, transcoder)`	Prepend a value to an existing key using a custom transcoder
`client.touch(key, expiry)`	Update the expiry time of an item

8.1. Append Methods

The `append()` methods allow you to add information to an existing key/value pair in the database. You can use this to add information to a string or other data after the existing data.

The `append()` methods append raw serialized data on to the end of the existing data in the key. If you have previously stored a serialized object into Couchbase and then use append, the content of the serialized object will not be extended.

For example, adding an `Array` of integers into the database, and then using `append()` to add another integer will result in the key referring to a serialized version of the array, immediately followed by a serialized version of the integer. It will not contain an updated array with the new integer appended to it. De-serialization of objects that have had data appended may result in data corruption.

API Call	`client.append(casunique, key, value)`
Asynchronous	no
Description	Append a value to an existing key
Returns	`Object` (Binary object)
Arguments	
`long casunique`	Unique value used to verify a key/value combination
`String key`	Document ID used to identify the value
`Object value`	Value to be stored

The `append()` appends information to the end of an existing key/value pair. The `append()` function requires a CAS value. For more information on CAS values, see Section 7.4, “CAS `get` Methods”.

For example, to append a string to an existing key:

```
CASValue<Object> casv = client.gets("samplekey");
client.append(casv.getCas(),"samplekey", "appendedstring");
```

You can check if the append operation succeeded by using the return `OperationFuture` value:

```
OperationFuture<Boolean> appendOp =
    client.append(casv.getCas(),"notsamplekey", "appendedstring");

try {
    if (appendOp.get().booleanValue()) {
        System.out.printf("Append succeeded\n");
    }
    else {
        System.out.printf("Append failed\n");
    }
}
catch (Exception e) {
...
}
```

API Call	`client.append(casunique, key, value, transcoder)`
Asynchronous	no
Description	Append a value to an existing key
Returns	`Object` (Binary object)
Arguments	
`long casunique`	Unique value used to verify a key/value combination
`String key`	Document ID used to identify the value
`Object value`	Value to be stored
`Transcoder<T> transcoder`	Transcoder class to be used to serialize value

The second form of the `append()` method supports the use of custom transcoder.

8.2. Prepend Methods

The `prepend()` methods insert information before the existing data for a given key. Note that as with the `append()` method, the information will be inserted before the existing binary data stored in the key, which means that serialization of complex objects may lead to corruption when using `prepend()`.

API Call	`client.prepend(casunique, key, value)`
Asynchronous	yes
Description	Prepend a value to an existing key
Returns	`Future<Boolean>` (Asynchronous request value, as Boolean)
Arguments	
`long casunique`	Unique value used to verify a key/value combination
`String key`	Document ID used to identify the value
`Object value`	Value to be stored

The `prepend()` inserts information before the existing data stored in the key/value pair. The `prepend()` function requires a CAS value. For more information on CAS values, see Section 7.4, “CAS `get` Methods”.

For example, to prepend a string to an existing key:

```
CASValue<Object> casv = client.gets("samplekey");
client.prepend(casv.getCas(),"samplekey", "prependedstring");
```

You can check if the prepend operation succeeded by using the return `OperationFuture` value:

```
OperationFuture<Boolean> prependOp =
    client.prepend(casv.getCas(),"notsamplekey", "prependedstring");

try {
    if (prependOp.get().booleanValue()) {
        System.out.printf("Prepend succeeded\n");
    }
    else {
        System.out.printf("Prepend failed\n");
    }
}
catch (Exception e) {
...
}
```

API Call	`client.prepend(casunique, key, value, transcoder)`
Asynchronous	yes
Description	Prepend a value to an existing key
Returns	`Future<Boolean>` (Asynchronous request value, as Boolean)
Arguments	
`long casunique`	Unique value used to verify a key/value combination
`String key`	Document ID used to identify the value
`Object value`	Value to be stored
`Transcoder<T> transcoder`	Transcoder class to be used to serialize value

The secondary form of the `prepend()` method supports the use of a custom transcoder for updating the key/value pair.

8.3. Check-and-Set Methods

The check-and-set methods provide a mechanism for updating information only if the client knows the check (CAS) value. This can be used to prevent clients from updating values in the database that may have changed since the client obtained the value. Methods for storing and updating information support a CAS method that allows you to ensure that the client is updating the version of the data that the client retrieved.

The check value is in the form of a 64-bit integer which is updated every time the value is modified, even if the update of the value does not modify the binary data. Attempting to set or update a key/value pair where the CAS value does not match the value stored on the server will fail.

The `cas()` methods are used to explicitly set the value only if the CAS supplied by the client matches the CAS on the server, analogous to the Section 6.2, “Set Operations” method.

With all CAS operations, the `CASResponse` value returned indicates whether the operation succeeded or not, and if not why. The `CASResponse` is an `Enum` with three possible values:

- `EXISTS`

  The item exists, but the CAS value on the database does not match the value supplied to the CAS operation.

- `NOT_FOUND`

  The specified key does not exist in the database. An `add()` method should be used to add the key to the database.

- `OK`

  The CAS operation was successful and the updated value is stored in Couchbase

API Call	`client.cas(key, casunique, value)`
Asynchronous	no
Description	Compare and set a value providing the supplied CAS key matches
Returns	`CASResponse` (Check and set object)
Arguments	
`String key`	Document ID used to identify the value
`long casunique`	Unique value used to verify a key/value combination
`Object value`	Value to be stored

The first form of the `cas()` method allows for an item to be set in the database only if the CAS value supplied matches that stored on the server.

For example:

```
CASResponse casr = client.cas("caskey", casvalue, "new string value");

if (casr.equals(CASResponse.OK)) {
    System.out.println("Value was updated");
}
else if (casr.equals(CASResponse.NOT_FOUND)) {
    System.out.println("Value is not found");
}
else if (casr.equals(CASResponse.EXISTS)) {
    System.out.println("Value exists, but CAS didn't match");
}
```

API Call	`client.cas(key, casunique, value, transcoder)`

Asynchronous	no
Description	Compare and set a value providing the supplied CAS key matches
Returns	`CASResponse` (Check and set object)
Arguments	
`String key`	Document ID used to identify the value
`long casunique`	Unique value used to verify a key/value combination
`Object value`	Value to be stored
`Transcoder<T> transcoder`	Transcoder class to be used to serialize value

The second form of the method supports using a custom transcoder for storing a value.

API Call	`client.cas(key, casunique, expiry, value, transcoder)`
Asynchronous	no
Description	Compare and set a value providing the supplied CAS key matches
Returns	`CASResponse` (Check and set object)
Arguments	
`String key`	Document ID used to identify the value
`long casunique`	Unique value used to verify a key/value combination
`int expiry`	Expiry time for key. Values larger than 30*24*60*60 seconds (30 days) are interpreted as absolute times (from the epoch).
`Object value`	Value to be stored
`Transcoder<T> transcoder`	Transcoder class to be used to serialize value

This form of the `cas()` method updates both the key value and the expiry time for the value. For information on expiry values, see Section 4.4, “Expiry Values”.

For example the following attempts to set the key `caskey` with an updated value, setting the expiry times to 3600 seconds (one hour).

```
Transcoder<Integer> tc = new IntegerTranscoder();
CASResponse casr = client.cas("caskey", casvalue, 3600, 1200, tc);
```

API Call	`client.asyncCAS(key, casunique, value)`
Asynchronous	yes
Description	Compare and set a value providing the supplied CAS key matches
Returns	`Future<CASResponse>` (Asynchronous request value, as CASResponse)
Arguments	
`String key`	Document ID used to identify the value
`long casunique`	Unique value used to verify a key/value combination
`Object value`	Value to be stored

Performs an asynchronous CAS operation on the given key/value. You can use this method to set a value using CAS without waiting for the response. The following example requests an update of a key, timing out after 5 seconds if the operation was not successful.

```
Future<CASResponse> future = client.asyncCAS("someKey", casvalue, "updatedvalue");

CASResponse casr;

try {
    casr = future.get(5, TimeUnit.SECONDS);
} catch(TimeoutException e) {
    future.cancel(false);
}
```

API Call	`client.asyncCAS(key, casunique, value, transcoder)`
Asynchronous	yes
Description	Compare and set a value providing the supplied CAS key matches
Returns	`Future<CASResponse>` (Asynchronous request value, as CASResponse)
Arguments	
`String key`	Document ID used to identify the value
`long casunique`	Unique value used to verify a key/value combination
`Object value`	Value to be stored
`Transcoder<T> transcoder`	Transcoder class to be used to serialize value

Performs an asynchronous CAS operation on the given key/value using a custom transcoder. The example below shows the update of an existing value using a custom Integer transcoder.

```
Transcoder<Integer> tc = new IntegerTranscoder();
Future<CASResponse> future = client.asyncCAS("someKey", casvalue, 1200, tc);

CASResponse casr;

try {
    casr = future.get(5, TimeUnit.SECONDS);
} catch(TimeoutException e) {
    future.cancel(false);
}
```

API Call	`client.asyncCAS(key, casunique, expiry, value, transcoder)`
Asynchronous	yes
Description	Compare and set a value providing the supplied CAS key matches
Returns	`Future<CASResponse>` (Asynchronous request value, as CASResponse)
Arguments	
`String key`	Document ID used to identify the value
`long casunique`	Unique value used to verify a key/value combination
`int expiry`	Expiry time for key. Values larger than 30*24*60*60 seconds (30 days) are interpreted as absolute times (from the epoch).
`Object value`	Value to be stored
`Transcoder<T> transcoder`	Transcoder class to be used to serialize value

The final form of the `asyncCAS()` method supports a custom transcoder and setting the associated expiry value. For example, to update a value and set the expiry to 60 seconds:

```
Transcoder<Integer> tc = new IntegerTranscoder();
Future<CASResponse> future = client.asyncCAS("someKey", casvalue, 60, 1200, tc);

CASResponse casr;
```

```
try {
    casr = future.get(5, TimeUnit.SECONDS);
} catch(TimeoutException e) {
    future.cancel(false);
}
```

8.4. Delete Methods

The `delete()` method deletes an item in the database with the specified key. Delete operations are asynchronous only.

API Call	`client.delete(key)`
Asynchronous	yes
Description	Delete a key/value
Returns	`OperationFuture<Boolean>` (Asynchronous request value, as Boolean)
Arguments	
`String key`	Document ID used to identify the value

For example, to delete an item you might use code similar to the following:

```
OperationFuture<Boolean> delOp =
    client.delete("samplekey");

try {
    if (delOp.get().booleanValue()) {
        System.out.printf("Delete succeeded\n");
    }
    else {
        System.out.printf("Delete failed\n");
    }

}
catch (Exception e) {
    System.out.println("Failed to delete " + e);
}
```

8.5. Decrement Methods

The decrement methods reduce the value of a given key if the corresponding value can be parsed to an integer value. These operations are provided at a protocol level to eliminate the need to get, update, and reset a simple integer value in the database. All the Java Client Library methods support the use of an explicit offset value that will be used to reduce the stored value in the database.

API Call	`client.decr(key, offset)`
Asynchronous	no
Description	Decrement the value of an existing numeric key. The Couchbase Server stores numbers as unsigned values. Therefore the lowest you can decrement is to zero.
Returns	`long` (Numeric value)
Arguments	
`String key`	Document ID used to identify the value
`int offset`	Integer offset value to increment/decrement (default 1)

The first form of the `decr()` method accepts the keyname and offset value to be used when reducing the server-side integer. For example, to decrement the server integer `dlcounter` by 5:

```
client.decr("dlcounter",5);
```

The return value is the updated value of the specified key.

API Call	`client.decr(key, offset, default)`
Asynchronous	no
Description	Decrement the value of an existing numeric key. The Couchbase Server stores numbers as unsigned values. Therefore the lowest you can decrement is to zero.
Returns	`long` (Numeric value)
Arguments	
`String key`	Document ID used to identify the value
`int offset`	Integer offset value to increment/decrement (default 1)
`int default`	Default value to increment/decrement if key does not exist

The second form of the `decr()` method will decrement the given key by the specified `offset` value if the key already exists, or set the key to the specified `default` value if the key does not exist. This can be used in situations where you are recording a counter value but do not know whether the key exists at the point of storage.

For example, if the key `dlcounter` does not exist, the following fragment will return 1000:

```
long newcount =
    client.decr("dlcount",1,1000);

System.out.printf("Updated counter is %d\n",newcount);
```

A subsequent identical call will return the value 999 as the key `dlcount` already exists.

API Call	`client.decr(key, offset, default, expiry)`
Asynchronous	no
Description	Decrement the value of an existing numeric key. The Couchbase Server stores numbers as unsigned values. Therefore the lowest you can decrement is to zero.
Returns	`long` (Numeric value)
Arguments	
`String key`	Document ID used to identify the value
`int offset`	Integer offset value to increment/decrement (default 1)
`int default`	Default value to increment/decrement if key does not exist
`int expiry`	Expiry time for key. Values larger than 30*24*60*60 seconds (30 days) are interpreted as absolute times (from the epoch).

The third form of the `decr()` method the decrement operation, with a default value and with the addition of setting an expiry time on the stored value. For example, to decrement a counter, using a default of 1000 if the value does not exist, and an expiry of 1 hour (3600 seconds):

```
long newcount =
    client.decr("dlcount",1,1000,3600);
```

For information on expiry values, see Section 4.4, “Expiry Values”.

API Call	`client.asyncDecr(key, offset)`
Asynchronous	yes
Description	Decrement the value of an existing numeric key. The Couchbase Server stores numbers as unsigned values. Therefore the lowest you can decrement is to zero.

Returns	`long` (Numeric value)
Arguments	
`String key`	Document ID used to identify the value
`int offset`	Integer offset value to increment/decrement (default 1)

The asynchronous form of the `asyncDecr()` method enables you to decrement a value without waiting for a response. This can be useful in situations where you do not need to know the updated value or merely want to perform a decrement and continue processing.

For example, to asynchronously decrement a given key:

```
OperationFuture<Long> decrOp =
    client.asyncDecr("samplekey",1,1000,24000);
```

8.6. Increment Methods

The increment methods enable you to increase a given stored integer value. These are the incremental equivalent of the decrement operations and work on the same basis; updating the value of a key if it can be parsed to an integer. The update operation occurs on the server and is provided at the protocol level. This simplifies what would otherwise be a two-stage get and set operation.

API Call	`client.incr(key, offset)`
Asynchronous	no
Description	Increment the value of an existing numeric key. Couchbase Server stores numbers as unsigned numbers, therefore if you try to increment an existing negative number, it will cause an integer overflow and return a non-logical numeric result. If a key does not exist, this method will initialize it with the zero or a specified value.
Returns	`long` (Numeric value)
Arguments	
`String key`	Document ID used to identify the value
`int offset`	Integer offset value to increment/decrement (default 1)

The first form of the `incr()` method accepts the keyname and offset (increment) value to be used when increasing the server-side integer. For example, to increment the server integer `dlcounter` by 5:

```
client.incr("dlcounter",5);
```

The return value is the updated value of the specified key.

API Call	`client.incr(key, offset, default)`
Asynchronous	no
Description	Increment the value of an existing numeric key. Couchbase Server stores numbers as unsigned numbers, therefore if you try to increment an existing negative number, it will cause an integer overflow and return a non-logical numeric result. If a key does not exist, this method will initialize it with the zero or a specified value.
Returns	`long` (Numeric value)
Arguments	
`String key`	Document ID used to identify the value
`int offset`	Integer offset value to increment/decrement (default 1)

`int default`	Default value to increment/decrement if key does not exist

The second form of the `incr()` method supports the use of a default value which will be used to set the corresponding key if that value does not already exist in the database. If the key exists, the default value is ignored and the value is incremented with the provided offset value. This can be used in situations where you are recording a counter value but do not know whether the key exists at the point of storage.

For example, if the key `dlcounter` does not exist, the following fragment will return 1000:

```
long newcount =
    client.incr("dlcount",1,1000);

System.out.printf("Updated counter is %d\n",newcount);
```

A subsequent identical call will return the value 1001 as the key `dlcount` already exists and the value (1000) is incremented by 1.

API Call	`client.incr(key, offset, default, expiry)`
Asynchronous	no
Description	Increment the value of an existing numeric key. Couchbase Server stores numbers as unsigned numbers, therefore if you try to increment an existing negative number, it will cause an integer overflow and return a non-logical numeric result. If a key does not exist, this method will initialize it with the zero or a specified value.
Returns	`long` (Numeric value)
Arguments	
`String key`	Document ID used to identify the value
`int offset`	Integer offset value to increment/decrement (default 1)
`int default`	Default value to increment/decrement if key does not exist
`int expiry`	Expiry time for key. Values larger than 30*24*60*60 seconds (30 days) are interpreted as absolute times (from the epoch).

The third format of the `incr()` method supports setting an expiry value on the given key, in addition to a default value if key does not already exist.

For example, to increment a counter, using a default of 1000 if the value does not exist, and an expiry of 1 hour (3600 seconds):

```
long newcount =
    client.incr("dlcount",1,1000,3600);
```

For information on expiry values, see Section 4.4, “Expiry Values”.

API Call	`client.asyncIncr(key, offset)`
Asynchronous	yes
Description	Increment the value of an existing numeric key. Couchbase Server stores numbers as unsigned numbers, therefore if you try to increment an existing negative number, it will cause an integer overflow and return a non-logical numeric result. If a key does not exist, this method will initialize it with the zero or a specified value.
Returns	`Future<Long>` (Asynchronous request value, as Long)
Arguments	
`String key`	Document ID used to identify the value

`int offset`	Integer offset value to increment/decrement (default 1)

The `asyncIncr()` method supports an asynchronous increment on the value for a corresponding key. Asynchronous increments are useful when you do not want to immediately wait for the return value of the increment operation.

```
OperationFuture<Long> incrOp =
    client.asyncIncr("samplekey",1,1000,24000);
```

8.7. Replace Methods

The `replace()` methods update an existing key/value pair in the database. If the specified key does not exist, then the operation will fail.

API Call	`client.replace(key, expiry, value)`
Asynchronous	yes
Description	Update an existing key with a new value
Returns	`Future<Boolean>` (Asynchronous request value, as Boolean)
Arguments	
`String key`	Document ID used to identify the value
`int expiry`	Expiry time for key. Values larger than 30*24*60*60 seconds (30 days) are interpreted as absolute times (from the epoch).
`Object value`	Value to be stored

The first form of the `replace()` method updates an existing value setting while supporting the explicit setting of the expiry time on the item. For example to update the `samplekey`:

```
OperationFuture<Boolean> replaceOp =
    client.replace("samplekey","updatedvalue",0);
```

The return value is a `OperationFuture` value with a `Boolean` base.

API Call	`client.replace(key, expiry, value, transcoder)`
Asynchronous	yes
Description	Update an existing key with a new value
Returns	`Future<Boolean>` (Asynchronous request value, as Boolean)
Arguments	
`String key`	Document ID used to identify the value
`int expiry`	Expiry time for key. Values larger than 30*24*60*60 seconds (30 days) are interpreted as absolute times (from the epoch).
`Object value`	Value to be stored
`Transcoder<T> transcoder`	Transcoder class to be used to serialize value

The second form of the `replace()` method is identical o the first, but also supports using a custom Transcoder in place of the default transcoder.

8.8. Touch Methods

The `touch()` methods allow you to update the expiration time on a given key. This can be useful for situations where you want to prevent an item from expiring without resetting the associated value. For example, for a session database you

might want to keep the session alive in the database each time the user accesses a web page without explicitly updating the session value, keeping the user's session active and available.

API Call	`client.touch(key, expiry)`
Asynchronous	yes
Description	Update the expiry time of an item
Returns	`Future<Boolean>` (Asynchronous request value, as Boolean)
Arguments	
`String key`	Document ID used to identify the value
`int expiry`	Expiry time for key. Values larger than 30*24*60*60 seconds (30 days) are interpreted as absolute times (from the epoch).

The first form of the `touch()` provides a simple key/expiry call to update the expiry time on a given key. For example, to update the expiry time on a session for another 5 minutes:

```
OperationFuture<Boolean> touchOp =
    client.touch("sessionid",300);
```

Chapter 9. Statistics Operations

The Couchbase Java Client Library includes support for obtaining statistic information from all of the servers defined within a `CouchbaseClient` object. A summary of the commands is provided below.

Table 9.1. Java Client Library Statistics Methods

Method	Title
`client.getStats()`	Get the statistics from all connections
`client.getStats(statname)`	Get the statistics from all connections

API Call	`client.getStats()`
Asynchronous	no
Description	Get the database statistics
Returns	`Object` (Binary object)
Arguments	
	None

The first form of the `getStats()` command gets the statistics from all of the servers configured in your `CouchbaseClient` object. The information is returned in the form of a nested Map, first containing the address of configured server, and then within each server the individual statistics for that server.

API Call	`client.getStats(statname)`
Asynchronous	no
Description	Get the database statistics
Returns	`Object` (Binary object)
Arguments	
`String statname`	Group name of a statistic for selecting individual statistic value

The second form of the `getStats()` command gets the specified group of statistics from all of the servers configured in your CouchbaseClient object. The information is returned in the form of a nested Map, first containing the address of configured server, and then within each server the individual statistics for that server.

Chapter 10. View/Query Interface

Couchbase Server 2.0 extends the querying mechanisms by not only allowding key-based lookups, but also allowing you to query your datasets through a flexible mechanism called views. Those views are based on a common data aggregation approach called map/reduce. With Couchbase Server 2.0 you are able to keep using all of the Couchbase code you already have, and upgrade certain parts of it to use JSON documents without any hassles. In doing this, you can easily add the power of Views and querying those views to your applications.

For more information about using views for indexing and querying from Couchbase Server, here are some useful resources:

- For more information on Views, how they operate, and how to write effective map/reduce queries, see Couchbase Server 2.0: Views and Couchbase Sever 2.0: Writing Views.
- Sample Patterns: to see examples and patterns you can use for views, see Couchbase Views, Sample Patterns.
- Timestamp Pattern: many developers frequently ask about extracting information based on date or time. To find out more, see Couchbase Views, Sample Patterns.

The `View` Object is obtained by calling the `getView` method which provides access to the view on the server.

API Call	`client.getView(ddocname, viewname)`
Asynchronous	no
Description	Create a view object to be used when querying a view.
Returns	(none)
Arguments	
`String ddocname`	Design document name
`String viewname`	View name within a design document

```
View view = client.getView(docName, viewName)
```

Then obtain a new `Query` object.

API Call	`Query.new()`
Asynchronous	no
Description	Create a query object to be used when querying a view.
Returns	(none)
Arguments	
	None

```
Query query = new Query();
```

Once, the View and Query objects are available, the results of the server view can be accessed as below.

API Call	`client.query(view, query)`
Asynchronous	no
Description	Query a view within a design doc
Returns	(none)
Arguments	

`View view`	View object associated with a server view
`Query query`	View object associated with a server view

```
ViewResponse = client.query(view, query);
```

Before accessing the View, a list of options can be set with the query object (here is a short list of the most commonly used ones).

- `setKey(java.lang.String key)`

  to set the starting Key.

- `setRangeStart(java.lang.String startKey)`

  to set the starting Key.

- `setRangeEnd(java.lang.String endKey)`

  to set the ending Key.

- `setRange(java.lang.String startKey, java.lang.String endKey)`

  to set the starting and ending key, both.

- `setDescending(boolean descending)`

  to set the descending flat to true or false.

- `setIncludeDocs(boolean include)`

  to Include the original JSON document with the query.

- `setReduce(boolean reduce)`

  where the reduce function is included or excluded based on the Flag.

- `setStale(Stale stale)`

  where the possible values for stale are `FALSE`, `UPDATE_AFTER` and `OK` as noted in the Release Notes.

The format of the returned information of the query method is:

`ViewResponse` or any of the other inherited objects such as `ViewResponseWithDocs`, `ViewResponseNoDocs`, `ViewResponseReduced`.

The `ViewResponse` method provides a `iterator()` method for iterating through the rows as a `ViewRow` interface. The `ViewResponse` method also provides a `getMap()` method where the result is available as a map.

The following methods are available on the `ViewRow` interface.

- `getId()`

  to get the Id of the associated row.

- `getKey()`

  to get the Key of the associated Key/Value pair of the result.

- `getValue()`

  to get the Value of the associated Key/Value pair of the result.

- `getDocument()`

  to get the document associated with the row.

For usage of these classes, please refer to the Chapter 2, *Tutorial* which has been enhanced to include Views.

Chapter 11. Java Troubleshooting

This Couchbase SDK Java provides a complete interface to Couchbase Server through the Java programming language. For more on Couchbase Server and Java read our Java SDK Getting Started Guide followed by our in-depth Couchbase and Java tutorial. We recommended Java SE 6 (or higher) for running the Couchbase Client Library.

This section covers the following topics:

- Logging from the Java SDK
- Handling Timeouts
- Bulk Load and Exponential Backoff
- Retrying After Receiving a Temporary Failure

11.1. Configuring Logging

Occasionally when you are troubleshooting an issue with a clustered deployment, you may find it helpful to use additional information from the Couchbase Java SDK logging. The SDK uses JDK logging and this can be configured by specifying a runtime define and adding some additional logging properties. There are two ways to set up Java SDK logging:

- Use spymemcached to log from the Java SDK. Since the SDK uses spymemcached and is compatible with spymemcached, you can use the logging provided to output SDK-level information.
- Set your JDK properties to log Couchbase Java SDK information.
- Provide logging from your application.

To provide logging via spymemcached:

```
System.setProperty("net.spy.log.LoggerImpl", "net.spy.memcached.compat.log.SunLogger");
```

or

```
System.setProperty("net.spy.log.LoggerImpl", "net.spy.memcached.compat.log.Log4JLogger");
```

The default logger simply logs everything to the standard error stream. To provide logging via the JDK, if you are running a command-line Java program, you can run the program with logging by setting a property:

```
-Djava.util.logging.config.file=logging.properties
```

The other alternative is create a `logging.properties` and add it to your in your classpath:

```
logging.properties
handlers = java.util.logging.ConsoleHandler
java.util.logging.ConsoleHandler.level = ALL
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter
com.couchbase.client.vbucket.level = FINEST
com.couchbase.client.vbucket.config.level = FINEST
com.couchbase.client.level = FINEST
```

The final option is to provide logging from your actual Java application. If you are writing your application in an IDE which manages command-line operations for you, it may be easier if you express logging in your application code. Here is an example:

```
// Tell things using spymemcached logging to use internal SunLogger API
        Properties systemProperties = System.getProperties();
        systemProperties.put("net.spy.log.LoggerImpl", "net.spy.memcached.compat.log.SunLogger");
        System.setProperties(systemProperties);

        Logger.getLogger("net.spy.memcached").setLevel(Level.FINEST);
        Logger.getLogger("com.couchbase.client").setLevel(Level.FINEST);
```

```
        Logger.getLogger("com.couchbase.client.vbucket").setLevel(Level.FINEST);

        //get the top Logger
        Logger topLogger = java.util.logging.Logger.getLogger("");

        // Handler for console (reuse it if it already exists)
        Handler consoleHandler = null;
        //see if there is already a console handler
        for (Handler handler : topLogger.getHandlers()) {
            if (handler instanceof ConsoleHandler) {
                //found the console handler
                consoleHandler = handler;
                break;
            }
        }

        if (consoleHandler == null) {
            //there was no console handler found, create a new one
            consoleHandler = new ConsoleHandler();
            topLogger.addHandler(consoleHandler);
        }

        //set the console handler to fine:
        consoleHandler.setLevel(java.util.logging.Level.FINEST);
```

11.2. Handling Timeouts

The Java client library has a set of synchronous and asynchronous methods. While it does not happen in most situations, occasionally network IO can become congested, nodes can fail, or memory pressure can lead to situations where an operation can timeout.

When a timeout occurs, most of the synchronous methods on the client will return a RuntimeException showing a timeout as the root cause. Since the asynchronous operations give more specific control over how long it takes for an operation to be successful or unsuccessful, asynchronous operations throw a checked TimeoutException.

As an application developer, it is best to think about what you would do after this timeout. This may be something such as showing the user a message, it may be doing nothing, or it may be going to some other system for additional data.

In some cases you might want to retry the operation, but you should consider this carefully before performing the retry in your code; sometimes a retry may exacerbate the underlying problem that caused the timeout. If you choose to do a retry, providing in the form of a backoff or exponential backoff is advisable. This can be thought of as a pressure relief valve for intermittent resource problems. For more information on backoff and exponential backoff, see Section 11.4, “Bulk Load and Exponential Backoff”.

11.3. Timing-out and Blocking

If your application creates a large number of asynchronous operations, you may also encounter timeouts immediately in response to the requests. When you perform an asynchronous operation, Couchbase Java SDK creates an object and puts the object into a request queue. The object and the request are stored in Java runtime memory, in other words, they are stored in local to your Java application runtime memory and require some amount of Java Virtual Machine IO to be serviced.

Rather than write so many asynchronous operations that can overwhelm a JVM and generate out of memory errors for the JVM, you can rely on SDK-level timeouts. The default behavior of the Java SDK is to start to immediately timeout asynchronous operations if the queue of operations to be sent to the server is overwhelmed.

You can also choose to control the volume of asynchronous requests that are issued by your application by setting a timeout for blocking. You might want to do this for a bulk load of data so that you do not overwhelm your JVM. The following is an example:

```
List<URI> baselist = new ArrayList<URI>();
        baselist.add(new URI("http://localhost:8091/pools"));
```

```
        CouchbaseConnectionFactoryBuilder cfb = new CouchbaseConnectionFactoryBuilder();
        cfb.setOpQueueMaxBlockTime(5000); // wait up to 5 seconds when trying to enqueue an operation

        CouchbaseClient myclient = new CouchbaseClient(cfb.buildCouchbaseConnection(baselist, "default", "default", "
```

11.4. Bulk Load and Exponential Backoff

When you bulk load data to Couchbase Server, you can accidentally overwhelm available memory in the Couchbase cluster before it can store data on disk. If this happens, Couchbase Server will immediately send a response indicating the operation cannot be handled at the moment but can be handled later.

This is sometimes referred to as "handling Temp OOM", where where OOM means out of memory. Note though that the actual temporary failure could be sent back for reasons other than OOM. However, temporary OOM is the most common underlying cause for this error.

To handle this problem, you could perform an exponential backoff as part of your bulk load. The backoff essentially reduces the number of requests sent to Couchbase Server as it receives OOM errors:

```
package com.couchbase.sample.dataloader;

import com.couchbase.client.CouchbaseClient;
import java.io.IOException;
import java.net.URI;
import java.util.List;
import net.spy.memcached.internal.OperationFuture;
import net.spy.memcached.ops.OperationStatus;

/**
 *
   * The StoreHandler exists mainly to abstract the need to store things
   * to the Couchbase Cluster even in environments where we may receive
   * temporary failures.
 *
 * @author ingenthr
 */
public class StoreHandler {

  CouchbaseClient cbc;
  private final List<URI> baselist;
  private final String bucketname;
  private final String password;

  /**
   *
   * Create a new StoreHandler.  This will not be ready until it's initialized
   * with the init() call.
   *
   * @param baselist
   * @param bucketname
   * @param password
   */
  public StoreHandler(List<URI> baselist, String bucketname, String password) {
    this.baselist = baselist; // TODO: maybe copy this?
    this.bucketname = bucketname;
    this.password = password;


  }

  /**
   * Initialize this StoreHandler.
   *
   * This will build the connections for the StoreHandler and prepare it
   * for use.  Initialization is separated from creation to ensure we would
   * not throw exceptions from the constructor.
   *
   *
   * @return StoreHandler
   * @throws IOException
   */
  public StoreHandler init() throws IOException {
    // I prefer to avoid exceptions from constructors, a legacy we're kind
```

```
    // of stuck with, so wrapped here
    cbc = new CouchbaseClient(baselist, bucketname, password);
    return this;
  }

  /**
   *
   * Perform a regular, asynchronous set.
   *
   * @param key
   * @param exp
   * @param value
   * @return the OperationFuture<Boolean> that wraps this set operation
   */
  public OperationFuture<Boolean> set(String key, int exp, Object value) {
    return cbc.set(key, exp, cbc);
  }

  /**
   * Continuously try a set with exponential backoff until number of tries or
   * successful.  The exponential backoff will wait a maximum of 1 second, or
   * whatever
   *
   * @param key
   * @param exp
   * @param value
   * @param tries number of tries before giving up
   * @return the OperationFuture<Boolean> that wraps this set operation
   */
  public OperationFuture<Boolean> contSet(String key, int exp, Object value,
          int tries) {
    OperationFuture<Boolean> result = null;
    OperationStatus status;
    int backoffexp = 0;

    try {
      do {
        if (backoffexp > tries) {
          throw new RuntimeException("Could not perform a set after "
                  + tries + " tries.");
        }
        result = cbc.set(key, exp, value);
        status = result.getStatus(); // blocking call, improve if needed
        if (status.isSuccess()) {
          break;
        }
        if (backoffexp > 0) {
          double backoffMillis = Math.pow(2, backoffexp);
          backoffMillis = Math.min(1000, backoffMillis); // 1 sec max
          Thread.sleep((int) backoffMillis);
          System.err.println("Backing off, tries so far: " + backoffexp);
        }
        backoffexp++;

        if (!status.isSuccess()) {
          System.err.println("Failed with status: " + status.getMessage());
        }

      } while (status.getMessage().equals("Temporary failure"));
    } catch (InterruptedException ex) {
      System.err.println("Interrupted while trying to set.  Exception:"
              + ex.getMessage());
    }

    if (result == null) {
      throw new RuntimeException("Could not carry out operation."); // rare
    }

    // note that other failure cases fall through.  status.isSuccess() can be
    // checked for success or failure or the message can be retrieved.
    return result;
  }
}
```

There is also a setting you can provide at the connection-level for Couchbase Java SDK that will also help you avoid too many asynchronous requests:

```
List<URI> baselist = new ArrayList<URI>();
        baselist.add(new URI("http://localhost:8091/pools"));

        CouchbaseConnectionFactoryBuilder cfb = new CouchbaseConnectionFactoryBuilder();
        cfb.setOpTimeout(10000);  // wait up to 10 seconds for an operation to succeed
        cfb.setOpQueueMaxBlockTime(5000); // wait up to 5 seconds when trying to enqueue an operation

        CouchbaseClient myclient = new CouchbaseClient(cfb.buildCouchbaseConnection(baselist, "default", "default", "
```

11.5. Retrying After Receiving a Temporary Failure

If you send too many requests all at once to Couchbase, you can create a out of memory problem, and the server will send back a temporary failure message. The message indicates you can retry the operation, however the server will not slow down significantly; it just does not handle the request. In contrast, other database systems will become slower for all operations under load.

This gives your application a bit more control since the temporary failure messages gives you the opportunity to provide a backoff mechanism and retry operations in your application logic.

11.6. Java Virtual Machine Tuning Guidelines

Generally speaking, there is no reason to adjust any Java Virtual Machine parameters when using the Couchbase Java Client. In fact, in general you should not start with specific tuning, but instead should use defaults from the application server first, then measure application metrics such as throughput and response time. Then, if there is a need to make an improvement, make adjustments and re-measure.

The recommendations here are based on the Oracle (formerly Sun) HotSpot Virtual Machine and derivations such as the Java Virtual Machine shipped with Mac OS X and the OpenJDK project. Other Java virtual machines likely behave similarly.

It should be noted that by default, garbage collection times may easily go over 1sec. This can lead to higher than expected response times or even timeouts, as the default timeout is 2.5 seconds. This is true with simple tests even on systems with lots of CPUs and a good amount of memory.

The reason for this is that for the most part, by default, the JVM is weighted toward throughput instead of latency. Of course, much of this can be controlled with GC tuning on the JVM. With the hotspot JVM, look to this whitepaper: http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf

In the referenced whitepaper, the Concurrent Mark Sweep collector is recommended if your applciation needs short pauses. It also recommends advising the JVM to try to shorten pause times. Given the Couchbase client's 2.5 second default timeout, with our basic testing we found the following to be useful:

```
-XX:+UseConcMarkSweepGC -XX:MaxGCPauseMillis=850
```

The whitepaper refers to a couple of tools which may be useful in gathering information on JVM GC performance. For example, adding -XX:+PrintGCDetails and -XX:+PrintGCTimeStamps are a simple way to generate log messages which you may correlate to application behavior. The logs may show a full GC event taking, perhaps, several seconds during which no processing occurs and operations may timeout. Adjusting parameters related to how to perform a full GC, which collector to use, how long to pause the VM during GC and even adding incremental mode may help, depending on your application's workload. One other common tool for getting information is JConsole (http://docs.oracle.com/javase/6/docs/technotes/guides/management/jconsole.html). JConsole is more of an interactive tool, but it may help you identify changes you may want to make in the different memory spaces used by the JVM to further reduce the need to run a GC on the old generation.

There is a CPU time tradeoff when setting these tuning parameters. There are also other parameters which may provide additional help referenced in the whitepaper.

If you happen to be using JDK 7 update 4 or later, the G1 collector may be an even better option. Again, you should be guided by measuring performance from the application level.

Even with these, our testing showed some GCs near a half a second. While the Couchbase Client allows tuning of the timeout time to drop as low as you wish, we do not recommend dropping it much below one second unless you are planning to tune other parts of the system beyond the JVM.

For example, most people run applications on networks that do not offer any guarantee around response time. If the network is oversubscribed or minor blips occur on the network, there can be TCP retransmissions. While many TCP implementations may ignore it, RFC 2988 specifies rounding up to 1sec when calculating TCP retransmit timeouts.

Achieving either maximum throughput or minimum per-operation latency can be enhanced with JVM tuning, supported by overall system tuning at the extremes.

Appendix A. Release Notes

The following sections provide release notes for individual release versions of Couchbase Client Library Java. To browse or submit new issues, see Couchbase Client Library Java Issues Tracker.

Appendix B. Licenses

This documentation and associated software is subject to the following licenses.

B.1. Documentation License

This documentation in any form, software or printed matter, contains proprietary information that is the exclusive property of Couchbase. Your access to and use of this material is subject to the terms and conditions of your Couchbase Software License and Service Agreement, which has been executed and with which you agree to comply. This document and information contained herein may not be disclosed, copied, reproduced, or distributed to anyone outside Couchbase without prior written consent of Couchbase or as specifically provided below. This document is not part of your license agreement nor can it be incorporated into any contractual agreement with Couchbase or its subsidiaries or affiliates.

Use of this documentation is subject to the following terms:

You may create a printed copy of this documentation solely for your own personal use. Conversion to other formats is allowed as long as the actual content is not altered or edited in any way. You shall not publish or distribute this documentation in any form or on any media, except if you distribute the documentation in a manner similar to how Couchbase disseminates it (that is, electronically for download on a Web site with the software) or on a CD-ROM or similar medium, provided however that the documentation is disseminated together with the software on the same medium. Any other use, such as any dissemination of printed copies or use of this documentation, in whole or in part, in another publication, requires the prior written consent from an authorized representative of Couchbase. Couchbase and/or its affiliates reserve any and all rights to this documentation not expressly granted above.

This documentation may provide access to or information on content, products, and services from third parties. Couchbase Inc. and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Couchbase Inc. and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.