CouchDB Project

Luke Foley  
T00224345

Table of Contents

[Importing an Open Data Set 4](#_Toc180769867)

[Dataset Used 4](#_Toc180769868)

[Steps Taken 4](#_Toc180769869)

[Dataset Structure 8](#_Toc180769870)

[CRUD with Postman 8](#_Toc180769871)

[Adding a new document to the DB with POST – Shawshank Redemption: 10](#_Toc180769872)

[Reading the document that was just added with GET 12](#_Toc180769873)

[Updating the document with PUT – setting the address to the document ID, and updating some of the fields: 13](#_Toc180769874)

[Deleting a document using DELETE – deleting revision 2 of the Shawshank redemption document: 15](#_Toc180769875)

[MapReduce: 16](#_Toc180769876)

[Finding a movie (individual document) by move name: 16](#_Toc180769877)

[Find documents in a range: 17](#_Toc180769878)

[\_sum function to get total gross for the top 1000 imdb movies: 20](#_Toc180769879)

[\_count function to count number of movies: 22](#_Toc180769880)

[\_stats of ratings: 24](#_Toc180769881)

[Finding gross revenue by year and genre: 26](#_Toc180769882)

[Using Mango: 29](#_Toc180769883)

[Finding a particular document: 29](#_Toc180769884)

[Documents in Range 29](#_Toc180769885)

[Advantages/disadvantages of Mango/MapReduce 30](#_Toc180769886)

[CouchDB Clients for Python 36](#_Toc180769887)

[CouchDB-Python Client 36](#_Toc180769888)

[Functionality 36](#_Toc180769889)

[Usability 36](#_Toc180769890)

[Performance 36](#_Toc180769891)

[Support 36](#_Toc180769892)

[PyCouchDB Client 36](#_Toc180769893)

[Functionality 36](#_Toc180769894)

[Usability 37](#_Toc180769895)

[Performance 37](#_Toc180769896)

[Support 38](#_Toc180769897)

[Chosen Client 38](#_Toc180769898)

[Recommendation: 38](#_Toc180769899)

[GET Test with CouchDB-Python Client 39](#_Toc180769900)

[CouchDB Replication Protocol 40](#_Toc180769901)

[Crud App 40](#_Toc180769902)

[DBAAS 40](#_Toc180769903)

[Cloudant 40](#_Toc180769904)

# Importing an Open Data Set to CouchDB

## Dataset Used

For this project the following dataset was used: <https://www.kaggle.com/datasets/harshitshankhdhar/imdb-dataset-of-top-1000-movies-and-tv-shows?resource=download>

This is a dataset of the top 1000 movies on IMDB.

## Steps Taken

A screenshot of a computer

Description automatically generatedFirstly, downloaded and setup CouchDB. Then from the Project Fauxton interface, created a new database called movies:

The dataset downloaded from Kaggle was in CSV format, however, CouchDB requires data in JSON format for importing into its database. In order to convert the dataset to JSON, a simple Python script utilising the pandas library was used from a Jupyter Notebook in PyCharm:

A screen shot of a computer code

Description automatically generated

The pandas library made the conversion process very easy, producing the following file:

A screen shot of a computer

Description automatically generatedA screen shot of a computer program

Description automatically generated

I then attempted to import the new JSON dataset using the bulk import feature in curl. The command resulted in the following error:

A black background with white text

Description automatically generated

This was due to the file not being formatted entirely correctly. The documents in the dataset need to be contained in a “{“docs”:[]} tag. This was appended to the file manually:

A screenshot of a computer program

Description automatically generated

Rerunning the command with the correct tag appended, the import was successful. 1000 documents were added to the database:

A screenshot of a computer

Description automatically generated  
A screenshot of a computer

Description automatically generated

## Dataset Structure

A screenshot of a computer

Description automatically generated

# CRUD with Postman

After installing Postman, authentication details of the CouchDB database were added:

A screenshot of a computer

Description automatically generated

## Create

Adding a new movie document to the DB with POST – Shawshank Redemption:

A screenshot of a computer

Description automatically generated

Success Code 201. Success message with new ID and REV. The number of documents increased to 1001:

A screenshot of a computer

Description automatically generated

## Read

Reading the movie document that was just added, Shawshank Redemption, with GET. Adding the document ID to the address:

A screenshot of a computer

Description automatically generated

Success code 200. The movie details of Shawshank Redemption are returned.

## Update

Updating the document with PUT. Setting the address to the document ID. Changing some of the fields in the request body:

A screenshot of a computer

Description automatically generated

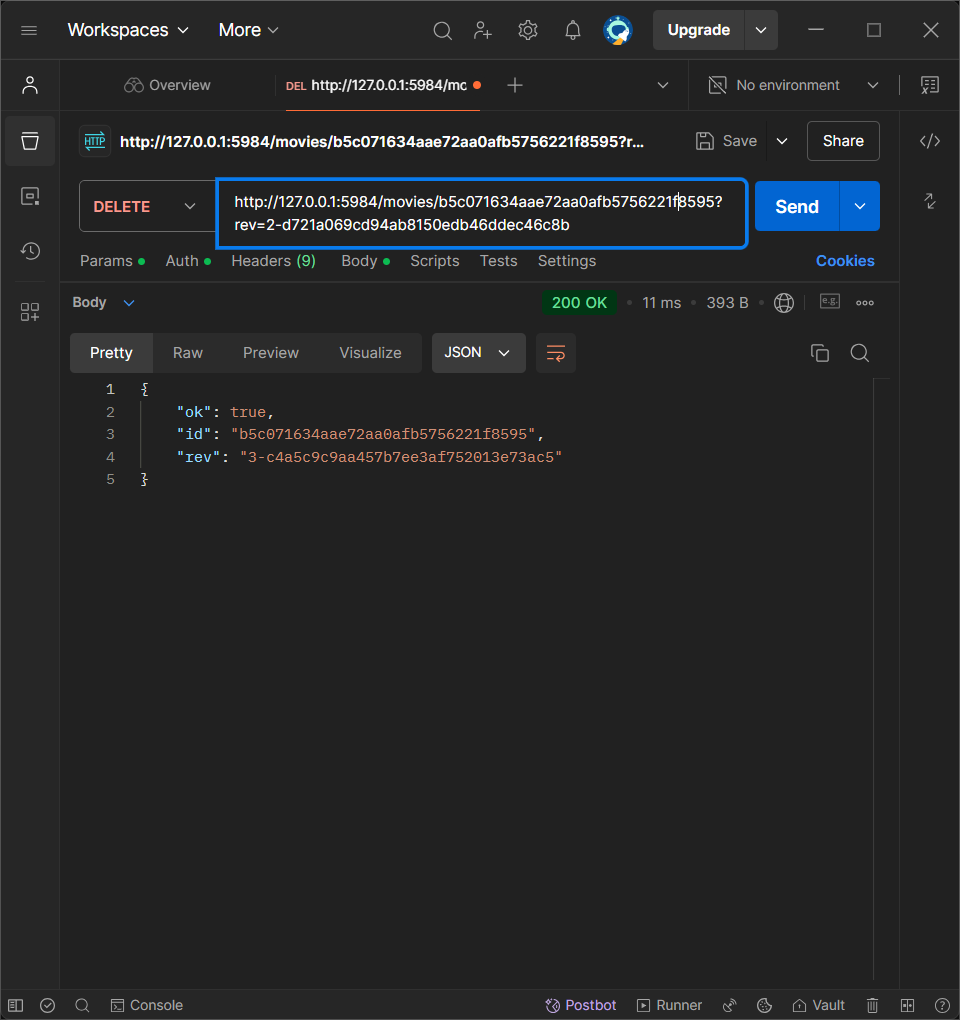
Success code 201, with a new revision for the document:

A screenshot of a computer

Description automatically generated

## Delete

Deleting the Shawshank Redemption document with a DELETE request. Address containing both the ID and the REV of the document:



Success code 200. A confirmation message is returned with a new revision.

# MapReduce:

## Finding a Document

Finding a movie by title:

**A screenshot of a computer

Description automatically generated**

This function checks each document for Series\_Title, and emits each document the matches the queried title.

Example in Postman: Get request with link to the view, and a key of “The Shawshank Redemption”

A screenshot of a computer

Description automatically generated

## Find Documents in Range:

Firstly, created a function to return each document by rating:

A screenshot of a computer

Description automatically generated

Testing by finding movies in a range of ratings between 9 and 10 in Postman – GET request with link to the view and a startkey of 9, and an endkey of 10:

A screenshot of a computer

Description automatically generated

The result shows a total of 1000 and an offset of 995, which means that the query resulted with 5 documents, or movies, with ratings between 9 and 10.

## \_sum Function

The \_sum reduce function adds up all the values emitted by the map function.

Example – This function emits the gross revenue values for the movies and adds them together with the \_sum reducer:



Postman test for total gross revenue:



## \_count Function

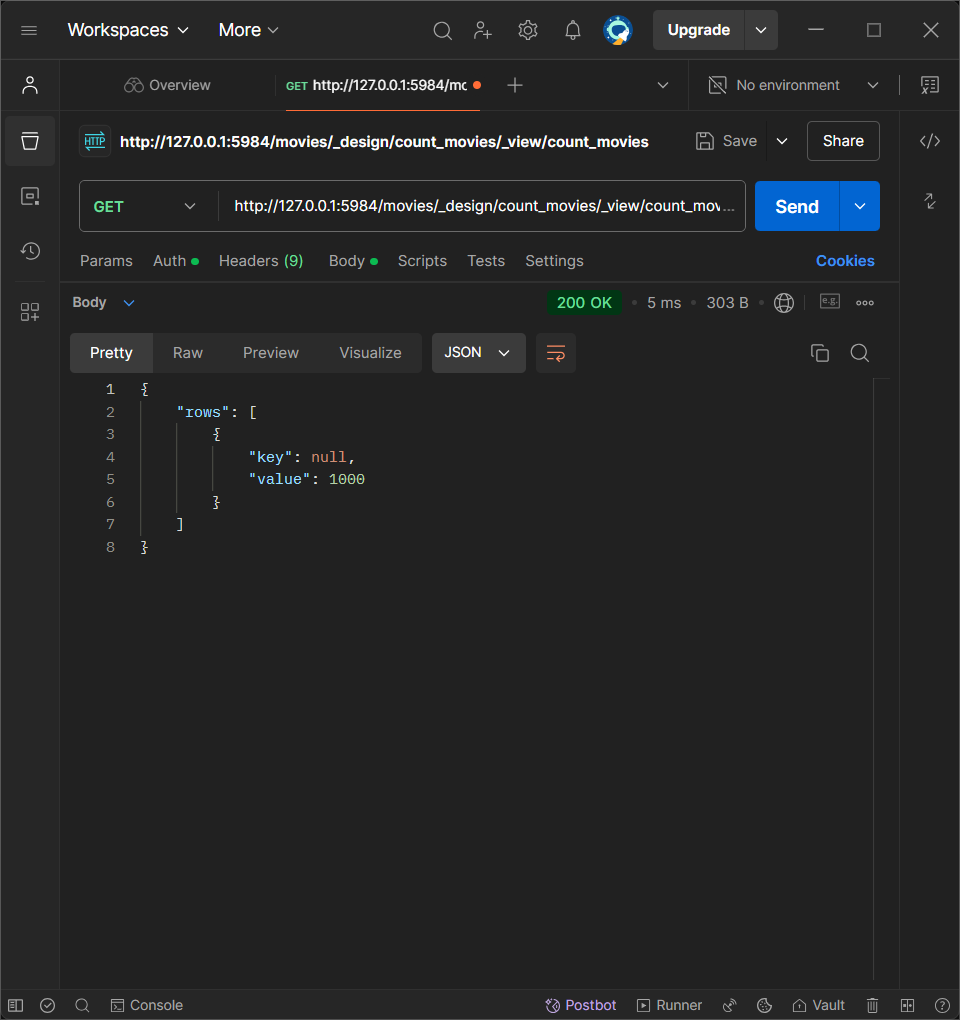
The \_count reduce function counts the number of documents emitted by the map function.

Example – This function emits every document ID in the database, and gets counted with the \_count reducer:

A screenshot of a computer

Description automatically generated

Postman test returns a count of 1000:



## \_stats Function

The \_stats reduce function provides statistical information about the values emitted by the map function. The stats provided are the sum, count, min, max, and the sum of squares.

Example – This function emits the ratings for all of the movies, and provides the sum, count, min, max and sum of squares for all of the ratings with the \_stats reducer:

A screenshot of a computer

Description automatically generated

Postman test:

A screenshot of a computer

Description automatically generated

## Finding Gross Revenue by Year and Genre:

In order to calculate the total gross revenue for movies grouped by year and by genre, year and genre must be emitted as a composite key in the map function, along with the gross revenue value. A \_sum reducer is also needed to provide the total gross value.

A screenshot of a computer

Description automatically generated

Postman test for total gross revenue by year (group level 1):

A screenshot of a computer

Description automatically generated

Postman test for total gross revenue by genre (group level 2):

A screenshot of a computer

Description automatically generated

# Using Mango:

## Finding a particular document:

This query retrieves all documents where “Series\_Title” equals “The Dark Knight”

A screenshot of a computer

Description automatically generated

## Documents in Range

This query retrieves all documents where “IMDB\_Rating” is greater than or equal to 9 (“$gte”: 9) and less than or equal to 10 (“$lte”: 9)

A screenshot of a computer

Description automatically generated

# Advantages/disadvantages of Mango/MapReduce

When working with \*\*CouchDB\*\*, there are two primary ways to query data: \*\*MapReduce\*\* and \*\*Mango\*\*. Both have their strengths and weaknesses, depending on the complexity of the data, the type of operations needed, and the overall structure of the application.

Let’s break down the \*\*advantages\*\* and \*\*disadvantages\*\* of building queries using \*\*MapReduce\*\* and \*\*Mango\*\*, with examples to help illustrate their differences.

---

## 1. \*\*MapReduce Queries in CouchDB\*\*

### \*\*What is MapReduce?\*\*

MapReduce is a programming model used in CouchDB to process large datasets in a distributed manner. It consists of:

- \*\*Map\*\*: A function that emits key-value pairs from each document.

- \*\*Reduce\*\*: A function that aggregates or summarizes the emitted key-value pairs (optional).

### \*\*Advantages of MapReduce\*\*

#### 1. \*\*Flexible Aggregation\*\*:

- \*\*Example\*\*: If you need to perform complex aggregations, such as summing up the gross revenue of movies by year and genre, MapReduce provides powerful mechanisms to aggregate and transform data.

- \*\*Powerful Reduce Functions\*\*: Functions like \*\*\_sum\*\*, \*\*\_count\*\*, and \*\*\_stats\*\* make it easier to summarize large datasets.

- \*\*Custom Reduce Functions\*\*: You can write custom JavaScript functions to perform complex calculations (e.g., calculating average IMDB ratings per genre).

#### 2. \*\*Efficiency in Handling Large Datasets\*\*:

- \*\*Incremental Updates\*\*: MapReduce views are calculated incrementally, meaning that once a view is created, CouchDB doesn’t need to recalculate everything from scratch when new documents are added. Only new documents are processed, making querying large datasets more efficient.

- \*\*Example\*\*: If you have a large dataset of movies, CouchDB processes and stores results for each movie as it’s added. Future queries become faster since the data doesn’t need to be recalculated each time.

#### 3. \*\*Customizable Keys for Complex Queries\*\*:

- \*\*Example\*\*: You can emit composite keys like `[year, genre]` and query the dataset with different levels of grouping. This makes it possible to create highly customized views of the data.

- \*\*Group and Group Level\*\*: You can group results by partial keys using the `group` and `group\_level` parameters, which are useful for hierarchical data.

### \*\*Disadvantages of MapReduce\*\*

#### 1. \*\*Complexity\*\*:

- \*\*Learning Curve\*\*: Writing MapReduce functions requires knowledge of JavaScript and CouchDB’s query model, which can be daunting for developers not familiar with functional programming.

- \*\*Example\*\*: A simple query like finding all movies released between 2000 and 2010 requires writing a map function and understanding how to structure queries based on emitted keys.

#### 2. \*\*Lack of Flexibility for Ad-Hoc Queries\*\*:

- \*\*Predefined Views\*\*: MapReduce views must be predefined, which limits the ability to perform ad-hoc queries without creating new views.

- \*\*Example\*\*: If you need to find all movies with a runtime of more than 150 minutes, but your MapReduce view only emits year and genre, you’ll have to create a new view or modify the existing one. This isn't efficient for one-time or infrequent queries.

#### 3. \*\*Slower for Simple Queries\*\*:

- \*\*Example\*\*: If you only need to find a specific document by ID or field (like `Series\_Title`), creating a MapReduce view just for that purpose is overkill and can be slower than simply using Mango.

- \*\*View Indexing\*\*: MapReduce views are only efficient after they’ve been created and indexed. For simple queries or small datasets, the overhead of building and maintaining views might not be justified.

---

## 2. \*\*Mango Queries in CouchDB\*\*

### \*\*What is Mango?\*\*

Mango is CouchDB’s declarative query language that allows you to write queries similar to MongoDB. It is simpler and more accessible than MapReduce for querying documents based on field values.

### \*\*Advantages of Mango\*\*

#### 1. \*\*Ease of Use and Simplicity\*\*:

- \*\*Example\*\*: Finding a particular movie by title is straightforward with Mango. The query looks like:

```json

{

"selector": {

"Series\_Title": "The Dark Knight"

}

}

```

- \*\*No JavaScript Knowledge Required\*\*: Unlike MapReduce, Mango doesn’t require you to write functions. Queries are written in JSON, making it much easier for developers familiar with JSON and REST APIs.

#### 2. \*\*Perfect for Ad-Hoc Queries\*\*:

- \*\*On-the-Fly Queries\*\*: Mango is perfect for one-time or ad-hoc queries that don’t require pre-defined views. This is especially useful when you’re exploring the data or performing queries that aren’t repetitive.

- \*\*Example\*\*: If you want to search for movies released between 2000 and 2010, you can just write a query:

```json

{

"selector": {

"Released\_Year": {

"$gte": "2000",

"$lte": "2010"

}

}

}

```

- \*\*No Index Required Initially\*\*: CouchDB will execute the query without requiring predefined views (although creating an index later can improve performance).

#### 3. \*\*Flexible Query Language\*\*:

- \*\*Rich Operators\*\*: Mango supports a variety of operators like `$eq`, `$gt`, `$lt`, `$in`, `$exists`, etc., for more flexible querying.

- \*\*Example\*\*: You can query for all movies with IMDB ratings greater than 8 using:

```json

{

"selector": {

"IMDB\_Rating": {

"$gt": 8

}

}

}

```

#### 4. \*\*Better for Simple Document Retrieval\*\*:

- \*\*Example\*\*: Finding a single document by ID or a specific field is much simpler and faster in Mango than MapReduce.

- \*\*Querying Across Multiple Fields\*\*: Mango allows you to easily filter based on multiple fields, like finding all action movies released between 2000 and 2010 with an IMDB rating greater than 8.

### \*\*Disadvantages of Mango\*\*

#### 1. \*\*Less Efficient for Aggregations\*\*:

- \*\*Limited Aggregation\*\*: Mango doesn’t provide built-in support for advanced aggregations like MapReduce. If you need to calculate sums, counts, or other aggregations, you would need to use MapReduce.

- \*\*Example\*\*: Summing the total gross revenue for each genre cannot be done directly with Mango. You would need to write a MapReduce view for this.

#### 2. \*\*Performance for Large Datasets\*\*:

- \*\*Indexes Needed for Speed\*\*: Mango queries can become slow with large datasets unless indexes are created. While CouchDB can run queries without indexes, it’s not efficient, and performance can degrade with large datasets.

- \*\*Example\*\*: If you frequently query for movies released between 2000 and 2010, you would need to create an index on the `Released\_Year` field to improve query performance.

#### 3. \*\*No Built-In Support for Grouping\*\*:

- \*\*Lack of Grouping\*\*: Mango doesn’t support grouping of data like MapReduce does with `group` and `group\_level`. For example, grouping data by year and genre would require MapReduce.

- \*\*Example\*\*: If you need to calculate the average IMDB rating per genre, Mango won’t be able to handle this easily. MapReduce is required for such operations.

---

### \*\*Conclusion\*\*

| Feature | \*\*MapReduce\*\* | \*\*Mango\*\* |

|--------------------------|-------------------------------------------------------|---------------------------------------------------|

| \*\*Aggregation\*\* | Advanced aggregation via custom reduce functions | No built-in aggregation |

| \*\*Ease of Use\*\* | Requires JavaScript and CouchDB knowledge | Simple, JSON-based queries |

| \*\*Performance\*\* | Efficient for large datasets with pre-defined views | Requires indexes for large datasets |

| \*\*Ad-Hoc Queries\*\* | Not suitable for on-the-fly or exploratory queries | Excellent for on-the-fly queries |

| \*\*Flexibility\*\* | Complex composite keys and grouping (via group\_level) | Flexible for simple document retrieval |

| \*\*Use Case\*\* | Best for complex aggregations, analytics | Best for simple queries and document retrieval |

### Recommendation:

- \*\*Use MapReduce\*\* if you need to perform \*\*complex aggregations\*\*, like summing or counting over large datasets, or if you need \*\*custom logic\*\* in your queries.

- \*\*Use Mango\*\* if you need to run \*\*simple queries\*\* or \*\*ad-hoc searches\*\* based on document fields, or if you are doing exploratory analysis and don’t want to define views upfront.

Let me know if you need further clarification on any of these!

# CouchDB Clients for Python

## CouchDB-Python Client

### Functionality

This client supports basic CRUD operations for documents.

It offers full support for querying CouchDB’s MapReduce views.

Users can execute Mango queries via the \_find endpoint, which is excellent for ad-hoc searches.

### Usability

The library can be easily installed using pip, and the API is user-friendly for connecting to the CouchDB server and accessing databases.

Connecting to the server and executing queries, such as retrieving documents, is straightforward and minimal setup is needed. For example, retrieving all documents in a database is as simple as:

import couchdb

server\_url = r"http://admin:mtu1234@127.0.0.1:5984/"

db = couchdb.Server(server\_url)['movies']

for doc\_id in db:

print(db[doc\_id])

### Performance

The client is efficient and lightweight, making it suitable for simple tasks such as fetching documents and executing queries. CouchDB's built-in features, including view indexing and incremental MapReduce, enhance performance with larger datasets. It also supports batch uploads for documents, which is particularly useful for large datasets.

### Support

The CouchDB-Python client has extensive documentation, providing examples for tasks like CRUD operations, MapReduce queries, and error handling: <https://couchdb-python.readthedocs.io/en/stable/index.html>

With CouchDB-Python being the most popular client, there is also a reasonable amount of support on GitHub and StackOverflow.

## PyCouchDB Client

### Functionality

PyCouchDB supports all basic CRUD operations: creating, reading, updating, and deleting documents. It also allows interaction with CouchDB views, which are essentially MapReduce queries.

However, unlike CouchDB-Python, PyCouchDB does not natively support Mango queries. This could be a limitation if you needed to perform ad-hoc queries without pre-defining views.

PyCouchDB lacks advanced support for bulk document operations, which are essential for uploading large datasets efficiently. It is designed with simplicity in mind and does not natively support more complex CouchDB features such as replication or document conflict handling.

### Usability

PyCouchDB can be easily installed using pip.

PyCouchDB provides a straightforward interface for interacting with CouchDB’s RESTful API. For example, connecting and retrieving all documents from a database can be done easily:

import pycouchdb

server = pycouchdb.Server("http://admin:mtu1234@127.0.0.1:5984/")

db = server.database('movies')

for doc in db.all():

print(doc)

The client is lightweight and easy to use, making it ideal for simple interactions with CouchDB.

### Performance

PyCouchDB is highly effective for small to medium datasets due to its lightweight design. It excels in basic operations such as retrieving documents and creating new ones very quickly.

Unlike CouchDB-Python, it lacks specialised support for incremental view indexing and does not optimize query performance for larger datasets.

While the client performs well for small-scale projects or one-off queries, it lacks advanced performance features for larger datasets like as bulk document uploading and efficient incremental querying.

### Support

The documentation for the PyCouchDB client is somewhat limited compared to other clients, but it provides clear guidance on basic operations: <https://pycouchdb.readthedocs.io/en/latest/>. For more advanced use cases or troubleshooting, you may need to directly consult CouchDB's API or rely on trial and error, which is not the ideal approach.

As PyCouchDB is not as widely used as clients such as CouchDB-Python, there is less community support, fewer online resources and less support on forums like StackOverflow or GitHub.

## Chosen Client

### Recommendation:

Based on the evaluation of the CouchDB-Python client, I recommend using this client for the project because:

It is easy to set up and use, making it suitable for data scientists.

It provides full support for CouchDB features, including MapReduce and Mango queries.

It performs well for both small and medium-sized datasets and scales well with CouchDB’s built-in optimisations.

### GET Test with CouchDB-Python Client

Installing the CouchDB-Python client with pip:

A screenshot of a computer program

Description automatically generated

Printing documents:

A screenshot of a computer program

Description automatically generated

# CouchDB Replication Protocol

**1. Replication Process Overview**

Replication in CouchDB is essentially a process of copying documents and their revisions from a source database to a target database. The source and target databases can reside on the same server, different servers, or even different CouchDB clusters. The replication process can be continuous or one-time and is designed to be fault-tolerant, allowing it to resume from where it left off in case of interruptions.

**2. Types of Replication**

CouchDB supports three primary types of replication:

* **One-time Replication:** This type of replication occurs once and then stops. It is useful for making occasional copies of a database.
* **Continuous Replication:** Here, changes made to the source database are continuously replicated to the target database as they happen. This ensures that the target stays up-to-date with the source.
* **Filtered Replication:** This allows for replicating only specific documents based on predefined criteria or filters, such as documents matching a certain type or containing specific field values.

**3. Replication Protocol Steps**

The CouchDB replication protocol consists of several steps, which can be outlined as follows:

**a. Checkpointing**

* At the start, CouchDB establishes a checkpoint to keep track of where replication left off if it had been previously attempted. This step helps avoid reprocessing already replicated data.
* The checkpoint can be stored in both the source and target databases, allowing the replication process to resume accurately in case of failure.

**b. Document Differences Detection**

* CouchDB identifies the documents that need to be replicated by comparing the revisions of documents in the source and target databases.
* This process involves sending a list of revision IDs from the source to the target and then determining which revisions the target lacks.

**c. Fetching Missing Documents**

* Once the target identifies the missing revisions, it requests those specific document revisions from the source database.
* The source database then transmits the requested documents and attachments (if any) to the target.

**d. Applying Changes**

* The target database updates its own documents and revision history based on the data received from the source.
* CouchDB uses a multi-version concurrency control (MVCC) mechanism, allowing documents to have multiple versions and preventing conflicts during replication.
* In case of conflicts (when two different revisions of the same document exist), CouchDB will flag the document as conflicted but will not automatically resolve it. Application-specific logic can be used to resolve conflicts.

**e. Updating the Checkpoint**

* After applying changes, the checkpoint is updated to reflect the latest replicated revision, marking the completion of a replication cycle.

**4. Conflict Handling**

CouchDB's replication protocol includes mechanisms for handling conflicts that arise when documents are updated concurrently across different databases. It uses a deterministic conflict resolution algorithm, where the "winning" document revision is determined by the highest revision number. However, all conflicting revisions are preserved, allowing applications to implement custom conflict resolution if needed.

**5. Security and Authentication**

CouchDB supports various authentication mechanisms, including Basic Authentication, Cookie-based Authentication, and OAuth. During replication, credentials can be provided to ensure secure communication between the source and target databases.

**6. Use Cases of CouchDB Replication**

The replication protocol is particularly useful for:

* **Offline-first Applications:** Synchronizing data between a local instance (e.g., a mobile app) and a central CouchDB server when the device comes back online.
* **Distributed Systems:** Replicating databases across geographically dispersed data centers to enhance availability and performance.
* **Backup and Restore:** Creating regular backups by replicating to a different CouchDB instance or storage location.

**7. CouchDB 2.x and Beyond: Clustering and Advanced Replication**

With CouchDB 2.x, the replication protocol was extended to support clustering. In a CouchDB cluster, the replication process takes advantage of sharding and partitioning, where each shard of the database can replicate independently. This enables more efficient and parallelized replication in large-scale distributed environments.

## Crud App

A simple node.js express website was made for this section. The site consists of 3 static pages:

Index (Main Page):

A screenshot of a computer

Description automatically generated

Add Movie:

A screenshot of a computer

Description automatically generated

Update Movie:

A screenshot of a computer

Description automatically generated

## Database Sync

As the app was built with node.js, PouchDB could be installed easily using npm:



Initialising the local PouchDB database and the remote CouchDB database:



Syncing the local PouchDB database with the remote CouchDB database:

A computer screen with colorful text

Description automatically generated

## CRUD

### Create – Add Movie

The add movie page allows the user to fill out the form with movie details.

Once the form is submitted, the *addMovie* function sends a POST request on the */add* route with the movie details in JSON.

A computer screen shot of a program

Description automatically generated

The movie details are then added to the local PouchDB database with a POST request. The server syncs this change to the CouchDB database.

### Read

The index page displays a list of all movies stored in the database in a table.

On loading the page, the *getMovies* function is called. The *getMovies* function fetches all movies on the */movies* route, sending a GET request. This returns all of the documents on the local PouchDB database. A row is then created on the table for each document returned.



A screen shot of a computer program

Description automatically generated

A computer screen shot of text

Description automatically generated

http://localhost:3000/movies/

A screen shot of a computer screen

Description automatically generated

### Update

A screen shot of a computer program

Description automatically generatedThe update page allows a user to modify the details of a movie.

When the update movie page is loaded, the *loadMovieDetails* function is called. This sends a GET request on the */movies/:id* route. This request returns the movie’s details from the local PouchDB database. The form is then filled out with the movie's details.

A screen shot of a computer code

Description automatically generated

A screen shot of a computer program

Description automatically generated

Once the form is submitted, the *updateMovie* function is called.This sends a PUT request on the */update/:id* route with the movie details from the form in JSON.

A computer screen shot of a program code

Description automatically generated

A PUT request is then sent to the local PouchDB database with the movie details.

### Delete

A screen shot of a computer program

Description automatically generatedThe delete button calls the *deleteMovie* function. This function prompts the user to confirm deletion. Then, a GET request is made to retrieve the revision for the document as this is needed for deletion. Once the revision has been retrieved, a DELETE request is made on the */delete/:id* route.

A screen shot of a computer code

Description automatically generated

A remove request is then sent to the local PouchDB database, with the ID and the revision.

# DBAAS

## Cloudant

After setting up an IBM Cloudant account, I was able to launch the cloudant dashboard:

A screenshot of a computer

Description automatically generated

From the dashboard a movies database was created:

A screenshot of a computer

Description automatically generated