Couchbase

EBOOK

Couchbase Data Access Patterns Guide

Building With Conduit
Real World Implementation

By Matthew Groves and Laurent Doguin

Contents

INTRODUCTION

Couchbase is a JSON document multi-model database. JSON and Couchbase support a number of different approaches to modeling and accessing data. This flexibility means that it's hard to be prescriptive. Being able to respond to change is one of the key benefits of using a database like Couchbase. However, if you're just starting out, this can also be daunting. This guide is an introduction to real world data modeling in action.

WHAT IS CONDUIT?

This guide used the Conduit Real World project to learn-by-example how to think about data access patterns with Couchbase. Conduit is a public spec of a medium.com clone, used for training, instruction, experimentation, proof-of-concept, etc. Each backed endpoint is documented with an HTTP verb, URL, request body, response body, required fields, and other information. This document goes through each endpoint and discusses the data interactions to use with Couchbase to implement the requirements. Visit the RealWorld Documentation to see the spec in detail, especially before diving into this guide for the first time. This guide only focuses on the Couchbase data access and modeling portion of the implementation. The authors of this guide have built the Couchbase implementation with Conduit on .NET and Java.

The Conduit project has the following implementations that use Couchbase available:

- ASP.NET Core + Couchbase
- Java + Spring + Couchbase
- Node.js + Express + Couchbase

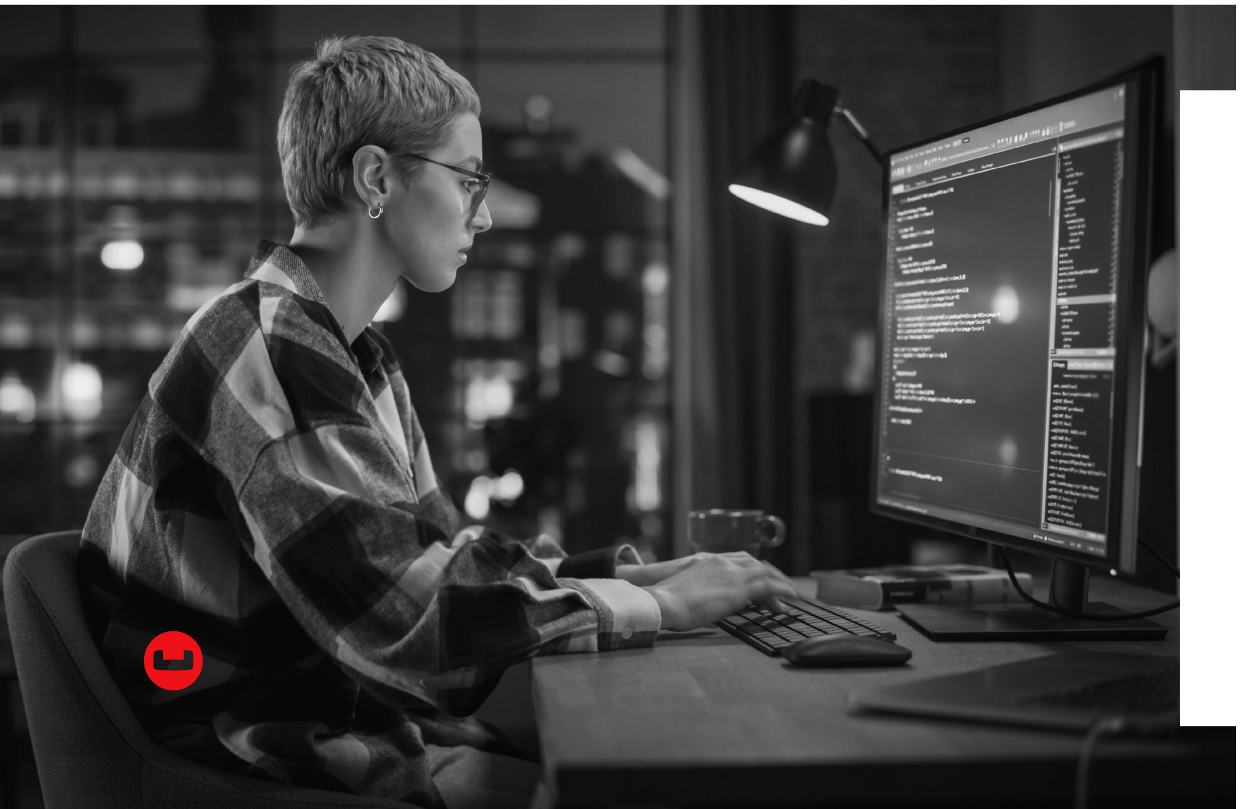
GETTING STARTED

You'll need a Couchbase instance, a bucket in that instance, and the web development tools and language of your choice. Couchbase Capella DBaaS is the easiest way to get started with Couchbase. If you haven't written a "hello world" with Couchbase yet, take that as your first step. Check out the SDK documentation using the language of your choice. You will likely be using a web framework, like ASP.NET, Spring Boot, Express, etc. If you need help with connecting your framework to Couchbase, you can check out the Couchbase Developer portal, the Couchbase Forums, or the Couchbase Discord.

WHO IS THIS FOR?

This guide is for developers or architects who are already familiar with the basics of connecting to Couchbase, and want to focus on choosing the right access method, and how to model data in JSON.

Also, while this guide focuses on the Conduit project, the principles within can be adapted to other domains. So even if you aren't building your own Conduit implementation, this guide can help you better understand how Couchbase fits in your application.



ENDPOINTS

The Conduit project backend spec is divided into 19 endpoints that manipulate Article and User entities, with related entities like Comments, Favorites, Following, Tags. You can use the tool of your choice to test the endpoints (including generated OpenAPI UI when available) but the Real World project has a pre-made Postman Collection for your convenience.

The business logic details may not be completely covered in this guide. The focus of this guide is all the database interactions and operations that make best use of Couchbase and Couchbase SDKs.

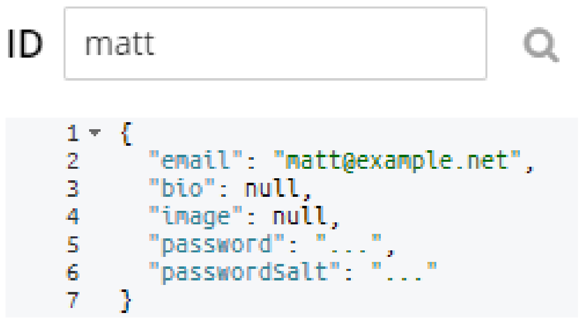
REGISTRATION

Quick Summary: Create a Users collection. Use username as the document key. Use an insert k/v operation to create new user documents.

DETAILS

This endpoint is where a User first gets created. Create a users collection in your Couchbase bucket (assume a "_default" scope for this entire guide).

One important choice to make is what to use the document key (aka document ID). Since usernames are often used in HTTP requests in other endpoints, having the document ID be the username is a good way to start.


ID matt

```
{
  "email": "matt@example.net",
  "bio": null,
  "image": null,
  "password": "...",
  "passwordSalt": "..."
}
```

The Conduit spec seems to make no allowance for username changing, which works perfectly with this design, as document keys are immutable. However, if you're building a system where changing usernames is a requirement, you might consider a surrogate key here instead (a GUID, UUID, random string, etc).

Also notice that the username is not in the body of the JSON.

To save a new user, use the key/value access pattern. Specifically, use an **insert** here. If someone else tries to register with the same username, a Couchbase insert will prevent a duplicate user being created.

One important security issue: notice that "password" and "passwordSalt" are stored in this user document. If you're building a proof-of-concept or prototype, this is okay. However, a 3rd party credentials providers like Okta, Auth0, FusionAuth, etc, would reduce the security risk of storing passwords in your own databases.

AUTHENTICATION

Quick Summary: Use a SQL++ query. Create an efficient index. Make sure to handle the document key.

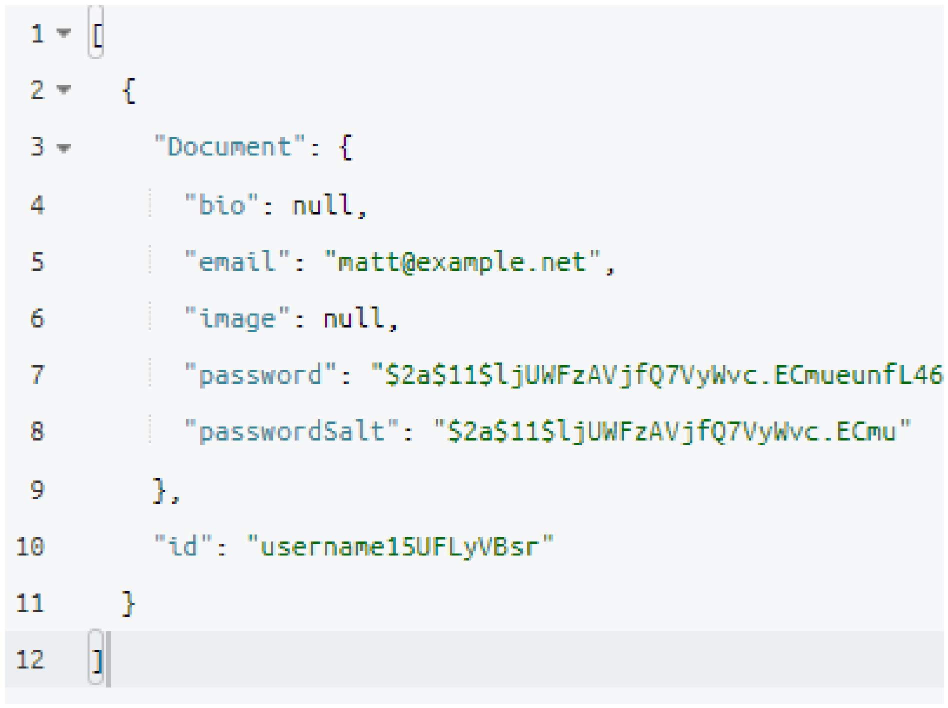
This endpoint will check a user's given login credentials and return a JWT token.

An oddity of the Conduit spec is that email (not username) is used for login. Since the document key is username, this endpoint will have to use a secondary lookup. For Couchbase, this means a SQL++ query.

```
SELECT u AS Document, META(u).id
FROM `Conduit`.`_default`.`Users` u
WHERE email = "matt@example.net"
```

Using SQL++, requires the use of an index. The quickest way to get this query running is to use a primary index: `CREATE PRIMARY INDEX ON `Conduit`.`_default`.`Users``. However, this index results in every user being scanned every time. A more efficient index would be: `CREATE INDEX `ix_users_email` ON `Conduit`.`_default`.`Users`(`email`)`. A covering index may be an option too for improved performance.

The response required by Conduit includes the username. But remember that the username is not in the document. It's the document key. That's why META(u).id is in the SELECT query. The results of this query look like:



```
[
  {
    "Document": {
      "bio": null,
      "email": "matt@example.net",
      "image": null,
      "password": "$2a$11$ljUWFzAVjfQ7VyWvc.ECmueunfL46
      "passwordSalt": "$2a$11$ljUWFzAVjfQ7VyWvc.ECmu"
    },
    "id": "username15UFLyVBsr"
  }
]
```

In your application code, you'll need to handle this structure. Perhaps you can create a generic customer wrapper object to contain ID and the document data in one structure for later use. Also note that password/passwordSalt are being returned by this query when they really don't need to be. If you are using a 3rd party auth provider, this data won't be in there.

Alternative Approach: If you want to avoid the overhead of a SQL++ query, you can create an "index" document, with the email address as the document key, and a body containing the username. This means you'll need two lookups: one to get the username and one to get the full user. This is an advanced technique that may involve duplication of data, and requires careful coordination of two documents for every user, but might be worth investigating if you haven't met your performance goals with the SQL++ technique.

GET CURRENT USER

Quick Summary: Get everything from JWT, or get the username claim from the JWT token, and use a Couchbase k/v **get** operation.

This method returns the currently logged-in user, given a JWT token. It's possible that you could entirely avoid interacting with the database if you put the entire user profile as claims into the JWT token.

However, a user profile might contain a lot of information, or information that's not appropriate to store in a JWT token. If that's the case, then a Couchbase k/v **get** operation works well here, assuming you've at least stored the username in the JWT token.

GET PROFILE

Quick Summary: Data access is nearly identical to Get Current User.

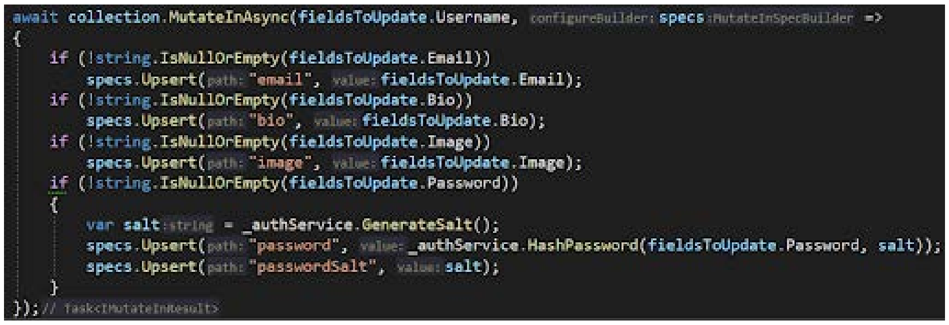
As far as data access goes, get profile is nearly identical to Get Current User. However, a "profile" is a public view of a user, and should only contain public information. A "get" operation returns an entire user, so make sure you have a mechanism in place to filter out sensitive fields like email address, password, information, etc.

UPDATE USER

Quick Summary: Use the subdocument API to make changes, and then use a k/v **get** to retrieve the whole profile.

This endpoint involves updating an existing user. One interpretation of the Conduit endpoint is that only fields that need to change are specified. For instance, if the HTTP request contains `{ "user" : { "bio" : "new bio" } }`, that means that only the bio field need be updated, and the other fields are left alone.

If that's the case, the Couchbase subdocument API can be used here. The subdocument API is part of the k/v API, and it allows you to interact with subsets of the larger JSON profile. For large user profiles, this is more efficient that bringing the whole document over the wire. The **subdocument API** will vary based on the SDK that you're using, but here's an example using C#:



```
await collection.MutateInAsync(fieldsToUpdate.Username, configureBuilder: specs :MutateInSpecBuilder =>
{
    if (!string.IsNullOrEmpty(fieldsToUpdate.Email))
        specs.Upsert(path: "email", value: fieldsToUpdate.Email);
    if (!string.IsNullOrEmpty(fieldsToUpdate.Bio))
        specs.Upsert(path: "bio", value: fieldsToUpdate.Bio);
    if (!string.IsNullOrEmpty(fieldsToUpdate.Image))
        specs.Upsert(path: "image", value: fieldsToUpdate.Image);
    if (!string.IsNullOrEmpty(fieldsToUpdate.Password))
    {
        var salt :string = _authService.GenerateSalt();
        specs.Upsert(path: "password", value: _authService.HashPassword(fieldsToUpdate.Password, salt));
        specs.Upsert(path: "passwordSalt", value: salt);
    }
});// Task<IMutateInResult>
```

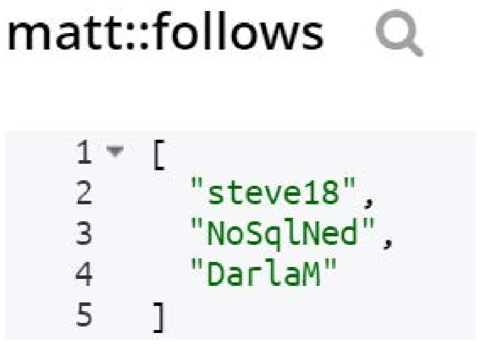
Also note that this endpoint returns the complete user. In that case, you may want to issue another full k/v **get** request, after the update has been completed.

Alternative approach: Use a full k/v **get**, make changes, use a full k/v **replace**, return the full document. For larger documents, subdocument API will likely improve efficiency.

FOLLOW USER / UNFOLLOW USER

Quick Summary: Use the data structures API to manage a document containing usernames. Any user in Conduit can "follow" other users. Content created by followed users shows up in a special "feed" when logging in.

To model this in JSON, one approach is for each user to have a "follow" document, containing a list of usernames that the user is following. The key of this document can be constructed for easy lookup (e.g. "<username>::follows") or it can be placed in its own collection and give the same key as the user. Here's an example of user "matt" following three other users:



matt::follows

```
[
    "steve18",
    "NoSqlNed",
    "DarlaM"
]
```

Note that this particular document is a JSON array. Couchbase SDKs have **data structure** capabilities that can manage arrays in a document. For instance, a "set" can be used to manage a unique list of usernames (ideal for this use case). These data structures are an abstraction over key/value and subdocument operations.

Another option is to have a "following" array embedded directly into user documents. However, if most of your operations rarely need both user AND following information, it might be more efficient to keep them separated.

Another concern is document size. In Couchbase, document size is limited to 20mb each. You can fit a huge array of average-sized usernames in 20mb, so that will not be a concern for average use. However, if it does become a concern, somehow, you can break the array up into two or more documents, with each document containing an ID of the next portion of the array.

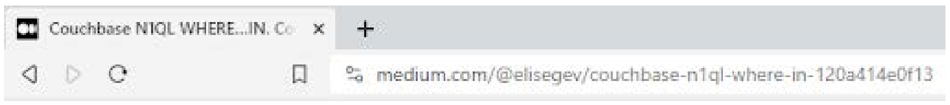
For validation, you may also want to verify the given user actually exists. This can be accomplished with an **exists** key/value operation.

The same access patterns apply to unfollow as well.

CREATE ARTICLE

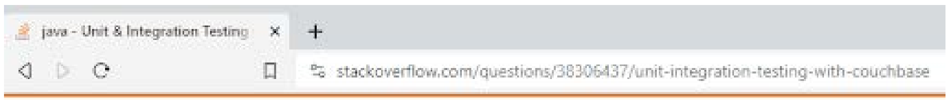
Quick Summary: Use an **insert** key/value operation. Use a unique string value as the identifier for an article.

This endpoint is called when an authenticated user is creating a new article (blog post, tweet, etc). Each article has a "slug", that's used in the url. For instance: https://myconduitsite/this-is-a-slug. A slug helps with usability and SEO. Since they're in a URL, slugs must be unique, so it might be tempting to use slug as the document key for articles. However, the Conduit spec shows that slugs can be changed. So, one option is to use an approach that sites like the real medium.com and stackoverflow.com use: create a unique immutable value for each article.



Couchbase N1QL WHERE...IN. Co

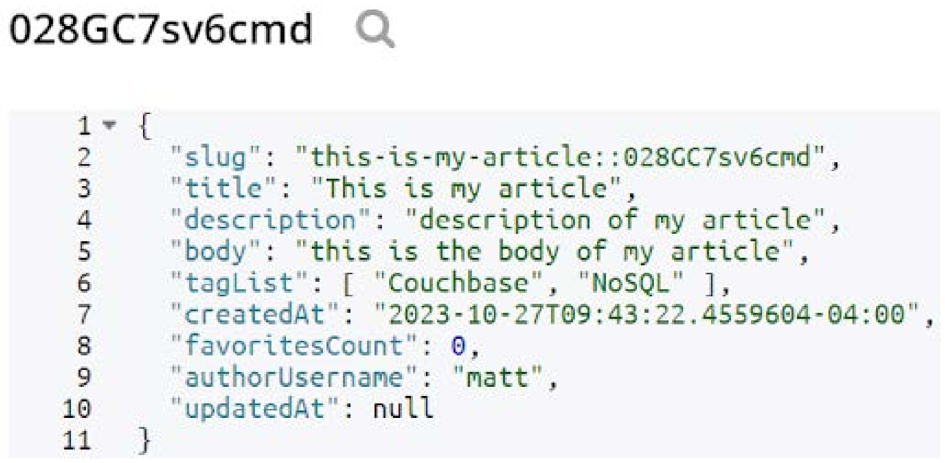
medium.com/@elisegev/couchbase-n1ql-where-in-120a414e0f13



java - Unit & Integration Testing

stackoverflow.com/questions/38306437/unit-integration-testing-with-couchbase

In both these cases, the URL is still SEO-friendly and human readable, but they also contain a random looking string/number that are used for the actual lookup.



028GC7sv6cmd

```
{
  "slug": "this-is-my-article::028GC7sv6cmd",
  "title": "This is my article",
  "description": "description of my article",
  "body": "this is the body of my article",
  "tagList": [ "Couchbase", "NoSQL" ],
  "createdAt": "2023-10-27T09:43:22.4559604-04:00",
  "favoritesCount": 0,
  "authorUsername": "matt",
  "updatedAt": null
}
```

There is some slight duplication in this example, storing the article key in both the document ID and as part of the slug string. This can be useful when querying. Also, I've used the delimiter "::" in the slug; you can use whatever you'd like.

This approach also solves the issue of if two people were to write an article with the exact same title. For instance, user matt could write "My opinion on Couchbase" and user laurent could write "My opinion on Couchbase," but each of them will have a unique key and slug. There are many approaches to handling slugs, but this way fits nicely with how Couchbase works.

Also note that since JSON does not have a date type, "createdAt" is being stored as a string. This is human readable, but storing a date/time as a UNIX timestamp is more efficient.

Make note of the "FavoritesCount", that will be important later on in the favorites endpoints.

AuthorUsername contains the user key that points to a separate user document. This means to get the full author information requires a second key/value lookup or a JOIN. For this endpoint, a simple **get** is probably the way to go, since this endpoint will only ever lookup a single user, and the key to that user (username) is in the JWT token.

Notice that tagList is an array containing literal tags. Having the tags embedded here reduces the amount of other joins or lookups that other, more relational designs, would incur. Renaming of tags would become costly, but as long as that doesn't happen very often, it's not a big deal. (Renaming tags in Stack Overflow, for example, is not a function that's available to normal users, and goes through a manual review process). More on tags later in the guide.

GET ARTICLE

Quick Summary: Use a **get** operation to retrieve the article. Use additional **get** operations when necessary to get author profile, following, and favorite information.

Similarly to getting a single user, a **get** operation is adequate to retrieve an article document from the Articles collection. The endpoint supplies the full slug, with unique ID appended at the end, so you'll need to parse that slug, basically ignoring everything except the unique ID to make your database get request.

At this point, if you look at the Conduit spec, you'll notice that getting an article should also include: if this article has been favorited by the current user, the author's profile information, and within that profile, if the current user is following that author. If you haven't built this functionality yet, it's okay to set placeholders. Or, you can skip ahead to favorite endpoints to learn more about how they function.

You can stick to using pure key/value to get all these values, especially since favorite/follow only need apply if the current user is logged in.

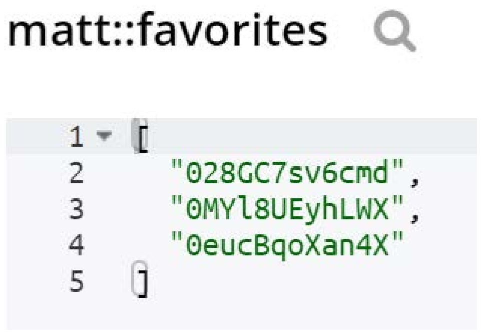
Alternatively: You can fetch all this information via SQL++, joining to favorite, follow, and author documents. This would potentially be a less performant way to retrieve an article, but it might be more maintainable and easier to read SQL++ code that multiple key/value lookups.

FAVORITE/UNFAVORITE ARTICLE

Quick Summary: Store favorites in a dedicated document for each user. Use an ACID transaction to update this document and the FavoritesCount in the corresponding article.

Articles can be "favorited" by logged-in users. "Favorited" can be used as a flag later to return a list of all the articles that a logged-in user has favorited. It's both a type of bookmark, and a way to measure how well liked a given article is.

The data model for favoriting will be very much like the data model for following/unfollowing. Create a document that has an array of article IDs that a given user has favorited.


matt::favorites

```
[
    "028GC7sv6cmd",
    "0MYl8UEyhLWX",
    "0eucBqoXan4X"
]
```

Note that with this design, even if an article changes title/slug, the article ID will remain the same, keeping the data consistent.

Just like following, use a **data structure** of a unique set to read/write these documents.

Unlike follows, there is one more piece of data that favorite/unfavorite must handle. In each article, there is a summary “FavoritesCount”. This number is how many times the article has been favorited. One approach to this would be a SQL++ query to sum up the count every time an article is accessed. However, that’s a lot of additional overhead for a number that doesn’t change very often.

Instead, whenever the “favorite/unfavorite” endpoint is used, increment/decrement the number in the corresponding article document. This is more denormalization of data, but the performance benefit is worth it.

For maximum consistency, use an **ACID transaction** to make sure that both documents are updated (or neither is updated, in case of an error).

LIST/FEED ARTICLES

Now that favorites, follows, and users have adequate modeling, it’s time to combine all them. This endpoint is meant for returning a paged list of all the most recent articles. A SQL++ query is a good fit for this endpoint:

- Multiple articles are to be returned
- The article IDs aren’t known in advance
- Articles need to be combined with favorite, following, and user information

Here is a SQL++ query that closely matches the output needed for this endpoint:

```
SELECT
   a.slug,
   a.title,
   a.description,
   a.body,
   a.tagList,
   a.createdAt,
   a.updatedAt,
   false AS favorited,
   a.FavoritesCount,
   {
        "username": META(u).id,
        u.bio,
        u.image,
        "following": ARRAY_CONTAINS(COALESCE(fol,[]), META(u).id)


   } AS author,
   COUNT(*) OVER() AS articlesCount


FROM Conduit._default.Articles a
JOIN Conduit._default.Users u ON a.authorUsername = META(u).id
```

```
/* these next lines are only for authenticated users */
/* usernames need parameterized */
LEFT JOIN Conduit._default.Favorites favCurrent
     ON META(favCurrent).id = ("mgroves" || ":favorites")
LEFT JOIN Conduit._default.`Follows` fol
     ON META(fol).id = ("mgroves" || "::follows")


/* for use with optional filter */
/* username need parameterized */
LEFT JOIN Conduit._default.Favorites favFilter ON META(favFilter).id
= ("jake" || "::favorites")


/* convenience variable for getting the ArticleKey from slug */
LET articleKey = SPLIT(a.slug, "::")[1]


WHERE 1=1
  /* optional filters */
  /*AND ARRAY_CONTAINS(a.tagList, "cruising")
  AND a.authorUsername = 'user-u4tjaxvr.2lw'
  AND ARRAY_CONTAINS(favFilter, articleKey)
  AND ARRAY_CONTAINS(fol, a.authorUsername)*/  /* used for Feed endpoint */


ORDER BY COALESCE(a.updatedAt, a.createdAt) DESC


/* needs parameterized */
LIMIT 20
OFFSET 0
```

Important notes:

- Not every request needs to perform certain JOINs and include certain predicates. Construct the dynamic parts of the query based on endpoint input. For instance, if the current user is not logged in, there's no need to include the first two LEFT JOIN statements at all.
- Also notice that the Feed and List queries will be very similar, enough so that you should consider sharing this query, however you are structuring your data access code.
- "COUNT(*) OVER() AS articlesCount": this is using a "windowing function" to get a total count of every article for paging purposes.
- Finally, much of the query should be parameterized: "mgroves" should be replaced with the logged-in user's name; LIMIT 20 OFFSET 0 should be replaced with the appropriate page size and page number; "following" can be set to a constant value of "false" if the user isn't logged in, and so on.

Alternative: Using this endpoint as designed in the Conduit spec is probably not a good idea for a truly large scale user-facing website. The query does a lot of work, and with a large concurrent user base, will put strain on your application. There are few websites that provide a true "firehose" of the latest articles without some constraint. But if you must do this, consider **caching** the results, so that future requests can pull the data without engaging the query engine. Couchbase is well-suited for caching, and you can set a TTL (time-to-live) on the results document, so that results can only be as stale as the TTL set.

Alternative: Another aspect of this query is that it's projecting a lot of fields: slug, title, body, tags, author, etc. An approach that might put less pressure on the query/index services would be to only return the IDs of the articles, and getting all of the rest of the information via key/value lookup. Depending on your performance goals and use case, this could help performance a lot. However, it's more work to build/maintain that kind of hybrid code base.

Alternative: To further reduce latency but reduce cache footprint, you could combine the above two alternatives: cache a list of IDs.

UPDATE ARTICLE

Quick Summary: Use k/v operations like get, exists, and subdocument.

Just like with Update User, this endpoint can benefit from **subdocument mutations**. That is, only update the parts of the user that are necessary.

Some additional concerns for this endpoint that involve data access:

- Checking to see that the current user is the author of the article. If not, they are forbidden from making changes to it (this can be a **get** operation).
- Making sure the given article exists (this can be an **exists** operation).
- If the title of the article changes, that means the slug must be changed. The new slug must still have the same article ID appended to the end.

Finally, when updating an article, the "updatedAt" field must be added/updated with the current date/time. Remember the JSON date/time considerations brought up earlier.

DELETE ARTICLE

Quick Summary: Use the delete method or use a soft delete, but make sure the other endpoints handle this situation.

Deleting data in Couchbase is as simple as using the **delete** (or remove) method. However, there are some important considerations:

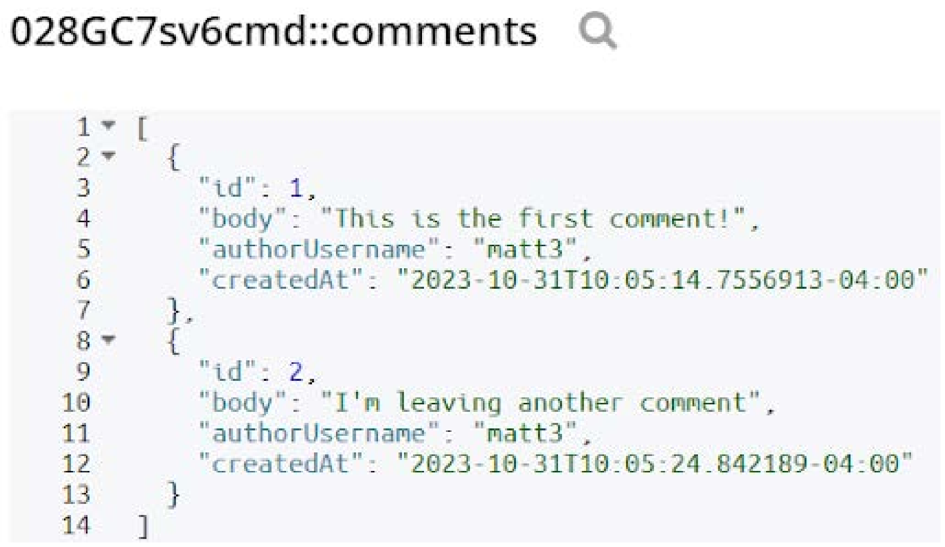
- Many users may have “favorited” this article already. What should be done about that data? Is it okay to leave it there? Should it be removed? Can it be removed later?
- Should articles actually be deleted? You could also perform a “soft” delete (by adding a “deleted” flag or a “deletedAt” date/time to an article (use **subdocument** to do this). If you do this, all other endpoints that may access this article must make sure to ignore articles that have this flag set.
- Potentially, you could copy these articles to a different collection, like ArticlesDeleted before deleting them (probably in an **ACID transaction**).
- You may also consider using **Couchbase Eventing** to handle or clean up deleted articles.

The decision is yours, and depends on your use case and data retention policies.

ADD COMMENTS TO AN ARTICLE

Quick Summary: Use a similar pattern to follow/favorite. Use autoincrement to give each comment a unique number.

Users can submit comments to articles. One approach to take is similar to favorites/follows: create a “comments” document that corresponds to a single article document. This document contains an array of comment data. It can have the same/similar key as the article it is commenting on, assuming that it’s in its own “Comments” collection, making it easy to lookup with just one more k/v operation:



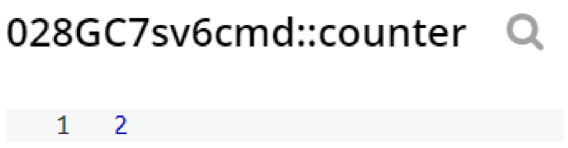
028GC7sv6cmd::comments

```
[
  {
    "id": 1,
    "body": "This is the first comment!",
    "authorUsername": "matt3",
    "createdAt": "2023-10-31T10:05:14.7556913-04:00"
  },
  {
    "id": 2,
    "body": "I'm leaving another comment",
    "authorUsername": "matt3",
    "createdAt": "2023-10-31T10:05:24.842189-04:00"
  }
]
```

Note that each comment has an ID number. This number exists solely for the purpose of being able to later delete the comment. For deletion, each comment must have a unique identifier: finding an individual comment by other means may not be reliable enough (perhaps you can think of a clever hashing method to identify comments). If your application doesn't support deleting of comments, then this ID number is not needed.

Couchbase does not have "autoincrement" fields in the same way that relational databases do. However, Couchbase does have **increment** (and decrement) commands. These commands, performed on a single document, will atomically update and return an ID number. This number can be used as a comment ID. An ACID transaction is not necessary: if the operation fails, just try again. An ID number may be skipped, but that's not a concern in this instance.

For each comment document, create a counter document. Here's the counter document for the above comment:



028GC7sv6cmd::counter

```
2
```

The same issues of JSON date/time applies.

Another issue that was brought up earlier with follow/favorite is document size limit. Couchbase documents are limited to 20mb in size. Depending on the average comment size, 20mb can store MANY comments. However, just like with favorite/follow, if this limit is a concern, you can architect a pattern where each comment document is numbered, and/or each comment document contains the document key for the next comment document.

Alternatively, you can limit the number of comments on an article. Given the overall state of internet discussion, this might be the wiser alternative.

GET COMMENTS FROM AN ARTICLE

Quick Summary: Use a SQL++ query with USE KEYS and UNNEST for efficient retrieval of comments and associated author information.

Comments are in a single document, however this endpoint requires joining comment information with each author's profile. Again you have the option of using SQL++ to perform the joins, or using k/v operations to lookup each author individually.

Given the relatively unbound nature of comments, and that each comment will probably have a different author, a SQL++ query may be the most efficient approach. And thankfully, SQL++ has a "USE KEYS" syntax, that reduces the need for indexing and offers an efficient lookup-based approach. Consider a SQL++ query like this:

```
SELECT VALUE
    {
        c.body,
        c.createdAt,
        c.id,
        "author" : {
           author.bio,
           author.image,
           "username": c.authorUsername,
           "following": ARRAY_CONTAINS(COALESCE(follow,[]), c.authorUsername)
        }
    }


FROM Conduit._default.Comments c2
USE KEYS "OjeIfc7nudAW::comments"
UNNEST c2 AS c
JOIN Conduit._default.Users author ON c.authorUsername =
META(author).id


/* join to the logged in user if not anonymous */
/* parameterized with logged in username */
LEFT JOIN Conduit._default.Follows follow
  ON ("mgroves" || "::follows") = META(follow).id
```

Parts of this need to be parameterized and dynamic, just like with List/Feed:

- USE KEYS "<article key>::comments"
- "mgroves" should be replaced with the logged-in user's username
- The LEFT JOIN can be omitted if the current user is not logged in.

Notice the use of UNNEST here: this is powerful SQL++ syntax that can treat each comment in the array as a single entity. And SELECT VALUE ensures that the resulting data won't be wrapped in an unnecessary JSON array.

The results of this query closely match the results expected by this endpoint.

Alternative Approach: To avoid any kind of JOIN to author information, each commenter's profile could be copied and embedded into the comment document. Be careful with this approach: it can improve performance, but at the cost of consistency and a large comment footprint.

DELETE COMMENT

Quick Summary: Use **data structures** delete for a true deletion. If soft deleting, make sure the other endpoints understand.

Just like with the delete article endpoint, the same principles of deletion apply here.

The main difference in a true deletion is that the comment ID number is required to identify the comment to delete.

Also, if you are using a **data structures** collection, make sure you understand how it works. For instance, if you're using a .NET object, Equals must be overridden, since objects use a reference comparison by default instead of a values comparison. You could also use a .NET record instead of class (which uses a values comparison by default).

GET TAGS

Quick Summary: Store acceptable tags in a single document.

Finally, the get tags endpoint returns all valid tags that can be used when creating an article, or when filtering a list/feed by tag.

Tags can become messy if you don't put restrictions on them. If you allow any tag, consider that this may include typos, synonyms, or rarely used tags. Consider articles tagged with one or more of "couchbase," "couch base," "couchbsae," etc. Instead, you can go with a design similar to StackOverflow: have a single list of acceptable tags and only allow those.

To manage that list, you could certainly hard code it. That wouldn't involve any database activity at all.

Another option is to create a single document, perhaps in a Tags collection, with an array of acceptable tags. This list could be retrieved with a single **get** operation. To update this list, you need only make changes to that document. While Conduit doesn't have any endpoints to manage a tag list, certainly this could be added to an "administrative" area in your application.

OTHER REQUIREMENTS

This concludes all the database access requirements necessary to support a Conduit implementation. While the Conduit spec is detailed and useful, there is a lot of wiggle room in how you actually implement your application, whether you are building a Conduit implementation, or using this document to help make decisions about your own application.

This entire document has featured these access patterns in Couchbase:

- Key/value (get, replace, insert, delete)
- Data Structures (list, set)
- ACID Transactions (begin,commit,rollback)
- SQL++ (SELECT …)

For Conduit, this is all that you really need. If you want to expand on Conduit, or are working on your own application, you might want to explore other Couchbase data access options.

FULL TEXT SEARCH

This would be the next logical step for Conduit: being able to search articles/comments/users based on a text search. With many databases, this requires creating a data pipeline to copy data to another tool like Solr or Elastic. With Couchbase, this service is built in. You only need to create an FTS index. Further, FTS can be used for geospatial, geographic, and faceted search.

MOBILE / MOBILE SYNC

If you want to make a mobile app version of Conduit, one approach you could take is to automatically sync data from Couchbase Capella to Couchbase Lite embedded on mobile devices, via Capella App Services. Using this approach means that your users can interact with your website even when they aren't connected to the internet. Further, any reads could be performed directly on the device, improving response time for mobile users. Again, no need to bring in another tool to enable this: Capella App Services are built right into Capella, and Sync Gateway can be used for on-prem deployments.

TIME SERIES

Couchbase supports time series formatted JSON data, and can query time series data within SQL++. As-is, Conduit data isn't particularly time series focused. However, if you plan to track traffic, engagement, performance metrics, A/B testing, etc, time series might be just what you need. Again, no need to bring in another database for basic time series querying and analysis: it's already built into Couchbase.

ANALYTICS

Again, Conduit data as presented here may not be suited for many analytics use cases. But if you want to answer bigger questions about your data, and do in a timely manner, the Analytics service can help. The Analytics service in Couchbase is perfect for operational analytics, sometimes called HOAP, HTAP, and translytical. You can write complex SQL++ queries that can feed into dashboards and visualizations, without hurting performance of the moment-to-moment data operations through the concept of "workload isolation."

EVENTING

As mentioned earlier as a possible candidate for data cleanup in the "delete" endpoint, eventing allows you to deploy custom code to Couchbase that responds to data mutations.

For instance, when an article is deleted, eventing can be used to automatically archive the data and clean up any lingering "favorite" data for that article. Eventing can respond to any add, update, or delete event, as well as timers. And yet again, you don't need to bring in another queueing or streaming service: eventing is built right into Couchbase.

TESTING

Functional Tests

You can use the Postman Collection, supplied by Real World, to verify your endpoints. This can be added to an automated test suite, for use in CI/CD pipelines using a command line tool like Newman.

One one unit test is required by the Real World project. However, creating a full suite of tests can help you to think about and prove out requirements.

Load Test

When building your site, sometimes it's not obvious which access pattern will provide the best performance. Key/value? SQL++? Combination of SQL++ and key/value?

If performance is the most important aspect, it can be useful to run benchmarks, comparing endpoints implemented with different data access approaches.

Some options and examples::

- Gatling. Laurent implemented load tests using Gatling to compare key/value and SQL++ endpoints: https://gitlab.com/devoxxfr-2023/env-tests/realworld-ottoman/-/tree/main/gatling?ref_type=heads
- BenchmarkDotNet: The ASP.NET + Couchbase example implemented load tests using BenchmarkDotNet to compare key/value and SQL++ endpoints: https://github.com/mgroves/realworld-aspnet-couchbase/tree/master/Conduit/Conduit.Benchmarks

SUMMARY

This guide is designed to help you further understand Couchbase data access patterns and their effect on performance in your application. While your application may not be a medium.com clone like Conduit, there's still plenty to learn from the access patterns and data models used to build Conduit.

Further resources:

- Got questions? Check out the Couchbase Forums or the Couchbase Discord
- Couchbase: Optimizing Performance
- Advanced SQL++ Features
- Couchbase Capella DBaaS

Couchbase