

Zachary Bowyer

5/11/2023

System Investigation Project Part 2

Investigation of Apache CouchDB

System summary

Apache CouchDB is a NoSQL database management system that uses self-contained JSON documents to store data and procedures, HTTP protocol to transfer data, and Javascript to program queries. To summarize the system description in the next section, I will briefly describe each key feature. First for the data model, CouchDB has databases, documents, design documents, and views, which are written in JSON and Javascript. Databases are akin to traditional tables, documents are akin to traditional rows, and design documents/views are akin to traditional queries. Each document has its own unique identifier and version control number. In terms of interfacing with the system, CouchDB relies on HTTP protocol, which is very simple to understand, and is quite commonly wrapped around to help with either GUIs or automation. For example, I was able to interface with CouchDB using a web portal, the curl command on the command line, and with a python package, which are shown later in this document. For indexing, CouchDB allows two types of indexing, which are primary and secondary indexing. Primary indexing is done through the automatic creation of document ids and secondary indexing is done through an explicit function in views. Consistency in CouchDB is designed to be eventually consistent, which means that changes in one node will eventually reflect the rest of the system. Of course this comes with both reliability and conflict tradeoffs. Related to this, transactions are not supported in CouchDB because documents are independent, have multiple versions, and are not always consistent with each other. Next, it is evident that CouchDB provides good replication options, as a user has the option to make either a single node system or a cluster system. These clusters act as backups for each other. CouchDB is scalable, as it provides the ability for customizable sharding among the clusters in a node. Finally, CouchDB offers basic, customizable, and robust security measures. Overall, CouchDB provides a simple and robust framework for data that does not need high consistency. One other major feature of CouchDB is that through its versioning system you will not lose data.

System Description

As stated previously, Apache CouchDB is a document-based NoSQL system. This means that it stores its data in self-contained JSON documents. First, CouchDB stores data in a database object, which can be equated to a relational table, however, you are not limited to any structure of data held in the database but it is considered best practice to use a different database object for each type of data you are storing for performance reasons. The structure of each document, while variable, has two key features that must be in all documents. These are “_id”, which is used as a key to the document, and “_rev”, which is the current revision version of the document. Naturally, the information stored in a document can be as complex as the JSON format allows. One might ask if data should be stored as multiple documents, or as multiple items in one document. From my research, the more complex your documents are, the harder they are to query, as there is no guarantee that all documents will be the same. Additionally, you are limiting yourself to using Javascript to do your queries. For example, if you

store 1000 'Person' objects in one document, you may have to loop through them all with Javascript to run a query, which is slower than checking one-thousand documents that all contain one 'Person' object. Next, to query data, a design document must be used. In essence, a design document is a JSON file that holds multiple views. Views are the relational equivalent to queries of CouchDB. They are simply Javascript files that use the 'map' function of Javascript that map documents to key-value pairs of the data. One important feature of views is that their results are stored permanently, rather than needing to be rerun each time like a query. Additionally, views are automatically updated every time a new document is created, modified, or removed. **Image 1** is an example of a document and **image 2** is an example of a design document containing multiple views. Notice how each document has an id and revision number. These are important components that affect querying, consistency, and replication. As shown, stored data in CouchDB is very simplistic. As will be discussed later, this data can be easily interfaced with using HTTP protocol, which is abstractable enough to allow any tool you want to interface with your database.

While CouchDB provides a simple interface for modifying data, creating databases, and querying data through 'Project Fauxton', which is a simple web portal, it should be understood that it is simply a wrapper over how the interface actually works, which is HTTP protocol. With the HTTP protocol, users can make GET, PUT, DELETE, etc., requests to modify, store, and delete databases, documents, design documents, and views. Of course when making these requests to the HTTP server, a username, password, ip, and port must be provided in the request. As mentioned earlier, the web portal is an easy to use GUI that allows users to perform those tasks at a click of a button. Of course, this is not needed. For example, users can use the Curl command to make HTTP requests, or use something like the Request library in Python, which will be shown later. Overall, it makes the most sense to do automation with a programming or scripting language and one-time, higher level changes with the portal. With that being said, I will show some examples of how the HTTP requests work with a CouchDB instance. Let's say you have set a username and password to "admin" and "password" and your ip and port are '127.0.0.1' and '5984'. To create a database called 'testdb', you simply need to run the HTTP request: PUT http://admin:password@127.0.0.1:5984/testdb. To create a document in this new table, you can run PUT http://admin:*****@127.0.0.1:5984/testdb/001-d>{"name":"Zach", "age":22}; the 001 is the id of the document.

Database indexing is a simple and powerful way to speed up data retrieval. Essentially, an 'index' is a pointer to some data. For CouchDB, indexing is natural, as it stores documents as key value pairs. As stated previously, all documents contain an "_id" field, which is its primary key. From there you could easily write a view that asks for the data with that specific "_id", which would be indexing. Additionally, you can define indices for specific queries/views by attribute through the keyword "index" in JSON for CouchDB. Therefore, in terms of what indices are supported, you can index both on entire documents by key and by attribute. One example of the primary index being created is shown in **image 1**, where you can see that "_id" was automatically generated. Finally, **image 3** is how a secondary index is created through a view. This defined secondary index through the creation of a data store that holds specific information

about age ranges and primary keys should increase the lookup speed when it comes to the 'age' attribute.

In terms of system consistency, this is a non-issue for the single node database option, but is for clusters. CouchDB opts to settle for the term coined as "eventual consistency", which is simply the notion that as updates are made in a node, they will eventually be reflected in all other nodes, which is of course a tradeoff for increased availability and partition tolerance according to the CAP theorem. This methodology prevents common issues with high consistency systems like locking, which is essentially forcing all nodes to halt execution until they are updated. Of course this can be very bad depending on the use case. With that in mind, the way CouchDB implements its eventual consistency is quite simple. After a document is updated, CouchDB uses what it calls "incremental replication" to update the rest of the system. This is a process where documents are periodically copied between nodes. The time between copies is not clear, my guess is that it is triggered each time a document is changed. For the topic of transactions, first it is important to understand that transactions are essentially a sequence of work, or multiple operations performed on a database that must either completely finish, or not finish at all. For example, if a system were to crash half-way through a transaction, the first few operations would be reversed by setting the database to the previous state. From our understanding of both transactions and CouchDB's architecture, it is clear that transactions are not suited for CouchDB, as all documents are independent, have multiple versions, and can be unsynced.

When it comes to scalability and replication, CouchDB offers the ability for users to make tradeoffs between latency and concurrency. For example, a user could make a system that is very fast or has duplicated storage on multiple machines. With that in mind, it is evident that CouchDB supports replication of data. The way this works is that first you naturally need multiple databases set up, which will be a source database and destination databases. Naturally, data will be copied over from the source database to the destination databases through HTTP requests. However, this process takes time, which alludes to the concurrency tradeoff previously mentioned. Slightly similar to how Git works, replications can be done in either direction, called 'pushes' and 'pulls' as well as in both directions simultaneously, called 'master - master' replication. This of course introduces the possibility of merge conflicts, which is handled by CouchDB by simply storing all possible versions of the conflicted documents in the databases, and picking one document as a winner based on a deterministic algorithm so that all databases will pick the same winner. For scalability, one common method used by database management systems is sharding. Sharding is described to be a 'horizontal partition' of data, which essentially is storing row subsets of the data to different machines. For example, if you have 100 rows of data, sharding would be storing the first 50 rows on one database, and the other 50 on a separate database. With that being said, when it comes to sharding, CouchDB offers a modifiable sharding protocol for node clusters. Without any setup CouchDB clusters are automatically sharded with default settings. In terms of the modifiable features, CouchDB allows users to: Choose the number of shards and replicas per database, move shards, copy shards, set shards/nodes to maintenance mode to pause interaction, update cluster metadata to reflect

target shards, force shard synchronization, monitor the internal replication of shards, remove shards, split shards, stop resharding jobs, and merge shards.

For security, Apache CouchDB has a very simple model which can be extended to make custom security systems if users wish. In short, each database in a CouchDB system has an associated administrator account which is able to grant permissions, create users, etc. This means that no random person with an internet connection can access or modify CouchDB instances. This security is why the example HTTP requests shown in this document all have usernames and passwords in them. One important feature that CouchDB offers is password hashing. This means that you include the password hash in code instead of the plain text password, for example. While there is the previously mentioned security method, referenced by CouchDB as “basic authentication”, there is also another, more professional authentication, which is cookie authentication. Once logged in, a user will generate a cookie that allows subsequent requests in an allotted time slot to not require password authentication, this prevents the need to constantly pass credentials to CouchDB for each request.

System installation

For my system installation process I will outline the steps I followed in order to accomplish the following: Download CouchDB, install CouchDB, create a database, create database documents, and create database views. I will then show how to run views that query data, add data, and delete data through three methods: Fauxton web portal, command line, and finally Python as I plan for my final implementation to be in Python. With that in mind, I first started by navigating to the [Apache CouchDB download page for Windows \(x64\)](#), where I then downloaded CouchDB version 3.3.2 as the file “apache-couchdb-3.3.2.msi”. From there, I had to tell my computer to bypass windows security to run the installation wizard. Next, I clicked ‘Next’, ‘Accept’, ‘Next’, ‘Next’. One important thing to note is that the wizard tried to set my default installation path to E:, which may not be what others want. For example, it is very common to install to the C drive, however my C drive is almost full, so I opted for my secondary drive. From there, it asked for a cookie value, to which I told it to generate a random one for me. It also asked me to set a username and password at some point. Finally, the program installed, and required a system reboot. One thing I noticed in my installation is that the CouchDB server automatically starts every time I launch my computer, which is both a good and a bad thing. It is a good thing since I do not need to redundantly reopen the server every day, but also a bad thing because it could be taking resources away from other processes I need on my personal computer. Looking back, I should have paid more attention to setup settings as I will likely have to fix this later. Once I had CouchDB installed and running, I navigated to the [Fauxton web portal](#), and logged in with the username and password I had previously set in the installation process. Next I [verified](#) that the installation worked. After that I clicked on the ‘wrench’ icon, which takes you to [this page](#), and there I clicked on the button to configure the database as a single node, using the default bind address, the default port, and the same username and password. I opted to ignore the message that appears on the page that says “Apache CouchDB is configured for production usage as a single node! Do you want to replicate data?”.

With this general setup finished, I finally created my own database called “testdb1” and set it to “Non-partitioned”, as that was the recommendation. One thing to note is that database names cannot contain uppercase characters. **Image 4** is proof of the database being created, and as shown the database was empty in that image. With the database being created, I added three documents to it, which included the data of three ‘Person’ objects, which all had an age, name, and type. The **image 5** shows these documents in the database, shown in ‘Table’ format. Next, I created a design document named “testdesigndoc1”, which holds three views that I created, which are subsequently labeled “Get all people”, “Get all people over 50”, and “Get all people under 24”. One thing to note is that you don’t explicitly run views, as they automatically re-run each time a change is made to any document, a document is added, or a document is deleted. If you want to view the output of a view, you can simply click on the view under the design document. Finally, deleting a document in the web portal can be done by clicking the check mark on the left side of it’s entry and clicking the garbage can button that appears above. Below in **images 6, 7, 8, 9, 10, and 11** are the three views created and their outputs. From here, I decided it would be worthwhile to additionally document the process of creating and modifying a CouchDB database through the command line. Instead of explaining the process through each step, I will simply show the commands I used with a short description.

Get database information.	curl http://admin:*****@127.0.0.1:5984 (* is password)
Create database table	Curl -X PUT http://admin:*****@127.0.0.1:5984/testdbcurl
Add document to table	curl -X PUT http://admin:*****@127.0.0.1:5984/testdbcurl/001 -d '{"name":"Zach","age":22}'
Create design document	curl -X PUT http://admin:*****@127.0.0.1:5984/testdbcurl/_design/testdesigndoc1 -H "Content-type: application/json" -d '{ "_id": "_design/testdesigndoc1", "language": "javascript" }'
Add view to existing design document	curl -X PUT http://admin:*****@127.0.0.1:5984/testdbcurl/_design/testdesigndoc1 -H "Content-type: application/json" -d '{ "_id": "_design/testdesigndoc1", "_rev": "1-ac85fc9abd1578d42425e6186d27ceb0", "language": "javascript", "views": { "GetPeople": { "map": "function(doc) {if(doc.type == \"Person\") {emit(doc._id, 1);}}" } } }'
Run view	curl -X GET http://admin:*****@127.0.0.1:5984/testdbcurl/_design/testdesigndoc1/_view/GetPeople
Remove document	curl -X DELETE http://admin:*****@127.0.0.1:5984/testdbcurl/001? rev=1-c7585ac854784ccaee8a988ff3a526ae

Finally, **image 12** shows a basic python script I wrote to query some data, which will be helpful for implementation in the future. Overall, doing things through the command line or Python is more complicated than doing it through the web portal, however it gives the advantage of automation. It makes much more sense to create tables, design documents, and views on the web portal, and then add/delete/modify specific documents automatically through curl/python.

System design (1-2 pages)

When it comes to how I will personally apply Apache CouchDB's architecture to my specific problem, which is answering data questions about the GTFS dataset, I plan on keeping my architecture simple. First, as stated in the previous assignment, I plan on using only one database/node called "GTFS", which will be my personal computer for simplicity reasons. From there, I plan on having each row of each table represented as a JSON document, where each attribute in the document corresponds to its respective table columns. Most importantly, each document will have its own "doc_type" attribute which will determine which table it came from which means the data can be better separated. In terms of document types, I will have 'Agency', 'Calendar', 'Calendar_date', 'Fare_attribute', 'Fare_rule', 'Route_direction', 'Route', 'Shape', 'Stop_feature', 'Stop_time', 'Stops', 'Transfers', and 'Trips' which will have a count of 56, 598, 7, 12, 184, 94, 989919, 39393, 3047657, 6558, 3730, and 64647 respectively. Next, I will have one design document that will encompass any query that I may need to make, and since I only need to implement three queries for this project as outlined in the project requirements for part 4, I will only need to implement a handful of views at most, so there is no need for many design documents. Of course my design documents will contain the views and subviews that I may need. An example of this data model is illustrated in **image 13**. The number at the top indicates how many of that document type will exist. In terms of indices for my datamodel, it is likely that I will need to create indexes for shapes, stop times, and stop features, as they have much more data than the other document types will have. Adding indexes to these should greatly increase the speed of the queries/views I will create.

While thinking of a data model design, I had a few other variations of my current model that I considered. However, due to various reasons I decided not to pursue them. My first idea was to have a different database for each document type, however I quickly found out that it is not easy to run queries on multiple databases at a time. For example if I wanted to do a join on two databases, it would be much more complicated than doing it on documents in the same database. Additionally, this is where I got the idea to use a simple universal tag identifier to keep the table like relationships in the data. One other idea I had was to combine particular small tables into one table. However, I decided against this as I am more familiar with and biased towards relational systems. So in order to preserve the defined relations as much as possible to the original description, I decided to make each file its own document type. Overall, I stuck with my current design because I couldn't easily use multiple databases as tables, and I wanted to see how close I could mimic a relational database with this specific document-based NoSQL system, which is mainly for my own practice over it being the best possible architecture for the problem.

When it comes to how I will actually get the data from the .txt files to my data model, I simply plan on writing a few python scripts that will loop through each file as .csv, as that is the format of those files. From there, I will create the JSON profiles from my current schema and create a new document for each row of each file. I expect creating the data will take some time as I will be looping over 4,152,855 items and making 4,152,855 HTTP requests. The request time alone should make it take a while to populate the database. If there were more data I would consider using an alternative method that is faster. In terms of data cleaning, fortunately the data is mostly clean. However, there are a handful of attributes in each document type that are mostly empty values. These are fare_attributes.transfers, fare_attributes.transfer_duration, fare_rules.rout_id, routes.route_color, routes.route_text_color, stops.parent_station, trips.trip_short_name, and trips.trip_type. My plan for these is to insert null values wherever empty data shows up.

Finally, I plan on executing three queries for my implementation of CouchDB on the GTFS dataset per the project requirements in another document. These queries in English are: "List all routes that go to the Portal City Center", "List all days when the max red line run in a given time range", and "List all modes of transport routes, stop types, route type, time of arrival, and direction available near PSU Urban Center".

Conclusion

Overall, I believe that after writing this document I have a good understanding of the basics of Apache CouchDB. From scalability, to security, to implementation, to the data model, and others, I have shown that I have sufficient knowledge of the system to begin an implementation project. Additionally, I have outlined my project plan in the previous document, and have no intention of changing it or its milestones.

Images:

Image 1

```
{
  "_id": "2f1734be4e2019c86455ec70b0002563",
  "_rev": "2-efc4c449c8d6c67733ac7055c38dae49",
  "type": "Person",
  "name": "Zach",
  "age": 22
}
```

Image 2

```
{
  "_id": "_design/testdesigndoc1",
  "_rev": "8-618c5d6c5acdf83d690108f325f1e9d8",
  "views": {
    "Get all people": {
      "map": "function (doc) {\n  if(doc.type == \"Person\") {\n    emit(doc._id, 1);\n  }\n}"
    },
    "Get all people under 24": {
      "map": "function (doc) {\n  if(doc.type == \"Person\" && doc.age < 24) {\n    emit(doc._id, 1);\n  }\n}"
    },
    "Get all people over 50": {
      "map": "function (doc) {\n  if(doc.type == \"Person\" && doc.age > 50) {\n    emit(doc._id, 1);\n  }\n}"
    }
  },
  "language": "javascript"
}
```


Image 3

```
1 function (doc) {  
2   index("age", doc.age);  
3   if(doc.type == "Person" && doc.age < 24) {  
4     emit(doc._id, 1);  
5   }  
6 }
```

Image 4

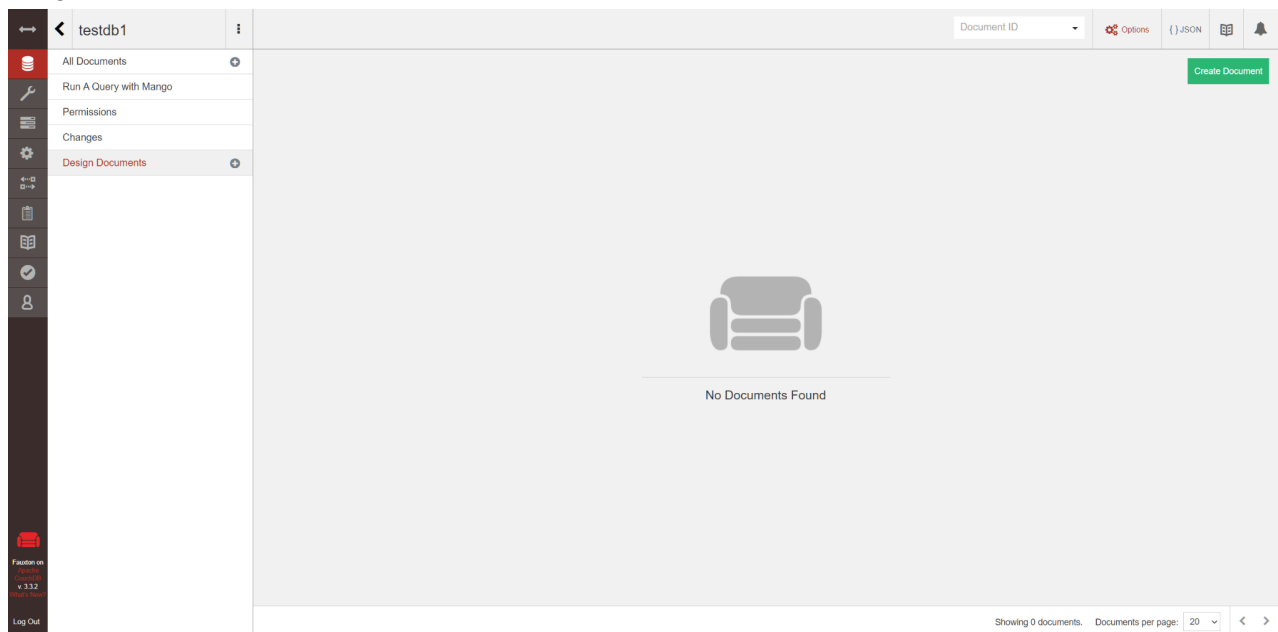


Image 5

_id	age	name	type	language
211734be4e2019c86455ec70b0002563	22	Zach	Person	
211734be4e2019c86455ec70b000363d	23	John	Person	
211734be4e2019c86455ec70b0004cdc	57	bob	Person	
._design/testdesigndoc1				javascript

Image 6

```
1 function (doc) {  
2   if(doc.type == "Person") {  
3     emit(doc._id, 1);  
4   }  
5 }
```

Image 7




 2f1734be4e2019c86455ec70b0002563	22	Zach	Person
 2f1734be4e2019c86455ec70b000363d	23	John	Person
 2f1734be4e2019c86455ec70b0004cdc	57	bob	Person

Image 8

```
1 function (doc) {  
2   if(doc.type == "Person" && doc.age > 50) {  
3     emit(doc._id, 1);  
4   }  
5 }
```

Image 9


 2f1734be4e2019c86455ec70b0004cdc	57	bob	Person
--------------------------------------------------------------------------------------------------------------------	----	-----	--------

Image 10

```
1 function (doc) {  
2   if(doc.type == "Person" && doc.age < 24) {  
3     emit(doc._id, 1);  
4   }  
5 }
```

Image 11

 2f1734be4e2019c86455ec70b0002563	22	Zach	Person
 2f1734be4e2019c86455ec70b000363d	23	John	Person

Image 12

```
C:\Users\Zach\Desktop\UW_Classes\2023Spring\Data514\SystemInvestigationProject\Part2\Part2.py
1 import requests
2 username = "admin"
3 password = "insert password here"
4
5 #Get database information
6 x = requests.get("http://" + username + ":" + password + "@127.0.0.1:5984")
7 print(x.text)
8
9 #Create database table
10 x = requests.put("http://" + username + ":" + password + "@127.0.0.1:5984/testdbpython")
11 print(x.text)
12
13 #Add documents to table
14 data = {"name": "Zach", "age": 22, "type": "person"}
15 x = requests.put("http://" + username + ":" + password + "@127.0.0.1:5984/testdbpython/001", json=data)
16 print(x.text)
17 data = {"name": "john", "age": 23, "type": "person"}
18 x = requests.put("http://" + username + ":" + password + "@127.0.0.1:5984/testdbpython/002", json=data)
19 print(x.text)
20
21 #Create design document
22 data = {"_id": "_design/testdesignndoc1", "language": "javascript"}
23 x = requests.put("http://" + username + ":" + password + "@127.0.0.1:5984/testdbpython/_design/testdesignndoc1", json=data)
24 print(x.text)
25
26 #Add view to existing document
27 data = {
28     "_id": "_design/testdesignndoc1",
29     "_rev": "1-8c8e8ec35da23da3e441cd3350006d92",
30     "language": "javascript",
31     "views": {
32         "GetPeople": {
33             "map": "function(doc) {if(doc.type == 'Person') {emit(doc._id, 1);}}"
34         }
35     }
36 }
37 x = requests.put("http://" + username + ":" + password + "@127.0.0.1:5984/testdbpython/_design/testdesignndoc1", json=data)
38 print("add view", x.text)
39
40 #Run view
41 x = requests.get("http://" + username + ":" + password + "@127.0.0.1:5984/testdbpython/_design/testdesignndoc1/_view/GetPeople")
42 print(x.text)
43
44 #Remove document
45 x = requests.delete("http://" + username + ":" + password + "@127.0.0.1:5984/testdbpython/002?rev=1-67e570bcc765cd935e8d105e565f5a1c")
46 print(x.text)
47
```

Image 13

GTFS (Database)		Design document View1, View2, View3	
Agency document (3) Total	Calendar document (56) Total	Calendar dates document (598) Total	Fare attributes document (7) Total
<pre>{ "id": "2f1734be4e2019c86455ec70b0002563", "_rev": "1-efcc4c449c8d6c7733ac7055c38dae49", "doc_type": "Agency", "agency_id": "TRIMET", "agency_name": "TriMet", "agency_url": "https://trimet.org/", "agency_timezone": "America/Los_Angeles", "agency_lang": "en", "agency_phone": "503-238-RIDE", "agency_fare": "https://trimet.org/fares/", "agency_email": "customerservice@trimet.org", "bikes_policy_url": "https://trimet.org/bikes/bikepolicies.htm" }</pre>	<pre>{ "id": "2f1734be4e2019c86455ec70b0002563", "_rev": "1-efcc4c449c8d6c7733ac7055c38dae49", "doc_type": "Calendar", "service_id": "C.645", "monday": 0, "tuesday": 0, "wednesday": 0, "thursday": 0, "friday": 0, "saturday": 0, "sunday": 0, "start_date": 20230416, "end_date": 20230429 }</pre>	<pre>{ "id": "2f1734be4e2019c86455ec70b0002563", "_rev": "1-efcc4c449c8d6c7733ac7055c38dae49", "doc_type": "Calendar_date", "service_id": "C.647", "date": 20230507, "exception_type": 1 }</pre> <pre>{ "id": "2f1734be4e2019c86455ec70b0002563", "_rev": "1-efcc4c449c8d6c7733ac7055c38dae49", "doc_type": "stop_feature", "stop_id": 2, "feature_name": "Crosswalk near stop", }</pre>	<pre>{ "id": "2f1734be4e2019c86455ec70b0002563", "_rev": "1-efcc4c449c8d6c7733ac7055c38dae49", "doc_type": "Fare_attribute", "fare_id": "B", "agency_id": "TRIMET", "price": "2.5", "currency_type": "USD", "payment_method": 1, "transfers": "null", "transfer_duration": 9000 }</pre>
Fare rules document (12) Total	Route directions document (184) Total	Routes document (94) Total	Shapes document (989919) Total
<pre>{ "id": "2f1734be4e2019c86455ec70b0002563", "_rev": "1-efcc4c449c8d6c7733ac7055c38dae49", "doc_type": "Fare_rule", "fare_id": "B", "origin_id": "B", "route_id": "null", "contains_id": "208" }</pre>	<pre>{ "id": "2f1734be4e2019c86455ec70b0002563", "_rev": "1-efcc4c449c8d6c7733ac7055c38dae49", "doc_type": "Route_direction", "route_id": 1, "direction_id": 0, "direction_name": "To Gresham" }</pre>	<pre>{ "id": "2f1734be4e2019c86455ec70b0002563", "_rev": "1-efcc4c449c8d6c7733ac7055c38dae49", "doc_type": "Route", "route_id": 1, "agency_id": "TRIMET", "route_short_name": "1", "route_long_name": "Vermont", "route_type": 3, "route_url": "https://trimet.org/schedules/r001.htm", "route_color": "61A60E", "route_text_color": "FFFFFF", "route_sort_order": 500, }</pre>	<pre>{ "id": "2f1734be4e2019c86455ec70b0002563", "_rev": "1-efcc4c449c8d6c7733ac7055c38dae49", "doc_type": "Shape", "shape_id": 534052, "shape_pt_lat": 45.47623, "shape_pt_lon": -122.721885, "shape_pt_sequence": 1, "shape_dist_traveled": 0.0, }</pre>
Trips document (64647) Total	Stop times document (3047657) Total	Stops document (6558) Total	Transfers document (3730) Total
<pre>{ "id": "2f1734be4e2019c86455ec70b0002563", "_rev": "1-efcc4c449c8d6c7733ac7055c38dae49", "doc_type": "trip", "route_id": 1, "service_id": "W.645", "trip_id": 12292061, "trip_short_name": "null", "direction_id": 0, "block_id": 6174, "shape_id": 534051, "trip_type": "null", "wheelchair_accessible": 1 }</pre>	<pre>{ "id": "2f1734be4e2019c86455ec70b0002563", "_rev": "1-efcc4c449c8d6c7733ac7055c38dae49", "doc_type": "stop_time", "trip_id": 12292061, "arrival_time": "15:10:00", "departure_time": "15:10:00", "stop_id": 13170, "stop_sequence": 1, "stop_headsign": "Vermont Shattuck Loop via Maplewood", "pickup_type": 0, "drop_off_type": 1, "shape_dist_traveled": 0.0, "timepoint": 1, "continuous_drop_off": "null", "continuous_pickup": "null" }</pre>	<pre>{ "id": "2f1734be4e2019c86455ec70b0002563", "_rev": "1-efcc4c449c8d6c7733ac7055c38dae49", "doc_type": "stop", "stop_id": 2, "stop_code": 2, "stop_name": "A Ave & Chandler", "tts_stop_name": "ae avenue & chandler", "stop_desc": "Eastbound Stop in Lake Oswego (Stop ID 2)", "stop_lat": 45.420609, "stop_lon": -122.675671, "zone_id": "B", "stop_url": "https://trimet.org/home/stop/2", "location_type": 0, "parent_station": "null", "direction": "East", "position": "Nearside" }</pre>	<pre>{ "id": "2f1734be4e2019c86455ec70b0002563", "_rev": "1-efcc4c449c8d6c7733ac7055c38dae49", "doc_type": "transfer", "from_stop_id": 46, "to_stop_id": 5889, "transfer_type": 0 }</pre>

Reference links

Download link: <https://neighbourhood.ie/download-apache-couchdb-win/>

CouchDB documentation: <https://docs.couchdb.org/en/stable/>

CouchDB replication: <https://docs.couchdb.org/en/stable/replication/index.html>

GTFS dataset: <http://developer.trimet.org/GTFS.shtml>