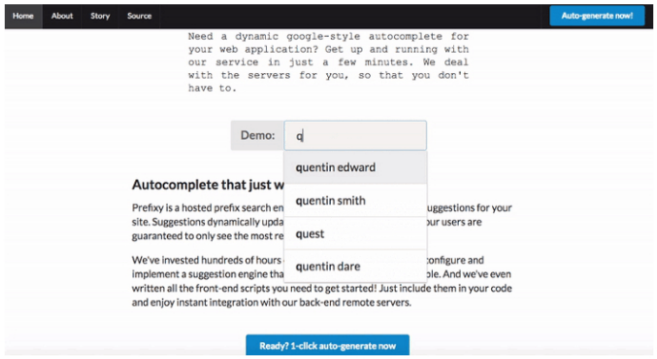
How We Built Prefixy: A Scalable Prefix Search Service for Powering Autocomplete

Prefixy Team Jan 20, 2018 · 16 min read



Prefixy in action

. . .

If you've ever googled something, you probably take it for granted that suggestions appear based on what you've typed so far. Yet, these suggestions are an essential part of the googling experience. It would almost feel more weird if you were typing out a search query and autocomplete suggestions *did not* show up.

Indeed, Google's autocomplete is powerful and useful, so it's no surprise that there are many open source implementations of autocomplete widely available (e.g. Twitter's typeahead.js and SeatGeek's Soulheart to name a couple).

But these preexisting solutions are limited, because they often work off of a fixed data set of suggestions. Moreover, almost all of them leave it up to you to figure out how to store suggestions on the backend.

This led us to ask the question "How can we create a hosted service that makes it trivially easy for any developer to equip any search field on their app with Google style autocomplete?" In just a few steps a developer should have an autocomplete that displays the most popular crowd-sourced suggestions for any given search query.

This question began a deep dive analysis into the data structures, algorithms, and system design, for what would ultimately become Prefixy.

. . .

What is Prefixy?

Prefixy is a hosted prefix search service that powers autocomplete suggestions. It dynamically updates and ranks suggestions based on popularity. With just a simple code snippet, any developer can utilize our service to set up powerful and adaptive autocomplete on their application.

If you'd like to take Prefixy for a spin: try it here!

Who Are We?

Who Are We?

Walid Wahed
Software Engineer
(NYC)

Tiffany Han
Software Engineer
(SF)

Jay Shenk
Software Engineer
(Dayton)

Before we get into it, allow us to introduce ourselves. We are a remote team of three software engineers. We also happen to be searching for our next opportunities, so please don't hesitate to reach out to Walid Wahed, Tiffany Han, or Jay Shenk if you think we'd be a good fit for your team!

Takeaways of This Article

This article will cover the following topics:

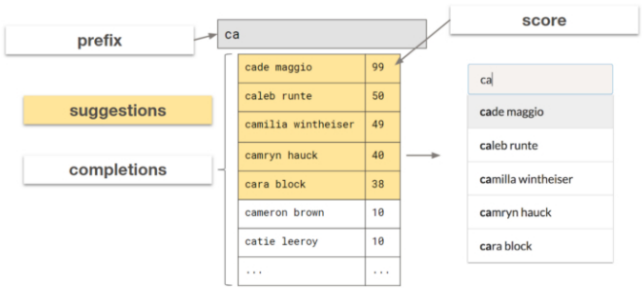
- How we approached the R&D phase of the project
- How we thought about tradeoffs as we designed and built a complex system from scratch
- How we evaluated data structures, algorithms, and data stores for our particular use case
- What we did to ensure that our system can scale

Even if you never end up implementing your own system for autocomplete, many of the processes and lessons we share here are applicable to almost any complex software engineering project.

Anatomy of a Prefix Search

Let's first define some important terms, since these will be helpful in understanding the rest of this article. Take a look at the following definitions along with the diagram below:

- **prefix:** what a user has typed so far
- **completions:** all possible ways a given prefix can be completed in order to form a word or phrase
- **suggestions:** the top ranked completions which will be presented to the user
- **score:** an integer representing the popularity of a given completion
- **selection:** the suggestion that the user chooses or a new completion that the user submits



Now that we're all on the same page as far as terminology, let's jump into it!

. . .

Setting Design Goals and Clarifying Requirements

From the beginning, it was obvious that there were two major requirements to address in our system.

1. **Lightning fast reads:** As a user types their search query, suggestions should appear nearly instantaneously. If the suggestions appear too slowly, the autocomplete is no longer useful. For instance, Facebook found that an autocomplete needs to return suggestions in less than 100 milliseconds.
2. **Relevancy of suggestions:** We need to return the most relevant suggestions to the user. We define "relevant" as the most popular suggestions based on what other users of the same app have recently searched for.

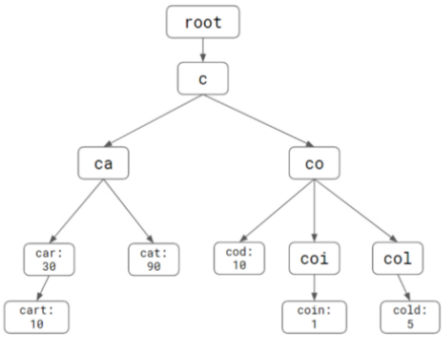
These requirements imply that we need to prioritize speed of reads over almost everything else. We also need some sort of ranking algorithm in order to keep track of the most popular suggestions to be returned.

. . .

Data Structures & Algorithms

After clarifying requirements, our next task was to determine the best data structures and algorithms to support these requirements. We knew that this was a mission critical component in achieving our goals, so we dedicated significant time to this analysis.

Trying Out a Trie

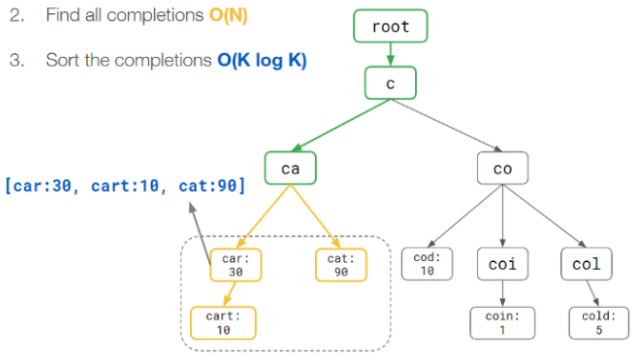


Tries were interesting to us from the get-go, since they happen to be a natural fit for prefix search. Tries allow for O(L) lookup of a prefix, where L is the length of the prefix. Tries also allow for some space savings, since all descendants of a given node share that node as a common prefix.

How would searching for completions work with a trie?

Let's say were looking for all completions that start with "ca". The steps are:

1. Find the prefix **O(L)**
2. Find all completions **O(N)**
3. Sort the completions **O(K log K)**



So the total time complexity is O(L) + O(N) + O(K log K), where L is the length of the prefix, N is the total number of nodes in the trie, and K is the

number of completions associated with any given prefix.

The main bottleneck of this approach is the O(N) finding of all completions. The fundamental issue is that as the system's library of completions grows, it will take longer and longer to return suggestions. This is far too slow to support our requirements, since we will capture ever more completions as the service is used.

Top highlight

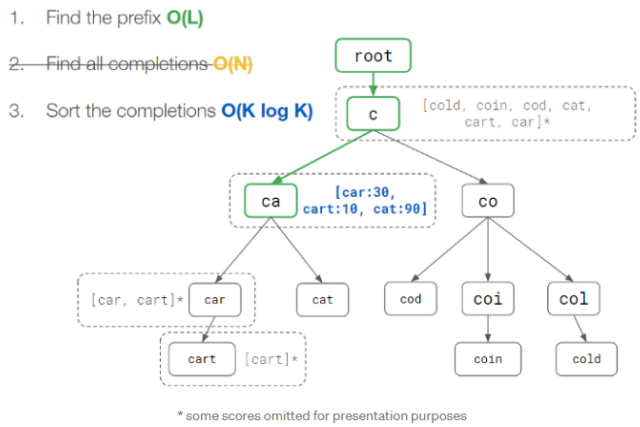
You may be wondering, "why is it O(N) to find the completions?" It is O(N) because after traversing down to the prefix node, we still have to visit every single one of its children to find all the completions. In the worst-case scenario, this is an O(N) operation.

Naturally this led us to wonder how we can remove the O(N) bottleneck.

A Little Precomputation

We found that we can store a prefix along with all of its completions together in the same node. This eliminates the O(N) bottleneck, since we no longer have to traverse to find all the completions.

Now it takes just two steps to search for completions:



* some scores omitted for presentation purposes

The time complexity is now significantly better, but this does come at a cost. We consume more space, since we'll store many of the same completions in different prefix nodes. For instance, the completion "cat" will be stored twice: in "c"'s node and in "ca"'s node. This solution also requires more writes to keep our completions up to date. But since speed of reads was our priority, we were happy to make these trade offs.

So far so good, but can we do better than this? As you're about to see, it turns out the answer is a resounding "yes!"

Setting Limits on the Amount of Data We Store

Limiting L, or the max length of prefixes we store

We can reasonably limit the max length of prefixes we store. For instance, consider the following completion:

> *"Tyrannosaurus Rex lived during the late Cretaceous period."*

Search queries of this length are few and far between. As such, you can imagine that there are diminishing returns to storing *every single prefix* contained within this completion. Thus, we can safely set the max limit of L to around 20 characters (or some other reasonable amount).

Limiting K, or the number of completions we store for any given prefix

Similarly, we can reasonably limit the amount of completions we store for a given prefix. Remember that most of the time, we only need to show the top

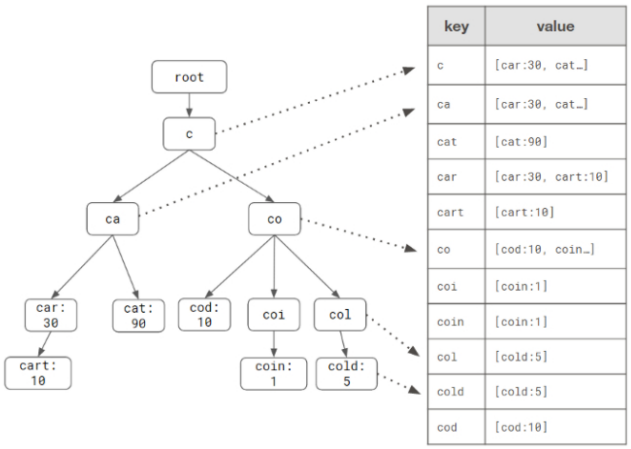
5 or 10 suggestions. This means we only need to store enough completions to enable accurate popularity ranking. We will get into the nitty gritty of the ranking algorithm we use later. But for now, trust that storing around 50 completions for a given prefix is accurate enough for our needs.

Summary

Limiting L and K saves us a whole lot of space. What's more important though, is that it improves our time complexity significantly. By holding L and K constant, our time to search improves from O(L) + O(K log K) to O(1) + O(1). This reduces to O(1) time!

Prefix Hash Tree

At this point, we were quite happy with the O(1) reads as this will scale nicely. Even so, we did find one last way to optimize our solution even further. Instead of storing our data in a conventional trie, we realized we can store our data in a hybrid data structure called a Prefix Hash Tree.



To do this, we map every prefix to a hash key. We also map the prefix's completions to the hash value associated with that key.

This gives us a few advantages over the conventional trie. We can access any prefix in a single O(1) step. Moreover, the key/value structure is easy to implement in a NoSQL data store (we'll get to why that's beneficial soon!).

There *are* a few disadvantages to using the prefix hash tree though. We lose the conventional trie's ability to share common characters of prefixes. A hash also inherently requires more space in order to maintain its load factor. But again, we were more than willing to trade space for speed of reads.

In the end, we decided to use the prefix hash tree as our primary data structure to store prefixes. The following table summarizes our data structure analysis so far:

	Search	Insert	Update / Delete	Space Consideration
Naive Trie	O(L) + O(N) + O(K log K)	O(L)	O(L)	completions share prefixes
Trie With Completions	O(L) + O(K log K)	O(L)	O(LK)*	bucket of completions at each prefix node
hold L constant	O(1) + O(K log K)	O(1)	O(K)	reduces number of prefixes we store
hold K constant	O(1) + O(1)	O(1)	O(1)	caps size bucket of completions size
Prefix Hash Tree w/ completions, constant L & K	O(1) *no traversal!*	O(1)	O(1)	slightly more space to accommodate hash table space allocation, plus have to duplicate prefixes ("c", "ca", "car")

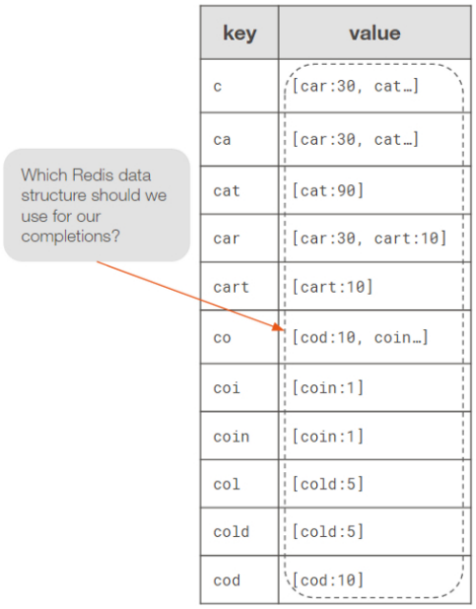
* need to find completion at each prefix when updating or deleting

. . .

Choosing Our Primary Data Store: Enter Redis

As we just mentioned, one benefit of the prefix hash tree is that it can easily map to a key/value NoSQL database. In our case, we chose to use Redis for two main reasons:

1. Redis is an entirely in-memory data store, which means it can perform reads and writes much faster than a hard disk data store. This is great for our use case, since fast reads are a priority for us.
2. Since Redis has a number of native in-memory data structures, we can use one of these data structures to store our completions. If we do this right, we may be able to work with our completions in a much simpler and more efficient way.



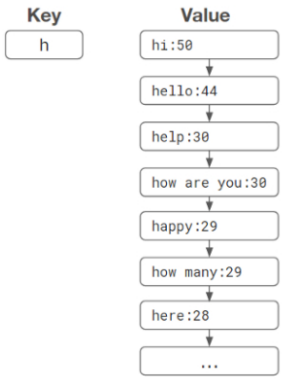
So now the question we turned to was, "Which Redis data structure should we use to store our completions?"

Choosing a Redis Data Structure to Hold Our Completions

To store completions, there were really only two competing options that we considered: the Redis List, and the Redis Sorted Set. We're now going to compare these two data structures in light of two main functions in our system: (1) `SEARCH`, which we invoke to return suggestions back to the user; and (2) `INCREMENT`, which we invoke to increase a completion's score after a user selects a suggestion.

Option 1: Redis List

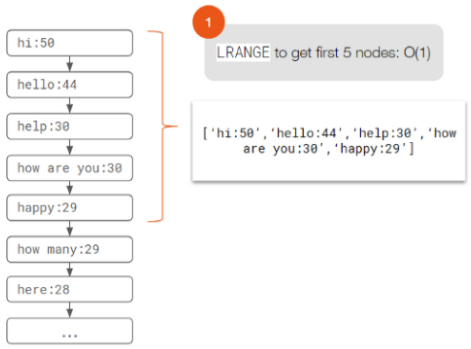
A list in Redis is a doubly-linked list, where each node holds a single string. This implies that if we choose the list, we would have to store both a completion and its score together within this string.



Searching with the Redis List

To get the top suggestions for a given prefix, we just need to issue a single Redis `LRANGE` command. As long as we keep each list sorted, the top

suggestions will always be at the head of the list. This means we'll be able to access any prefix's suggestions in O(1) time, which is perfect for our use case.



Searching a Redis List

Incrementing with the Redis List

On the other hand, incrementing requires the following steps:

Incrementing with a Redis List

There are several disadvantages to this algorithm:

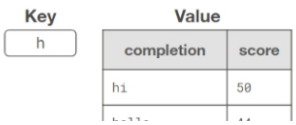
- The algorithm has an overall time complexity of O(K), and it requires at least two round trips to Redis.
- The first request requires us to carry a large payload, since we need to retrieve the entire list.
- We need to take care of sorting and de-duping within our own application logic.
- We may run into concurrency issues with this many steps for the update logic.

Summary

The Redis List offers O(1) search, which we like. However, updating the list presents us with a whole host of problems. Let's see if the Redis Sorted Set can do any better.

Option 2: Redis Sorted Set

The Redis Sorted Set is a collection of items that each consist of a string and a corresponding score. Redis maintains order in the sorted set, first by score and then by lexicographical order. Moreover, Redis also takes care of deduping within the data structure, so each item is guaranteed to be unique.

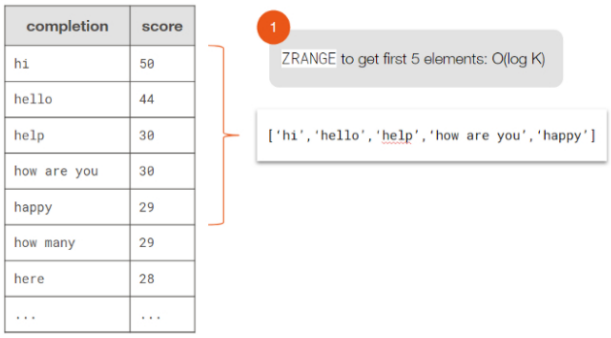


hello	44
help	30
how are you	30
happy	29
how many	29
here	28
...	...

Sounds promising indeed! Now let's examine how each of our key functions would work.

Searching with the Redis Sorted Set

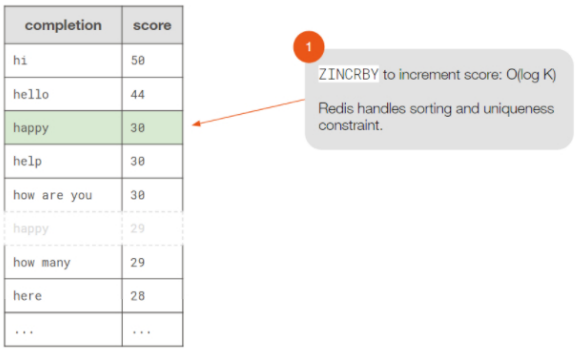
To search for suggestions, we would simply issue the Redis `ZRANGE` command. This will return suggestions back to us in O(log K) time. Not quite as good as O(1), but still pretty awesome.



Searching a Redis Sorted Set

Incrementing with the Redis Sorted Set

To our delight, we found that to increment a score, all we have to do is issue a single `ZINCRBY` command! And like the search, this update also happens in O(log K) time.



Performing an increment with a Redis Sorted Set

The advantages of this one step algorithm include: fewer round trips to Redis, smaller payloads, and lower chance of concurrency issues. Redis will also maintain order and uniqueness for us, which is both simpler and more efficient than doing it ourselves.

Verdict: Sorted Set

The Redis Sorted Set provides us with fast reads as well as simple and efficient incrementing. These considerations made our eventual choice to go with the Redis Sorted Set a no-brainer.

An Algorithm to Maintain Our K Limit

As you just saw, incrementing a score in a Redis Sorted Set simply involves

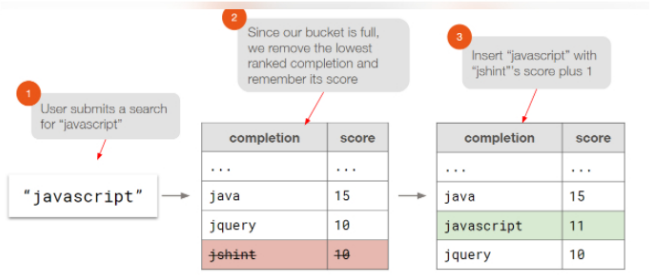
using the `ZINCRBY` command. If the completion specified already exists in the sorted set, its score will be incremented. On the other hand, if the completion is a brand new completion, it will be added to the sorted set.

Remember though that we decided to limit K, the number of completions stored for any given prefix. What should we do when we reach our K limit? We need some algorithm in our application logic in order to handle this scenario.

Here we are indebted to Salvatore Sanfilippo, the creator of Redis. We use a solution suggested by him in one of his blog posts.

The steps are:

1. When the K limit is reached and the completion to increment is a brand new completion, remove the completion with the lowest score.
2. Add the new completion with the removed completion's score plus 1.



Why use the removed completion's score plus 1? Doing this ensures that a new completion will not be immediately replaced by a subsequent new completion. This is important because we need to give all completions a shot at rising to the top suggestions.

. . .

Expanding Our Persistence Strategy

At this point we realized that as the service grows, keeping all of the data in memory may quickly become expensive. Further, we reasoned that not all prefixes actually need to be in memory all the time. So why not keep only fresh prefixes in memory while persisting stale prefixes on disk?

As such, we decided to treat Redis in a cache-like manner by setting an LRU eviction policy in Redis. So now Redis will hold only the most recently used prefixes and once max memory is reached, Redis will evict the *least* recently used prefixes.

Using MongoDB for Hard Disk Persistence

For hard disk persistence, we decided to use MongoDB. We chose MongoDB primarily because its key-document metaphor mapped nicely to the key-value setup we utilized in Redis. Having similar models of abstraction amongst data stores is convenient, because it makes the persistence process easier to reason about.

Working with MongoDB: dealing with a cache miss

Now when we perform a search, we first check Redis. Redis will usually have what we're looking for due to the efficacy of the LRU policy. So in most cases, we can return results after just a single call to Redis.

But in case we can't find our data in Redis, we now check against the complete set of prefixes in MongoDB. Once we find what we're looking for in MongoDB, we then "reinstate" that data back into memory. We do this because someone recently searched for the data, so technically it's no longer stale.

Updating Redis in case we have a cache miss

After the Redis update is complete, we simply pull again from Redis, and return top suggestions back to the user!

Pulling from Redis again now that it's updated

The logic for searching in the event of a cache miss requires us to perform quite a few more steps. The good news is that since we update Redis in every cache miss, all subsequent requests for the same data can now be returned from memory. Thus, the slower and more complicated logic for searching only happens in a relatively small number of cases.

Working with MongoDB: incrementing completions

Now our increment logic also requires a few more steps. On a high level, we always first write to Redis. Note that the sequence of steps is important, because we still rely on the Redis Sorted Set data structure to maintain order and uniqueness for us. After Redis does its thing, we then pull the now up-to-date completions from Redis. Finally, we serialize these completions and push them into MongoDB.

We always update Redis first since it handles ranking for us

And we're done! Both Redis and MongoDB are updated now.

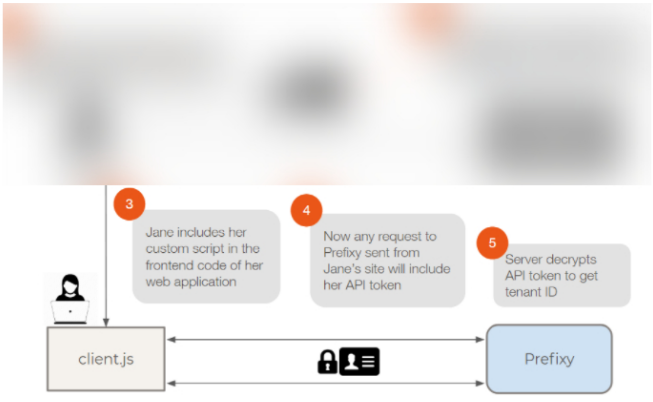
. . .

Implementing Support for Multi-Tenancy

There was just one last hurdle to jump in order to reach our final goal of creating a hosted service: we had to provide support for multi-tenancy in order to keep the data for different users of the service separate.

Adding a token generator server to our system

To implement multi-tenancy, we first added a token generator server to our system.



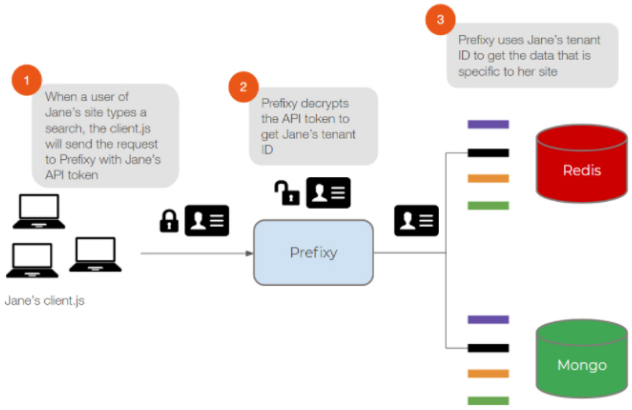
When a developer requests it, we generate a unique tenant ID and then encrypt it into a JSON Web Token (JWT). We embed this JWT into a custom script that initializes the front end javascript client. Finally, we instruct the developer to include this custom script in the front end code of her web app.

Generating tokens by encrypting tenant IDs

Now the front end client will automatically issue HTTP requests to Prefixy at the appropriate times. All these requests will contain the developer's JWT, which our Prefixy server will use to get the developer's tenant ID.

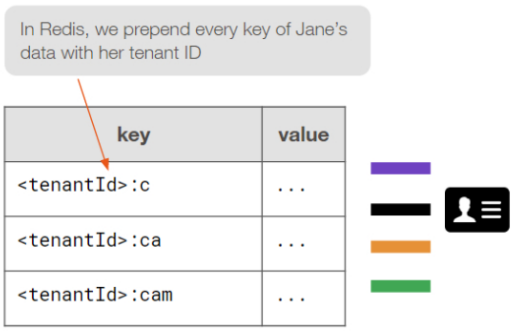
Our Prefixy server receives the token and decrypts it

But how do we use the tenant ID to implement multi-tenancy on the backend? Good question! We use the tenant ID to namespace the user's data.



Implementing multi-tenancy in Redis

In Redis, we prepend every key of a developer's data with her tenant ID. So on the backend, all of our Redis keys look like this: `<tenantId>:<prefix>`.



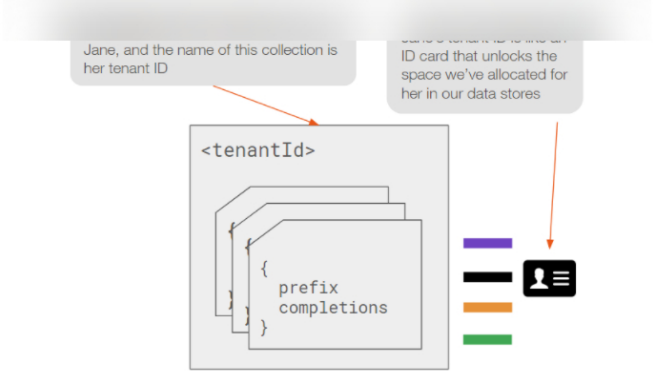
Multi-tenancy in Redis

Now when a search request from the developer's app comes in, we prepend her tenant ID to the search query. The query will now be in the right format (e.g. `<tenantId>:<prefix>`), and we simply do a one-to-one key/value lookup.

Prepending a tenant ID to each search query

Implementing multi-tenancy in MongoDB

In MongoDB, we allocate a collection to each tenant ID. Collections are provided to us by MongoDB as a built-in way to namespace our data.



Multi-tenancy in MongoDB

Now to search in MongoDB, we simply chain the `find` method off the appropriate collection. The search will now be name-spaced under that collection.

Mongo searches are namespaced under a tenant's collection

. . .

How the Front End Client Communicates with Prefixy

So how does the front end client communicate with Prefixy? Here's a quick glimpse into the front end client code.

Fetching suggestions

In order to fetch suggestions, we set up an event listener which listens for any typing being done on the search box. When this event is fired, the client makes a GET request to the appropriate API endpoint on our Prefixy server.

Listening for typing and fetching suggestions

Once the suggestions come back, we invoke the `draw` method. This method takes each suggestion and appends a corresponding list item to the dropdown list of suggestions.

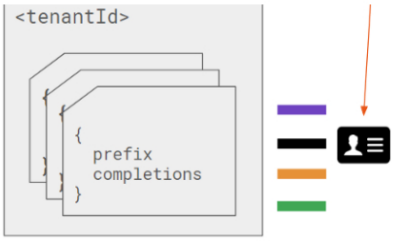
Iterating over the JSON returned to draw suggestions

. . .

Final System Architecture

Phew! That's about all folks. We made it to the end of this whirlwind tour. Our final system architecture includes the following components and is pictured below:

- **Prefixy:** home of our core application logic, and comes with a CLI that we use for admin tasks
- **Redis:** keeps high priority suggestions in memory and handles ranking of the completions
- **MongoDB:** the system of record that persists all suggestions



Multi-tenancy in MongoDB

Now to search in MongoDB, we simply chain the `find` method off the appropriate collection. The search will now be name-spaced under that collection.

Mongo searches are namespaced under a tenant's collection

. . .

```
    return completions[0].completions;
}
```

mongoMultitenancy.js hosted with ❤ by GitHub view raw

Mongo searches are namespaced under a tenant's collection

. . .

How the Front End Client Communicates with Prefixy

So how does the front end client communicate with Prefixy? Here's a quick glimpse into the front end client code.

Fetching suggestions

In order to fetch suggestions, we set up an event listener which listens for any typing being done on the search box. When this event is fired, the client makes a GET request to the appropriate API endpoint on our Prefixy server.

```
function valueChanged() {
  const value = this.input.value;
  if (value.length >= this.minChars) {
    this.fetchSuggestions(value, suggestions => {
      this.visible = true;
      this.suggestions = suggestions;
      this.draw();
    });
  } else {
    this.reset();
  }
}

function fetchSuggestions(query, callback) {
  const params = { prefix: query, token: this.token };
  if (this.suggestionCount) {
    params.limit = this.suggestionCount;
  }
  axios.get(this.completionsUrl, { params })
    .then(response => callback(response.data));
}
```

clientFetch.js hosted with ♥ by GitHub view raw

Listening for typing and fetching suggestions