Design a URL shortener

You know, like bit.ly.

Let's call it ca.ke!

Step 1 is to scope the project. System design questions like this are usually intentionally left open-ended, so you have to ask some questions and make some decisions about exactly what you're building to get on the same page as your interviewer.

So, what are we building? What features might we need?

Features

Is this a full web app, with a web interface? No, let's just build an API to start.

Since it's an API, do we need authentication or user accounts or developer keys? No, let's just make it open to start.

Can people modify or delete links? Let's leave that out for now.

If people can't delete links...do they persist forever? Or do we automatically remove old ones? First, it's worth considering what policies we could use for removing old ones:

1. We could remove links that were *created* some length of time ago...like 6 months.
2. We could remove links that haven't been *visited* in some length of time...like 6 months.

(2) seems less frustrating than (1). Are there cases where (2) could still frustrate users? If a link is on the public web, it's likely to get hit somewhat regularly, at least by spiders. But what if it's on the private web (e.g. an internal "resources" page on a private university intranet)? Or...what if someone printed a bunch of *pamphlets* that had the URL on it, didn't give out any pamphlets for a few months, then started giving them out again? That seems like a pretty reasonable thing that might happen (putting a URL on a printed piece of paper is a great reason to use a link shortener!) and having the link suddenly stop working would be quite frustrating for the user. Worse, what if a *book* already had the shortlink printed in a million copies? So let's let links exist forever.

Should we let people *choose* their shortlink, or just always auto-generate it? For example, say they want `ca.ke/parkers-resume`. Let's definitely support that.

Do we need analytics, so people can see how many people are clicking on a link, etc? Hmmm, good idea. But let's leave it out to start.

It's okay if your list of features was different from ours. Let's proceed with these requirements so we're working on the same problem.

Next step: Design goals. If we're designing something, we should know what we're optimizing for! What are we optimizing for?

Design Goals

Here's what we came up with:

1. We should be able to store a *lot* of links, since we're not automatically expiring them.
2. Our shortlinks should be as short as possible. The whole point of a link shortener is to make *short* links! Having shorter links than our competition could be a business advantage.
3. Following a shortlink should be *fast*.
4. The shortlink follower should be resilient to load spikes. One of our links might be the top story on Reddit, for example.

> It's worth taking a moment to really think about the *order* of our goals. Sometimes design goals are at odds with each other (to do a better job of one, we need to do a worse job of another). So it's helpful to know which goals are more important than others.

It's okay if your list wasn't just like ours. But to get on the same page, let's move forward with these design goals.

Next step: building the data model. Think about the database schema or the models we'll want. What things do we need to store, and how should they relate to each other? This is the part where we answer questions like "is this a many-to-many or a one-to-many?" or "should these be in the same table or different tables?"

Data Model

It's worthwhile to be careful about how we name things. This'll help us communciate clearly with our interviewer, and it'll show that we care about using descriptive and consistent names! Many interviewers look for this.

Let's call our main entity a **`Link`**. A `Link` is a mapping between a `shortLink` on our site, and a `longLink`, where we redirect people when they visit the `shortLink`.

```
Link
- shortLink
- longLink
```

The `shortLink` could be one we've randomly generated, or one a user chose.

Of course, we don't need to store the *full* `ShortLink` URL (e.g. `ca.ke/mysite`), we just need to store the "slug"—the part at the end (e.g. "`mysite`").

So let's rename the `shortLink` field to "`slug`."

```
Link
- slug
- longLink
```

Now the name `longLink` doesn't make as much sense without `shortLink`. So let's change it to `destination`.

```
Link
- slug
- destination
```

And let's call this whole model/table `ShortLink`, to be a bit more specific.

```
ShortLink
- slug
- destination
```

> Investing time in carefully naming things from the beginning is always impressive in an interview. A *big* part of code readability is how well things are named!

Next: sketching the code. Don't get hung up on the details here—pseudocode is fine.

Think of this part as *sprinting to a naive first draft design*, so you and your interviewer can get on the same page and have a starting point for optimizing. There may be things that come up as you go that are clearly "tricky issues" that need to be thought through. Feel free to skip these as you go—just jot down a note to come back to them later.

Our main goal here is to come up with a skeleton to start building things out from. Think about what endpoints/views we'll need, and what each one will have to do.

Views/Pages/Endpoints

First, let's make a way to *create* a `ShortLink`.

Since we're making an API, let's make it REST-style. If REST is one of those "I've heard it a bunch but only half know what it means" things for you, I highly recommend reading up on it (http://www.restapitutorial.com/lessons/restquicktips.html).

In normal REST style, our endpoint for creating a `ShortLink` should be named after the entity we're creating. Versioning apis is also a reasonable thing to do. So let's put our creation endpoint at **`ca.ke/api/v1/shortlink`**.

To create a new `ShortLink`, we'll send a POST request there. Our POST request will include one required argument: the `destination` where our `ShortLink` will point. It'll also optionally take a slug argument. If no slug is provided, we'll generate one. The response will contain the newly-created `ShortLink`, including its `slug` and `destination`.

```
$ curl --data '{"destination": "interviewcake.com"}' https://ca.ke/api/v1/shortlink
{
  "slug": "ae8uFt",
  "destination": "interviewcake.com"
}
```

In usual REST style, we should allow GET, PUT, PATCH, and DELETE requests as well to read, modify, and delete links. But since that's not a requirement yet, we'll just reject non-POST requests with an error 501 ("not implemented") (https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.5.2) for now.

So our endpoint might look something like this (pseudocode):

```
def shortlink(request):
    if request.method is not 'POST':
        return Error501

    destination = request.data.destination

    # if they included a slug, use that
    if request.data.slug:
        slug = request.data.slug

    # else, make them one
    else:
        slug = generate_random_slug()

    DB.insert('Links', {'slug': slug, 'destination': destination})

    response_body = {
        'slug' : slug,
        }

    return Success200(json.format(response_body))
```

Of course, we haven't defined exactly how `generateRandomSlug()` works. Considering it a bit, it quickly becomes clear this is a pretty tangled issue. We'll have to figure out:

1. What characters can we use in randomly generated slugs? More possible characters means more possible random slugs without making our shortlinks longer. But what characters are allowed in URLs?

2. How do we ensure a randomly generated slug hasn't already been used? Or if there *is* such a collision, how do we handle it?

So let's jot down these questions, put them aside, and come back to them after we're done sketching our general app structure.

Second, let's make a way to *follow* a `ShortLink`. That's the whole point, after all!

Our shortened URLs should be as short as possible. So as mentioned before, we'll give them this format: **`ca.ke/$slug`**.

Where `$slug` is the slug (either auto-generated by us or specified by the user). We could make it clearer that this is a redirect endpoint, by using a format like `ca.ke/r/$slug`, for example. But that adds 2 precious characters of length to our shortlink URLs!

> One potential challenge here: if/when we build a web app for our service, we'll need some way of differentiating our own pages from shortlinks. For example, if we want an about page at `ca.ke/about`, our back-end will need to know "about" isn't just a shortlink slug. In fact, we might want to "reserve" or "block" shortlinks for pages we think we might need, so users don't grab URLs we might want for our own site. *Alternately* we could just say *our* pages have paths that're always prefixed with something, like `/w/`. For example, `ca.ke/w/about`.

The code for the redirection endpoint is pretty simple:

```
def redirect(request):
    destination = DB.get('Links', 'destination', {'slug': request.path})
    return Redirect302(destination)
```

Next: slug generation. Let's return to those questions we came up with about slugs. How long should they be, what characters should we allow, and how should we handle random slug collisions?

Slug generation

> A note about methodology: Our default process for answering questions like this is often "make a reasonable guess, brainstorm potential issues, and revise." That's fine, but sometimes it feels more organized and impressive to do something more like "brainstorm design goals, then design around those goals." So we'll do that.

Let's look back up at the design goals we came up with earlier. The first two are immediately relevant to this problem:

1. We should be able to store a *lot* of links.
2. Our shortlinks should be as short as possible.

Looking at a few examples, we can quickly notice that the more characters we allow in our shortlinks, the more *different* `ShortLink`s we can have without making our `ShortLink`s *longer*. Specifically, if we allow c different characters, for n-character-long slugs we have c^n distinct possibilities.

> **How did we get that math?**
>
> We drew out a few examples and looked for patterns.
>
> It helps to start with small numbers. Suppose we only allowed 2 different characters for our slugs: 'a' and 'b', How many possible 1-character slugs would we have? 2: 'a' and 'b'.
>
> How many possible 2-character slugs? Well, we have 2 possibilities for the first character, and for *each* of those 2 possible first characters, we have 2 possible second characters. That's $2 * 2 = 4$ possibilities overall.

> How many possible 3-character slugs? 2 possibilities for the first character, and for *each* of those 2 possible first characters, we have 2 possible second characters, and for each of those possible first and second characters, another 2 possible third characters. That's $2 * 2 * 2 = 8$ possibilities overall.
>
> Looks like this is a multiplication thing. Or in fact, a *power* thing. In general, we have 2^n possible n-character slugs, if we only allow 2 possible choices (a and b) for each character.
>
> And if we want to allow c different possible characters, instead of just 2? c^n possibilities for an n-character-long slug.

So if we're trying to accomodate as many slugs as possible, we should allow as many characters as we can! So let's do this:

1. Figure out the max set of characters we can allow in our random shortlinks.
2. Figure out how many distinct shortlinks we want to accomodate.
3. Figure out how long our shortlinks must be to accomodate that many distinct possibilities.

> Sketching a process like this before jumping in is hugely impressive. It shows organized, methodical thinking. Whenever you're not sure how to proceed, take a step back and try to write out a process for getting to the bottom of things. It's fine if you end up straying from your plan—it'll still help you organize your thinking.

What characters can we allow in our randomly-generated slugs?

What are the *constraints* on c? Let's think about it:

1. We should only use characters that are actually allowed in URLs.
2. We should *probably* only pick characters that are relatively easy to type on a keyboard. Remember the use case we talked about where people are typing in a `ShortLink` that they're reading off a piece of paper?

So, what characters are allowed in URLs? It's okay to not know the answer off the top of your head. But you should be able to tell your interviewer that you know how to figure it out! Googling or searching on Stack Overflow is a fine answer. It's even cooler to say, "I'm sure this is defined in an RFC somewhere."

> **What's an RFC?** RFC (https://en.wikipedia.org/wiki/Request_for_Comments) stands for "request for comments." The first ones are from 1969 (back in the day of ARPANET, the precursor to the internet), and we still get new ones every year (http://www.rfc-editor.org/rfc-index.html). RFCs define lots of conventions for how internet communications work. Like how status code 404 means "not found" (https://tools.ietf.org/html/rfc2616#section-10.4.5). Hilariously, some of the latest RFCs spec a custom XML vocabulary for *writing RFCs*. Yo dog. Also there's this one on the history of calling variables "Foo." (https://www.rfc-editor.org/rfc/rfc3092.txt) Easy to get lost browsing these (http://www.rfc-editor.org/rfc-index.html)...

It turns out the answer is "only alphanumerics, the special characters "`$-_.+!*'(),`", and reserved characters used for their reserved purposes may be used unencoded within a URL" (RFC 1738 (https://www.rfc-editor.org/rfc/rfc1738.txt)). "Reserved characters" with "reserved purposes" are characters like '?', which marks the beginning of a query string, and '#', which marks the beginning of a fragment/anchor. We definitely shouldn't use any of those. If we allowed '?' in the beginning of our slug, the characters after it would be interpreted as part of the query string and not part of the slug!

So just alphanumerics and the "special characters" `$-_.+!*'(),`. Are accented alphabetical characters allowed? No, according to RFC 3986 (http://tools.ietf.org/html/rfc3986#section-2).

What about uppercase and lowercase? Domains aren't case-sensitive (http://tools.ietf.org/html/rfc3986#section-3.2.2) (so `google.com` and `Google.com` will always go to the same place), but the *path* portion of a URL *is* case-sensitive. If I query `parker.com/foo` and

`parker.com/Foo`, I'm requesting *different* documents (although, as a site owner, I *may* choose to return the same document in response to both requests). So yes, lowercase and capital versions of the same letter can be treated as different characters in our slugs.

Okay, so it seems like the set of allowed characters is `A-Z`, `a-z`, `0-9`, and `"$-_.+!*'(),"`. The apostrophe character seems a little iffy, since sometimes URLs are surrounded by single quotes in HTML documents. So let's pull that one.

In fact, in keeping with point (2) above about ease of typing, let's pull *all* the "special characters" from our list. It seems like a small loss on character count (8 characters) in exchange for a big win on readability and typeability. If we find ourselves wanting those extra characters, we can add add 'em back in.

Ah, but what if a user is specifying her *own* slug? She might want to use underscores, or dashes, or parentheses...so let's say for *user-specified* slugs, we allow `"$-_.+!*(),"` (still no apostrophe).

> While we're on the topic of making URLs easy to type, we might want to consider constraining our character set to clear up common ambiguities. For example, not allowing both uppercase letter O and number 0. Or lowercase letter l and number 1. Font choice can help reduce these ambiguities, but we don't have any control over the fonts people use to display our shortlinks. This is a worthwhile consideration, but at the moment it's adding complexity to a question we're still trying to figure out. So let's just mention it and say, "This is something we want to keep an eye on for later, but let's put it aside for now." Your interviewer understands that you can't accomodate *everything* in your initial design, but she'll appreciate you showing an ability to anticipate what problems may come up in the user experience.

Okay, so with `a-z`, `A-Z`, and `0-9`, we have 26 + 26 + 10 = 62 possible characters in our randomly-generated slugs. And for user-generated slugs, we have another 10 characters (`"$-_.+!*(),"`), for 72 total.

How many distinct slugs do we need?

About how many slugs do we need to be able to accomodate? This is a good question to ask your interviewer. She may want you to make a reasonable choice yourself. There's no one right answer; the important thing is to show some organized thinking.

Here's one way to come up with a ballpark estimate: about how many new slugs might we create on a busy day? Maybe 100 per minute? Hard to imagine *more* than that. That's $100 * 60 * 24 \approx 145$ thousand new links a day. 52.5 million a year. What's a number of years that feels like "almost forever"? I'd say 100. So that's 5.2 *trillion* slugs. That seems sufficiently large. It's pretty dependent on the accuracy of our estimate of 100 per minute. But it seems to be a pretty reasonable ceiling, and a purposefully high one. If we can accomodate that many slugs, we expect we'll be able to keep handing out random slugs effectively indefinitely.

How short can we make our slugs while still getting enough distinct possibilities?

Let's return to the formula we came up with before: with a c-character-long alphabet and slugs of length n, we get c^n possible slugs. We want ~5 trillion possible slugs. And we decided on a 62-character alphabet. So $62^n \approx 5$ trillion. We just have to solve for n.

We might know that we need to take a logarithm to solve for n. But even if we know that, this is a tricky thing to eyeball. If we're in front of a computer or phone, we can just plug it in to wolfram alpha. Turns out the answer is ≈ 7.09. So 7 characters gets us most of the way to our target number of distinct possibilities.

It's worth checking how many characters we could save by allowing "`$-_.+!*()`," as well. So $72^n = 5.2$ trillion. We get $n \approx 6.8$. Including the special characters would save us something like .3 characters on our slug length.

Is it worth it? Of course, there's no such thing as a fraction of a character. If we really *had to* accomodate *at least* 5.2 trillion random slugs, we'd have to round up, which would mean 7.09 would round up to 8-character slugs for our 62-character alphabet (not including special characters) and 6.8 would round up to 7-character slugs for our 72-character alphabet (including special characters).

But we don't *really* have to accomodate 5.2 trillion or more slugs. 5.2 trillion was just a ballpark estimate—and it was intended to be a *high ceiling* on how many slugs we expect to get. So let's stick with our first instinct to remove those special characters for readability purposes, and let's choose 7 characters for our slugs.

> At a glance, looks like bit.ly agrees with our choices! They seem to use the same alphabet as us (`A-Z`, `a-z`, and `0-9`) and use 7 characters for each randomly-generated slug.

> For some added potential brevity, and some added possible random slugs, we could also allow for random slugs with *fewer* than 7 characters. How many additional random slugs would that get us?
>
> If you're a whiz with mathematical series,you might know intuitively that the sum of these fewer-than-7-character slugs will be *far* less than the 7-character slugs. We can actually compute this to confirm.
>
> 62^6 (for 6-char slugs), *plus* 62^5 (for 5-char slugs), *plus* 62^4 (for 4-char slugs) + 62^3 + 62^2 + 62. About 57 billion random slugs. Which isn't that much in comparison to 5.2 *trillion*—it's two orders of magnitude less.
>
> Since this doesn't win us much, let's skip it.
>
> One interesting lesson here: going from 6 characters to 7 characters gave us a *two orders of magnitude* leap in our number of possible slugs. Going from 7 characters to 8 should have an even more dramatic effect. So if and when we *do* start running out of 7-character random slugs, allowing just 1 more character will *dramatically* push back the point where we run out of random slugs.

Okay, we know the characters we'll use for slugs. And we know how many characters we'll use.

Next: how do we generate a random slug?

We could just make a random choice for each character:

```
def generate_random_slug():
    alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789"
    num_chars = 7
    return ''.join([random.choice(alphabet) for char in num_chars])
```

But how do we ensure slugs are unique? Two general strategies:

1. "Re-roll" when we hit an already-used slug
2. Adjust our slug generation strategy to only ever give us un-claimed slugs.

If we're serious about our first 2 design goals (short slugs, and accomodating many different slugs), option (2) is clearly better than option (1). Why? As we have more and more slugs in our database, we'll get more and more collisions. For example, when we're 3/4 of the way through our set of possible 7-character slugs, we'd expect to have to make *four* "rolls" before arriving at a slug that isn't taken already. And it'll just keep going up from there.

So let's try to come with a strategy for option (2). How could we do it?

The answer is base conversion.

Using base conversion to generate slugs

We usually use base-10 numbers, which allow 10 possible numerals: `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, and `9`.

Binary is base-2 and has 2 possible numerals: `0` and `1`.

Our random slug alphabet has 62 possible numerals (`A-Z`, `a-z`, and `0-9`). So we can think of each of our possible "random" slugs as a unique *number*, expressed in base-62!

So let's keep track of a global `currentRandomSlugId`. When a request for a new random slug comes in, we simply convert that number to base-62 (using our custom numeral set) and return it. Oh, and we *increment* the `currentRandomSlugId`, in preparation for the next request for a random slug.

```
def generate_random_slug():
    global current_random_slug_id
    slug = base_conversion(current_random_slug_id, base_62_alphabet)
    current_random_slug_id += 1
    return slug
```

> Where should we store our `currentRandomSlugId`? We can keep it in memory on our webserver, perhaps with a regular writethrough to the database, to make it persistent even if the webserver crashes. But what if we have multiple front-end webservers?

How do we do the base conversion? This is easiest to show by example.

Take the number 125 in base 10.

It has a 1 in the 100s place, a 2 in the 10s place, and a 5 in the 1s place. In general, the places in a base-10 number are:

- 10^0
- 10^1
- 10^2
- 10^3
- *etc*

The places in a base-62 number are:

- 62^0
- 62^1
- $62^2 = 3,844$
- $62^3 = 238,328$
- *etc*

So to convert 125 to base-62, we distribute that 125 across these base-62 "places." The highest "place" that can take some is 62^1, which is 62. 125/62 is 2, with a remainder of 1. So we put a 2 in the 62's place and a 1 in the 1's place. So our answer is 21.

What about a higher number—say, 7,912?

Now we have enough to put something in the 3,844's place (the 62^2's place). 7,912 / 3,844 is 2 with a remainder of 224. So we put a 2 in the 3,844's place, and we distribute that remaining 224 across the remaining places—the 62's place and the 1's place. 224 / 62 is 3 with a remainder of 38. So we put a 3 in the 62's place and a 38 in the 1's place. We have this three-digit number: 2 3 38.

Now, that "38" represents *one* numeral in our base-62 number. So we need to convert that 38 into a specific choice from our set of numerals: a-z, A-Z, and 0-9.

Let's number each of our 62 numerals, like so:

```
0: 0,
1: 1,
2: 2,
3: 3,
...
10: a,
11: b,
12: c,
...
36: A,
37: B,
38: C,
...
62: Z
```

As you can see, our "38th" numeral is "C." So we convert that 38 to a "C." That gives us 23C.

> Can we convert from slugs back to numbers? Yep, easy. Take 23C, for example. Translate the numerals back to their id numbers, so we get 2 3 38. That 2 is in the 3844's place, so we take 2 * 3844. That 3 is in the 62's place, so we take 3 * 62. That 38 is in the 1's place, so we take 38 * 1. We add up all those results to get our original 7,912.

One potential issue: the `currentRandomSlugId` could give us something that a user has already claimed as a user-generated slug. We'll need to check for that, and if it happens we'll just increment the `currentRandomSlugId` and try again (and again, potentially, until we hit a "random" slug that hasn't been used yet).

```
def generate_random_slug():
    global current_random_slug_id
    while True:
        slug = base_conversion(current_random_slug_id, base_62_alphabet)
        current_random_slug_id += 1

        # make sure that slug isn't already used
        existing = db.get(slug)
        if not existing:
            return slug
```

Okay, this'll work! What's next? Let's look back at our design goals!

1. We should be able to store a *lot* of links.
2. Our shortlinks should be as short as possible.
3. Following a shortlink should be *fast*.
4. The shortlink follower should be resilient to load spikes.

We're all set on (1) and (2)! Let's start tackling (3) and (4). How do we scale our link follower to be fast and resilient to load spikes?

> Beware of premature optimization! That always looks bad. Don't just jump around random ideas for optimizations. Instead, focus on asking yourself *which thing is likely to bottleneck first* and optimizing around that.

The database read to get the destination for the given slug is certainly going to be our first bottleneck. In general, database operations usually bottleneck before business logic.

To figure out how to get these reads nice and fast, we should get specific about *how* we're storing our shortlinks. To start, what kind of database should we use?

Database choice is a very broad issue. And it's a contentious one. There are lots of different opinions about how to approach this. Here's how we'll do it:

Broadly (this is definitely a simplification), there are two main types of databases these days:

1. Relational databases (RDBMs) like MySQL and Postgres.
2. "NoSQL"-style databases like BigTable and Cassandra.

In general (again, this is a simplification), relational databases are great for systems where you expect to make lots of complex queries involving joins and such—in other words, they're good if you're planning to look at the *relationships* between things a lot. NoSQL databases don't handle these things quite as well, but in exchange they're faster for writes and simple key-value reads.

Looking at our app, it seems like relational queries aren't likely to be a big part of our app's functionality, even if we added a few of the obvious next features we might want. So let's go with NoSQL for this.

Which NoSQL database do we use? Lots of options, each with their own pros and cons. Let's keep our discussion and pseudocode generic for now.

> We might consider adding an abstraction layer between our application and the database, so that we can change over to a new one if our needs change or if some new hotness comes out.

Okay, so we have our data in a NoSQL-type database. How do we un-bottleneck database reads?

The first step is to make sure we're indexing the right way. In a NoSQL context, that means carefully designing our keys. In this case, the obvious choice is right: making the key for each row in the `ShortLink` table be *the slug*.

> If we used a SQL-type database like MySQL or Postgres, we usually default to having our key field be a standard auto-incrementing integer called "id" or "index." But in this case, because we know that slugs will be unique, there's no need for an integer id—the slug is enough of a unique identifier.
>
> BUT here's where it gets clever: what if we *represented the slug* as an auto-incrementing integer field? We'd just have to use our base conversion function to convert them to slugs! This would also give us tracking of our global `currentRandomSlugId` for free—MySQL would keep track of the highest current id in the table when it auto increments. Careful though: user-generated slugs throw a pretty huge monkey wrench into things with this strategy! How can you maintain uniqueness across user-generated and randomly-generated slugs without breaking the auto-incrementing ids for randomly-generated slugs?

How else can we speed up database reads?

We could put as much of the data *in memory* as possible, to avoid disc seeks.

This becomes especially important when we start getting a heavy load of requests to a single link, like if one of our links is on the front page of Reddit. If we have the redirect URL right there in memory, we can process those redirects quickly.

Depending on the database we use, it might already have an in-memory cache system. To get more links in memory, we may be able to configure our database to use more space for its cache.

If reads are still slow, we could research adding a caching layer, like memcached. Importantly, this *might* not save us time on reads, if the cache on the database is already pretty robust. It adds complexity—we now have two sources of truth, and we need to be careful to keep them in sync.

For example, if we let users edit their links, we need to push those edits to both the database and the cache. It could also *slow down* reads if we have lots of cache misses.

If we *did* add a caching layer, there are a few things we could talk about:

1. The eviction strategy. If the cache is full, what do we remove to make space? The most common answer is an LRU ("least recently used") strategy.
2. Sharding strategy. Sharding our cache lets us store more stuff in memory, because we can use more machines. But how do we decide which things go on which shard? The common answer is a "hash and mod strategy"—hash the key, mod the result by the number of shards, and you get a shard number to send your request to. But then how do you add or remove a shard without causing an unmanageable spike in cache misses?

Of course, we could shard our underlying database instead of, or in addition to caching. If the database has a built-in in-memory cache, sharding the data would allow us to keep more of our data in working memory without an additional caching layer! Database sharding has some of the same challenges as cache sharding. Adding and removing shards can be painful, as can migrating the schema without site downtime. That said, some NoSQL databases have great sharding systems built right in, like Cassandra.

This should get our database reads nice and fast.

The next bottleneck might be processing the actual web requests. To remedy this, we should set up multiple webserver workers. We can put them all behind a load balancer that distributes incoming requests across the workers. Having multiple web servers adds some complexity to our database (and caching layers) that we'll need to consider. They'll need to handle more simultaneous connections, for example. Most databases are pretty good at this by default.

Okay, now our redirects should go pretty quick, and should be resilient to load spikes. We have a solid system that fits all of our design goals!

1. We can store a *lot* of links.
2. Our shortlinks are as short as possible.
3. Following a shortlink is *fast*.
4. The shortlink follower is resilient to load spikes.

Bonus

As with all system design questions, there are a bunch more directions to go into with this one. A few ideas:

1. At some point we'd probably want to consider splitting our link *creation* endpoint across multiple workers as well. This adds some complexity: how do they stay in sync about what the `currentRandomSlugId` is?
2. Uptime and "single point of failure" (SPOF) are common concerns in system design. Are there any SPOFs in our current architecture? How can we ensure that an individual machine failure won't bring down our whole system?
3. Analytics. What if we wanted to show users some analytics about the links they've created? What anlytics could we show, and how would we store and display them?
4. Editing and deleting. How would we add edit and delete features?
5. Optimizing for implementation time. We built something optimized for scale. How would our system design be different if we were just trying to get an MVP off the ground as quickly as possible?