Explore - LeetCode

URL shortening systemReport Issue

Problem statement

Design a system for shortening a Uniform Resource Locator (URL). A user provides a long URL and the system returns a short URL. Upon clicking on the short url, users are redirected to the long URL.

> Start thinking about the problem. But do not go too far. First, and you already know this, we need to define functional requirements. The interviewer is eagerly waiting for us to start asking questions. What would you ask?

▼ I'm ready to learn more

Functional requirements

Typically, the problem statement already contains the key functional requirements. For example, from the problem statement above, we can obtain two functional requirements.

> What are they?

▼ I'm ready to learn more

- Given a long URL, the system returns a short URL.
- Given a short URL, the system redirects users to the original long URL.

Next, we should discuss with the interviewer what additional requirements the system has.

> Thoughts?

▼ I'm ready to learn more

- The short URL should be as short as possible.
- The short URL must be unique.
- The short URL should be non-predictable (random).
- The short URL should contain only letters (a-z, A-Z,) and digits (0-9).

Non-functional requirements

Out of non-functional requirements we discussed in the course, which ones do you consider important for the URL shortening system? I am sure you have come up with the list similar to the below

> Take time to think.

▼ I'm ready to learn more

- High availability (to minimize redirection failures)

- High scalability (to support lots of redirection requests)
- Low latency (to redirect quickly)
- High durability (to ensure that redirection mappings are not lost)

Key actors

Let’s define key actors in the system. We can usually obtain this information from the problem statement. And if not, you already know what to do - use the working backwards approach - start a conversation with the interviewer about who and how is going to use the system.

> Hint: there are two key actors in the system. Can you guess who they are?

▼ I'm ready to learn more
There are two key actors in the system: users that create short URLs, let’s call them URL writers, and users that read short URLs, let’s call them URL readers.

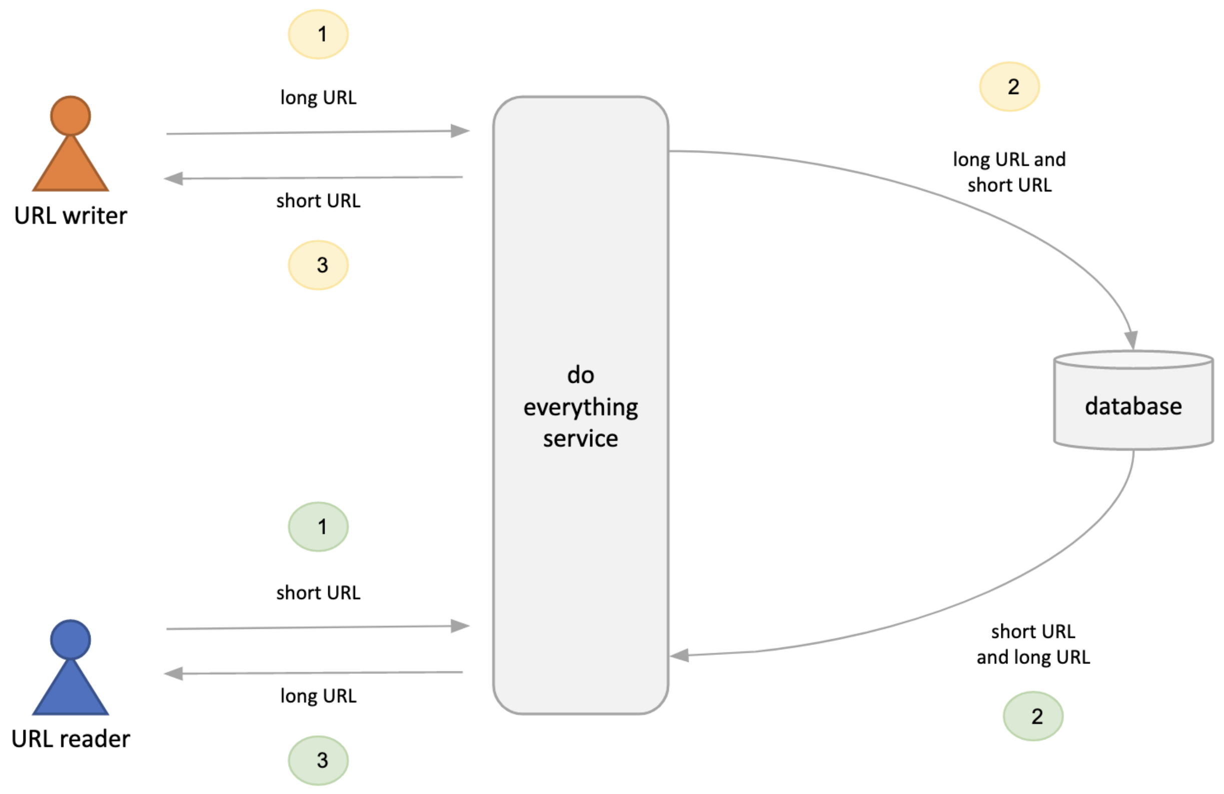
After we identified key actors, it is important to note which actors predominate in the system. Since it will help to better understand scalability needs: should we scale writes in the system or should we scale reads in the system or maybe both.

First draft of key components

We came to the point where we can try to outline the first draft of the high-level architecture. Something very simple. To spark the discussion with the interviewer and get more ideas for ourselves.

> Hint: try to separate write and read paths in the system, draw them separately on the diagram. What do you have?

▼ I'm ready to learn more
I am pretty sure you will come up with some variation of the following



"Do everything" service handles all incoming requests from users. And both short and long URLs are persistently stored in the database. That was not that hard, right? But what is next?

Let's generate questions

Next is to start asking yourself questions. And write them down on the whiteboard. No need to go too deep into details, no need to come up with the exhaustive list, just try to capture some more evident questions. The interviewer will help you here as well. Here is what I came up with.

> Think of your list and let's compare them.

▼ I'm ready to learn more

For the write path - How do we generate short URLs? - Should we have a separate component (service) that generates short URLs? - How to ensure uniqueness of short URLs? - SQL or NoSQL database to store the mapping between long and short URLs?

For the read path - How to retrieve long URLs quickly? - Should we have a separate component for handling read requests? - How to scale read requests? - How to properly handle large spikes or read requests (popular URLs)?

General questions (mostly related to non-functional requirements that are not covered in above questions) - How to achieve high availability? - How to ensure high durability?

Questions help us break down the problem into smaller and more manageable subproblems. The interviewer will also contribute her questions to this list. Most likely, there will not be enough time to discuss all the questions. Talk to the interviewer what specific areas/questions she is most interested in.

Let’s generate ideas

With questions, we have demonstrated our analytical thinking. Now it is time to demonstrate our knowledge. By generating ideas/answers for the questions above. Ideas that are backed by system design concepts we know.

> Give yourself enough time to think over every question below. Your answers do not need to be complete. And most likely they won't. You just need to generate ideas. The interviewer might want to dig deeper into details. And that is okay as well. This is a great opportunity to collaborate.

How do we generate short URLs?

▼ answer
We need a unique ID generator. Probably the first thing that comes to mind is UUID (universally unique identifier, e.g. f6cba8a5-1f65-4b62-8850-86732a68f4c7), a 128-bit random number. Is it a good option? I am not sure yet, there are at least two problems that need further analysis: UUID are not unique (although probability of collisions is very low) and 128-bit number "feels" like too long.

To ensure uniqueness, we should rely on a database. For every newly generated UUID, we check if it already exists in the database, and we generate another UUID. With a low probability of collisions, this process will complete in just a few iterations.

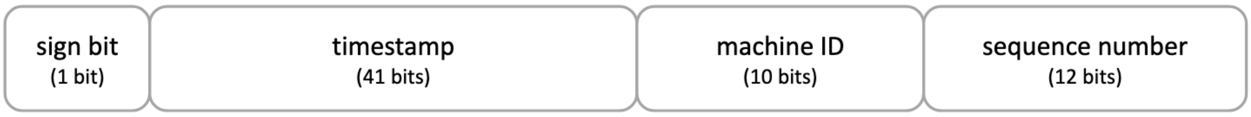
As for the length of UUID numbers, let’s try to understand if there is an optimal length of short URLs and how far UUID numbers are from this optimal value. We have 62 possible characters for short URLs: 26 (a–z) letters + 26 (A-Z) letters + 10 (0-9) digits. With 62 possible characters, we can generate 62 short URLs with length 1, 622 short URLs of length 2, and so on. This number grows fast, and there are more than 218 trillions of possible 8-character-long short URLs (628). Hence, 8 characters is more than enough for a short URL. And UUID has a length of 32 (hexadecimal digits). This is indeed "too long" if we take into account a functional requirement of creating short URLs as short as possible.

The interviewer will definitely challenge us at this moment by pointing out that UUID has the length of 32 in base-16 encoding, which contains only 16 characters (0-9, a,b, c, d, e, f). But we have 62 possible characters at our disposal, and it is more appropriate to talk about base-62 encoding. How long will the UUID be in base-62 encoding? Let’s see.

One character in base-62 encoding consumes 6 bits (5 bits is not enough to represent all 62 numbers (25=32), and 6 bits is enough (26=64). UUID consists of 128 bits, which means UUID in base-62 will have the length of 128 / 6 ~ 22 characters.

> Note: Real URL shortening systems often use base-58 encoding, since characters such as O (uppercase O), 0 (zero), and I (capital I), l (lowercase L) are hard to distinguish and thus avoided.

22-character short URLs are still much longer than the optimal length of 7-8 characters. Let’s see if we can reduce the length further. UUID has 128 bits. What if we take a random number that is much shorter, e.g. 64 bit? This will bring us to at most 11-character URLs in base-62 encoding. There are many ways to generate a random 64-bit integer. One popular solution in distributed systems is the idea inspired by Twitter’s system for generating unique ID numbers at high scale called Snowflake. It has the following structure:

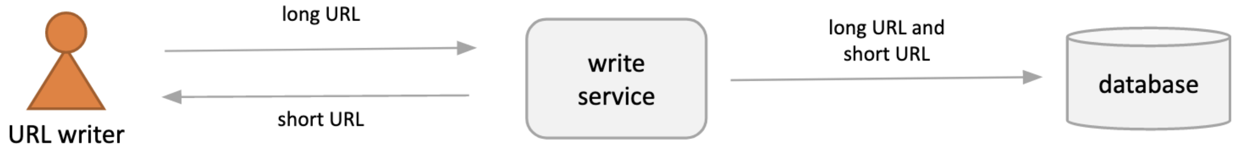


One more option for generating a smaller random number is to take some hash function (e.g. MD5, 128-bit long) and truncate it (use only the first 48 bits). Caveat: collisions are possible.

Should we have a separate component (service) that generates short URLs?

▼ answer

Yes, we should introduce a separate web service for generating short URLs. Let’s call it a Write service. Having a separate component will allow us scale write requests independently of read requests and we can change internal implementation more easily in the future.



How to ensure uniqueness of short URLs?

▼ answer

Database will help. We first check if the generated short URL is already present in the database, and return a new URL to the client only when the uniqueness constraint is honored. To check URL presence in the database quickly and cheaply, we should utilize a Bloom filter, a memory-efficient probabilistic data structure to quickly (O(1)) check whether an element is present in a set.

SQL or NoSQL database to store the mapping between long and short URLs?

▼ answer

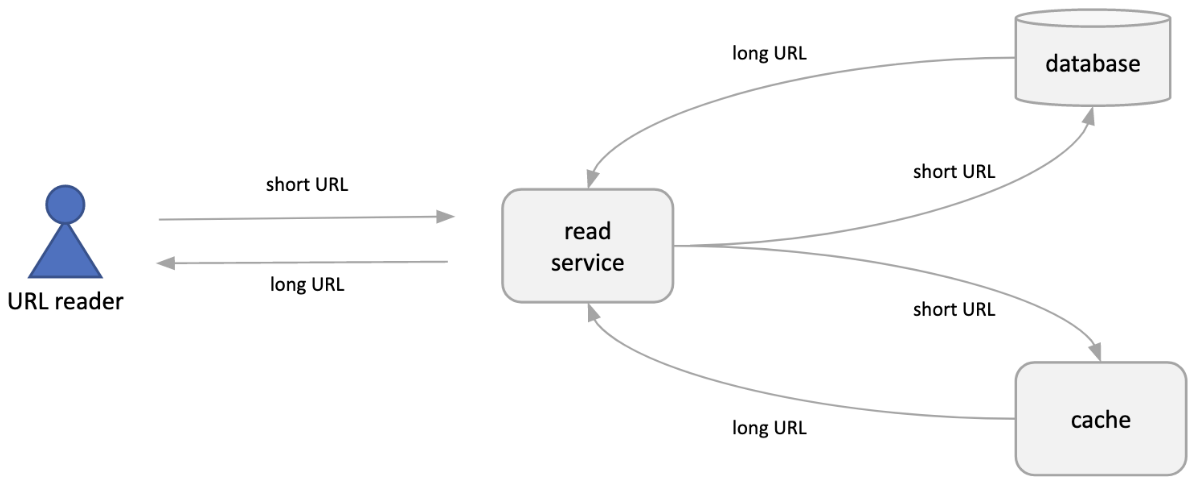
The URL shortening system is read-heavy. Any database that scales well for reads will work. And as we already know, both SQL databases (e.g. using solutions like Vitess) and NoSQL databases can scale reads very well. NoSQL database (e.g. MongoDB) will be a better choice. Why? Let's discuss it in the second module of the course.

How to retrieve long URLs quickly?

▼ answer
Caching, right? The cache will store the mapping between short URLs and long URLs. We can use either cache-aside and read-through pattern for updating the cache. With read-through being a better option. As for the eviction policy, the Least Recently Used (LRU) eviction policy is a good choice.

Should we have a separate component for handling read requests?

▼ answer
Yes, we should introduce a separate web service for handling read requests. Let's call it a Read service. Having a separate component will allow us scale read requests independently of write requests and we can make this service act as a read-through cache.



How to scale read requests?

▼ answer
We are going to use horizontally scalable distributed cache solutions (Memcached, Redis) and add more read replicas to the database when needed. As mentioned earlier, we can also utilize Bloom filters to reduce the load on both the cache and the database.

How to properly handle large spikes or read requests (popular URLs)?

▼ answer
We should utilize the local cache on Read service machines to store the same hot URL on multiple machines. This way, read requests can be handled by any of the Read service machines in the cluster, avoiding heavy load on one particular machine.

How to achieve high availability?

▼ answer

As we already know, achieving high availability requires both architecture and processes. In our system, each individual component should be highly available. Which means we have redundancy for every component in the system, heavily rely on load balancing, protect our components from atypical behaviors of clients and downstream dependencies, quickly detect and resolve failures.

How to ensure high durability?

▼ answer

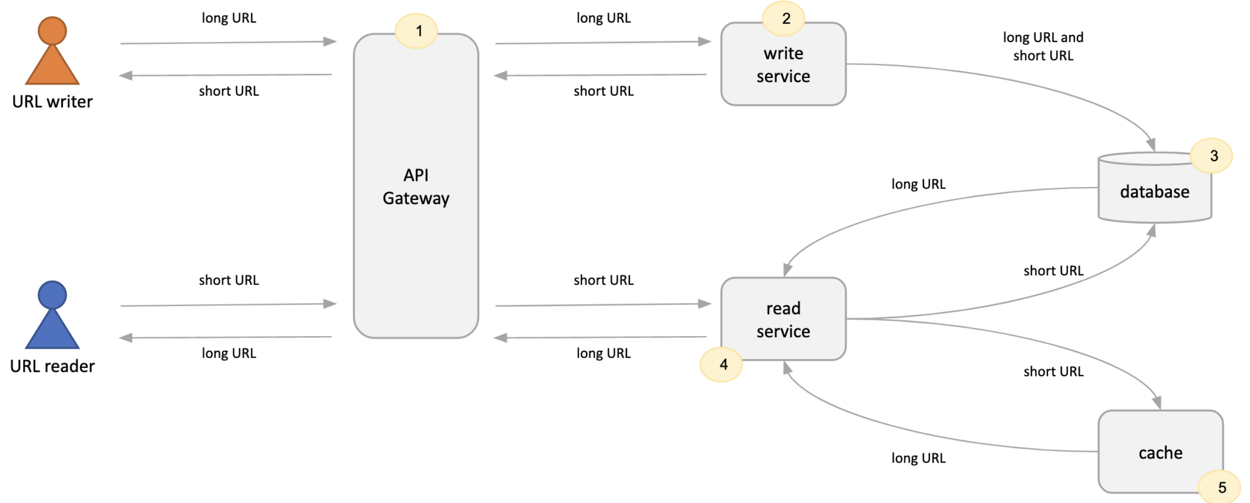
By copying data at multiple different levels. Specifically, we rely on database data replication and backups.

Reference architecture

Let's put together all the components we discussed above and briefly summarize their roles in the overall architecture.

> What did you get?

▼ I'm ready to learn more



Final notes

- In this example, we assumed a single round of questions and ideas generation. In a real interview, we should expect several rounds of questions and answers.
- Some important aspects such as API design, database schema design and database considerations, monitoring and security are omitted from this example. We are going to return to this problem in the subsequent modules of the course to discuss these additional aspects in more detail.