**Design a URL Shortener like Bit.ly**

[[](https://www.linkedin.com/in/evan-king-40072280/)](https://www.linkedin.com/in/evan-king-40072280/" \t "_blank)

**[Evan King](https://www.linkedin.com/in/evan-king-40072280/" \t "_blank)**

[Ex-Meta Staff Engineer](https://www.linkedin.com/in/evan-king-40072280/" \t "_blank)

easy

35 min

**Understanding the Problem**

**🔗 What is** [**Bit.ly**](https://bitly.com/)**?** Bit.ly is a URL shortening service that converts long URLs into shorter, manageable links. It also provides analytics for the shortened URLs.

Designing a URL shortener is a very common beginner system design interview question. Whereas in many of the other breakdowns on Hello Interview we focus on depth, for this one, I'm going to target a more junior audience. If you're new to system design, this is a great question to start with! I'll try my best to slow down and teach concepts that are otherwise taken for granted in other breakdowns.

[**Functional Requirements**](https://www.hellointerview.com/learn/system-design/in-a-hurry/delivery#1-functional-requirements)

The first thing you'll want to do when starting a system design interview is to get a clear understanding of the requirements of the system. Functional requirements are the features that the system must have to satisfy the needs of the user.

In some interviews, the interviewer will provide you with the core functional requirements upfront. In other cases, you'll need to determine these requirements yourself. If you're familiar with the product, this task should be relatively straightforward. However, if you're not, it's advisable to ask your interviewer some clarifying questions to gain a better understanding of the system.

The most important thing is that you zero in on the top 3-4 features of the system and don't get distracted by the bells and whistles.

We'll concentrate on the following set of functional requirements:

**Core Requirements**

1. Users should be able to submit a long URL and receive a shortened version.
   * Optionally, users should be able to specify a custom alias for their shortened URL.
   * Optionally, users should be able to specify an expiration date for their shortened URL.
2. Users should be able to access the original URL by using the shortened URL.

**Below the line (out of scope):**

* User authentication and account management.
* Analytics on link clicks (e.g., click counts, geographic data).

These features are considered "below the line" because they add complexity to the system without being core to the basic functionality of a URL shortener. In a real interview, you might discuss these with your interviewer to determine if they should be included in your design.

[**Non-Functional Requirements**](https://www.hellointerview.com/learn/system-design/in-a-hurry/delivery#2-non-functional-requirements)

Next up, you'll want to outline the core non-functional requirements of the system. Non-functional requirements refer to specifications about how a system operates, rather than what tasks it performs. These requirements are critical as they define system attributes like scalability, latency, security, and availability, and are often framed as specific benchmarks—such as a system's ability to handle 100 million daily active users or respond to queries within 200 milliseconds.

**Core Requirements**

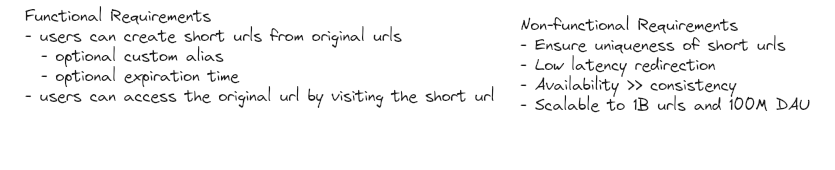
1. The system should ensure uniqueness for the short codes (no two long URLs can map to the same short URL)
2. The redirection should occur with minimal delay (< 100ms)
3. The system should be reliable and available 99.99% of the time (availability > consistency)
4. The system should scale to support 1B shortened URLs and 100M DAU

**Below the line (out of scope):**

* Data consistency in real-time analytics.
* Advanced security features like spam detection and malicious URL filtering.

An important consideration in this system is the significant imbalance between read and write operations. The read-to-write ratio is heavily skewed towards reads, as users frequently access shortened URLs, while the creation of new short URLs is comparatively rare. For instance, we might see 1000 clicks (reads) for every 1 new short URL created (write). This asymmetry will significantly impact our system design, particularly in areas such as caching strategies, database choice, and overall architecture.

Here is what you might write on the whiteboard:



Bit.ly Non-Functional Requirements

**The Set Up**

[**Defining the Core Entities**](https://www.hellointerview.com/learn/system-design/in-a-hurry/delivery#3-defining-the-core-entities)

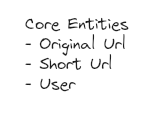
We recommend that you start with a broad overview of the primary entities. At this stage, it is not necessary to know every specific column or detail. We will focus on the intricacies, such as columns and fields, later when we have a clearer grasp. Initially, establishing these key entities will guide our thought process and lay a solid foundation as we progress towards defining the API.

Just make sure that you let your interviewer know your plan so you're on the same page. I'll often explain that I'm going to start with just a simple list, but as we get to the high-level design, I'll document the data model more thoroughly.

In a URL shortener, the core entities are very straightforward:

1. **Original URL**: The original long URL that the user wants to shorten.
2. **Short URL**: The shortened URL that the user receives and can share.
3. **User**: Represents the user who created the shortened URL.

In the actual interview, this can be as simple as a short list like this. Just make sure you talk through the entities with your interviewer to ensure you are on the same page.



Bit.ly Entities

[**The API**](https://www.hellointerview.com/learn/system-design/in-a-hurry/delivery#4-api-or-system-interface)

The next step in the delivery framework is to define the APIs of the system. This sets up a contract between the client and the server, and it’s the first point of reference for the high-level design.

Your goal is to simply go one-by-one through the core requirements and define the APIs that are necessary to satisfy them. Usually, these map 1:1 to the functional requirements, but there are times when multiple endpoints are needed to satisfy an individual functional requirement.

9/10 you'll use a REST API and focus on choosing the right HTTP method or verb to use.

* **POST**: Create a new resource
* **GET**: Read an existing resource
* **PUT**: Update an existing resource
* **DELETE**: Delete an existing resource

To shorten a URL, we’ll need a POST endpoint that takes in the long URL and optionally a custom alias and expiration date, and returns the shortened URL. We use post here because we are creating a new entry in our database mapping the long url to the newly created short url.

// Shorten a URL

POST /urls

{

"long\_url": "https://www.example.com/some/very/long/url",

"custom\_alias": "optional\_custom\_alias",

"expiration\_date": "optional\_expiration\_date"

}

->

{

"short\_url": "http://short.ly/abc123"

}

For redirection, we’ll need a GET endpoint that takes in the short code and redirects the user to the original long URL. GET is the right verb here because we are reading the existing long url from our database based on the short code.

// Redirect to Original URL

GET /{short\_code}

-> HTTP 302 Redirect to the original long URL

We'll talk more about which HTTP status codes to use during the high-level design.

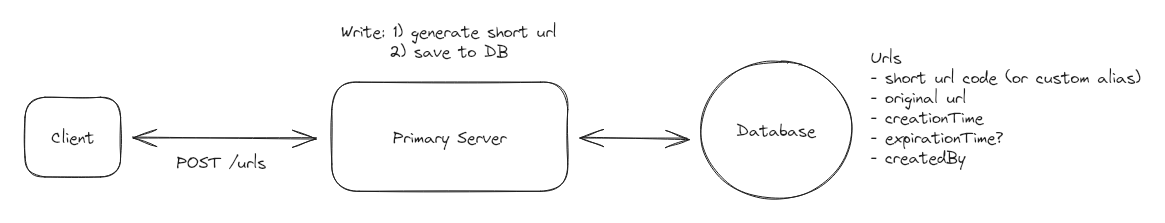
[**High-Level Design**](https://www.hellointerview.com/learn/system-design/in-a-hurry/delivery#high-level-design-10-15-minutes)

We'll start our design by going one-by-one through our functional requirements and designing a single system to satisfy them. Once we have this in place, we'll layer on depth via our deep dives.

1) Users should be able to submit a long URL and receive a shortened version

The first thing we need to consider when designing this system is how we're going to generate a short url. Users are going to come to us with long urls and expect us to shrink them down to a manageable size.

We'll outline the core components necessary to make this happen at a high-level.



Create a short url

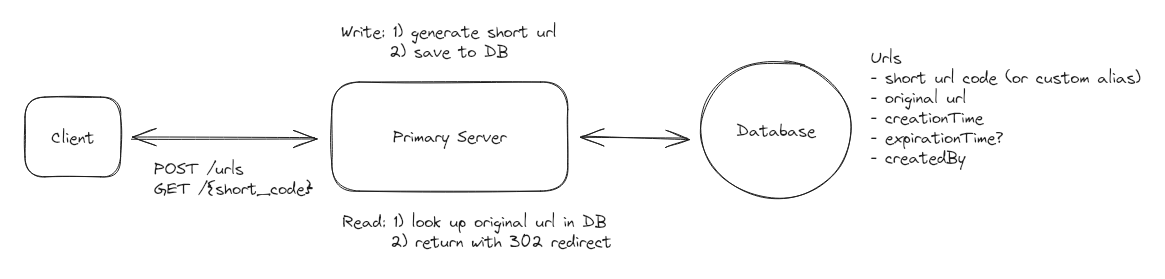
1. **Client**: Users interact with the system through a web or mobile application.
2. **Primary Server**: The primary server receives requests from the client and handles all business logic like short url creation and validation.
3. **Database**: Stores the mapping of short codes to long urls, as well as user-generated aliases and expiration dates.

When a user submits a long url, the client sends a POST request to /urls with the long url, custom alias, and expiration date. Then:

1. The Primary Server receives the request and validates the long URL. This is to ensure that the URL is valid (there's no point in shortening an invalid URL) and that it doesn't already exist in our system (we don't want collisions).
   * To validate that the URL is valid, we can use popular open-source libraries like [**is-url**](https://www.npmjs.com/package/is-url) or write our own simple validation.
   * To check if the URL already exists in our system, we can query our database to see if the long URL is already present.
2. If the URL is valid and doesn't already exist, we can proceed to generate a short URL unless
   * For now, we'll abstract this away as some magic function that takes in the long URL and returns a short URL. We'll dive deep into how to generate short URLs in the next section.
   * If the user has specified a custom alias, we can use that as the short code (after validating that it doesn't already exist).
3. Once we have the short URL, we can proceed to insert it into our database, storing the short code (or custom alias), long URL, and expiration date.
4. Finally, we can return the short URL to the client.

2) Users should be able to access the original URL by using the shortened URL

Now our short URL is live and users can access the original URL by using the shortened URL. Importantly, this shortened URL exists at a domain that we own! For example, if our site is located at short.ly, then our short urls look like short.ly/abc123 and all requests to that short url go to our Primary Server.



Redirect to original url

When a user accesses a shortened URL, the following process occurs:

1. The user's browser sends a GET request to our server with the short code (e.g., GET /abc123).
2. Our Primary Server receives this request and looks up the short code (abc123) in the database.
3. If the short code is found and hasn't expired (by comparing the current date to the expiration date in the database), the server retrieves the corresponding long URL.
4. The server then sends an HTTP redirect response to the user's browser, instructing it to navigate to the original long URL.

There are two main types of HTTP redirects that we could use for this purpose:

1. **301 (Permanent Redirect)**: This indicates that the resource has been permanently moved to the target URL. Browsers typically cache this response, meaning subsequent requests for the same short URL might go directly to the long URL, bypassing our server.

The response back to the client looks like this:

HTTP/1.1 301 Moved Permanently

Location: https://www.original-long-url.com

1. **302 (Temporary Redirect)**: This suggests that the resource is temporarily located at a different URL. Browsers do not cache this response, ensuring that future requests for the short URL will always go through our server first.

The response back to the client looks like this:

HTTP/1.1 302 Found

Location: https://www.original-long-url.com

In either case, the user's browser (the client) will automatically follow the redirect to the original long URL and users will never even know that a redirect happened.

For a URL shortener, a 302 redirect is often preferred because:

* It gives us more control over the redirection process, allowing us to update or expire links as needed.
* It prevents browsers from caching the redirect, which could cause issues if we need to change or delete the short URL in the future.
* It allows us to track click statistics for each short URL (even though this is out of scope for this design).

[**Potential Deep Dives**](https://www.hellointerview.com/learn/system-design/in-a-hurry/delivery#deep-dives-10-minutes)

At this point, we have a basic, functioning system that satisfies the functional requirements. However, there are a number of areas we could dive deeper into to reduce the likelihood of collision, support scalability, and improve performance. We can now look back at our non-functional requirements and see which ones still need to be satisfied or improved upon.

1) How can we ensure short urls are unique?

In our high-level design, we abstracted away the details of how we generate a short url but now it's time to get into the nitty-gritty! There are a handful of constraints we need to keep in mind as we design:

1. We need to ensure that the short codes are unique.
2. We want the short codes to be as short as possible (it is a url shortener afterall).
3. We want to ensure codes are efficiently generated.

Let's weigh a few options and consider their pros and cons.

**Bad Solution: Long Url Prefix**

**Approach**

The silliest thing we could do to shorten an input url is to just take the prefix of the input url as the short code. Image you had a url like www.linkedin.com/in/evan-king-40072280/ we could just take the first N (lets say 8 for now) characters of the url and use that as the short code. In this case www.short.ly/www.link.

**Challenges**

Clearly, this method would not meet constraint #1 about uniqueness. Any two urls that share the first N characters would end up mapping to the exact same short url. When a user comes and asks to be redirected via short url www.short.ly/www.link we would not know whether they want to visit www.linkedin.com/in/evan-king-40072280/, www.linkedin.com/in/stefanmai/, or any of the countless other urls that share the same prefix.

**Good Solution: Random Number Generator or Hash Function**

**Approach**

We need some entropy (randomness) to try to ensure that our codes are unique. We could try a random number generator or a hash function!

Using a random number generator to create short codes involves generating a random number each time a new URL is shortened. This random number serves as the unique identifier for the URL. We can use common random number generation functions like JavaScript's Math.random() or more robust cryptographic random number generators for increased unpredictability. The generated random number would then be used as the short code for the URL.

Alternatively, we could use a hash function to generate a fixed-size hash code. Hash functions take an input and return a deterministic, fixed-size string of characters. This is great because if two people try to generate a short code for the same long URL, they will get the same short code without needing to query the database. Hash functions also provide a high degree of entropy, meaning that the output is random and unique for each input and unlikely to collide.

With either the random number generator or the hash function, we can then take the output and encode it using a base62 encoding scheme and then take just the first N characters as our short code. N is determined based on the number of characters needed to minimize collisions.

Why base62? It's a compact representation of numbers that uses 62 characters (a-z, A-Z, 0-9). The reason it's 62 and not the more common base64 is because we exclude the characters + and / single those are reserved for url encoding.

Let's view a quick example of this in some pseudo code.

input\_url = "https://www.example.com/some/very/long/url"

random\_number = Math.random()

short\_code\_encoded = base62\_encode(random\_number)

short\_code = short\_code\_encoded[:8] *# 8 characters*

A diagram of a computer server

Description automatically generated

**Challenges**

Despite the randomness, there's a non-negligible chance of generating duplicate short codes, especially as the number of stored URLs increases. This is known as the [**Birthday Problem**](https://en.wikipedia.org/wiki/Birthday_problem), where collisions become more probable than intuition suggests. To reduce collision probability, we need high entropy, which means generating longer short codes. However, longer codes negate the benefit of having a short URL. Detecting and resolving collisions requires additional database checks for each new code, adding latency and complexity to the system. The necessity to check for existing codes on each insertion can become a bottleneck as the system scales. Thus, introducing a tradeoff between uniqueness, length, and efficiency -- making it difficult for us to achieve all three.

**Great Solution: Unique Counter with Base62 Encoding**

**Approach**

One way to guarantee we don't have collisions is to simply increment a counter for each new url. We can then take the output of the counter and encode it using base62 encoding to ensure it's a compacted representation.

Redis is particularly well-suited for managing this counter because it's single-threaded and supports atomic operations. Being single-threaded means Redis processes one command at a time, eliminating race conditions. Its INCR command is atomic, meaning the increment operation is guaranteed to execute completely without interference from other operations. This is crucial for our counter - we need absolute certainty that each URL gets a unique number, with no duplicates or gaps.

Each counter value is unique, eliminating the risk of collisions without the need for additional checks. Incrementing a counter and encoding it is computationally efficient, supporting high throughput. With proper counter management, the system can scale horizontally to handle massive numbers of URLs. The short code can be easily decoded back to the original ID if needed, aiding in database lookups.

A diagram of a computer server

Description automatically generated

**Challenges**

In a distributed environment, maintaining a single global counter can be challenging due to synchronization issues. All instances of our Primary Server would need to agree on the counter value. **We'll talk more about this when we get into scaling.** We also have to consider that the size of the short code continues to increase over time with this method.

To determine whether we should be concerned about length, we can do a little math. If we have 1B urls, when base62 encoded, this would result in a 6-character string. Here's why:

[**1,000,000,000 in base62 is '15ftgG'**](https://math.tools/calculator/base/10-62)

This means that even with a billion URLs, our short codes would still be quite compact. As we approach 62^7 (over 3.5 trillion) URLs, we'd need to move to 7-character codes. This scalability allows us to handle a massive number of URLs while keeping the codes short and effectively neutralizing that concern.

2) How can we ensure that redirects are fast?

When dealing with a large database of shortened URLs, finding the right match quickly becomes crucial for a smooth user experience. Without any optimization, our system would need to check every single pair of short and original URLs in the database to find the one we're looking for. This process, known as a "full table scan," can be incredibly slow, especially as the number of URLs grows into the millions or billions.

**Good Solution: Add an Index**

**Approach**

To avoid a full table scan, we can use a technique called indexing. Think of an index like a book's table of contents or a library's card catalog. It provides a quick way to find what we're looking for without having to flip through every page or check every shelf. In database terms, an index creates a separate, sorted list of our short URLs, each with a pointer to where the full information is stored in the main table. This allows the database to use efficient search methods, dramatically reducing the time it takes to find a matching URL.

1. B-tree Indexing: Most relational databases use B-tree indexes by default. For our URL shortener, we'd create a B-tree index on the short code column. This provides O(log n) lookup time, which is very efficient for large datasets.
2. Primary Key: We should designate the short code as the primary key of our table. This automatically creates an index and ensures uniqueness. By making the short code the primary key, we get the benefits of both indexing and data integrity, as the database will enforce uniqueness and optimize queries on this field.
3. Hash Indexing: For databases that support it (like PostgreSQL), we can use hash indexing on the short code column. This provides O(1) average case lookup time, which is faster than B-tree for exact match queries (as is our use case)

With these optimizations in place, our system can now find the matching original URL in a fraction of the time it would take without them. Instead of potentially searching through millions of rows, the database can find the exact match almost instantly, greatly improving the performance of our URL shortener service.

**Challenges**

Relying solely on a disk-based database for redirects presents some challenges, although modern SSDs have significantly reduced the performance gap. While disk I/O is slower than memory access, it's not prohibitively slow. A typical SSD can handle around 100,000 IOPS (Input/Output Operations Per Second), which is quite fast for many applications.

However, the main challenge lies in the sheer volume of read operations required. With 100M DAU (Daily Active Users), assuming each user performs an average of 5 redirects per day, we're looking at:

100,000,000 users \* 5 redirects = 500,000,000 redirects per day 500,000,000 / 86,400 seconds ≈ 5,787 redirects per second

This assumes redirects are evenly distributed throughout the day, which is unlikely. Most redirects will occur during peak hours, which means we need to design for high-traffic spikes. Multiplying by 100x to handle the spikes means we need to handle ~600k read operations per second.

Even with optimized queries and indexing, a single database instance may struggle to keep up with this volume of traffic. This high read load could lead to increased response times, potential timeouts, and might affect other database operations like URL shortening.

**Great Solution: Implementing an In-Memory Cache (e.g., Redis)**

**Approach**

To improve redirect speed, we can introduce an in-memory cache like Redis or Memcached between the application server and the database. This cache stores the frequently accessed mappings of short codes to long URLs. When a redirect request comes in, the server first checks the cache. If the short code is found in the cache (a cache hit), the server retrieves the long URL from the cache, significantly reducing latency. If not found (a cache miss), the server queries the database, retrieves the long URL, and then stores it in the cache for future requests.

The key here is that instead of going to disk we access the mapping directly from memory. This difference in access speed is significant:

* Memory access time: ~100 nanoseconds (0.0001 ms)
* SSD access time: ~0.1 milliseconds
* HDD access time: ~10 milliseconds

This means memory access is about 1,000 times faster than SSD and 100,000 times faster than HDD. In terms of operations per second:

* Memory: Can support millions of reads per second
* SSD: ~100,000 IOPS (Input/Output Operations Per Second)
* HDD: ~100-200 IOPS

A diagram of a data processing process

Description automatically generated

**Challenges**

While implementing an in-memory cache offers significant performance improvements, it does come with its own set of challenges. Cache invalidation can be complex, especially when updates or deletions occur, though this issue is minimized since URLs are mostly read-heavy and rarely change. The cache needs time to "warm up," meaning initial requests may still hit the database until the cache is populated. Memory limitations require careful decisions about cache size, eviction policies (e.g., LRU - Least Recently Used), and which entries to store. Introducing a cache adds complexity to the system architecture, and you'll want to be sure you discuss the tradeoffs and invalidation strategies with your interviewer.

**Great Solution: Leveraging Content Delivery Networks (CDNs) and Edge Computing**

3) How can we scale to support 1B shortened urls and 100M DAU?

We've done much of the hard work to scale already! We introduced a caching layer which will help with read scalability, now lets talk a bit about scaling writes.

**We'll start by looking at the size of our database.**

Each row in our database consists of a short code (~8 bytes), long URL (~100 bytes), creationTime (~8 bytes), optional custom alias (~100 bytes), and expiration date (~8 bytes). This totals to ~200 bytes per row. We can round up to 500 bytes to account for any additional metadata like the creator id, analytics id, etc.

If we store 1B mappings, we're looking at 500 bytes \* 1B rows = 500GB of data. The reality is, this is well within the capabilities of modern SSDs. Given the number of urls on the internet is our maximum bound, we can expect it to grow but only modestly. If we were to hit a hardware limit, we could always shard our data across multiple servers but a single Postgres instance, for example, should do for now.

So what database technology should we use?

The truth is: most will work here. We offloaded the heavy read throughput to a cache and write throughput is pretty low. We could estimate that maybe 100k new urls are created per day. 100k new rows per day is ~1 row per second. So any reasonable database technology should do (ie. Postgres, MySQL, DynamoDB, etc). In your interview, you can just pick whichever you have the most experience with! If you don't have any hands on experience, go with Postgres.

But what if the DB goes down?

It's a valid question, and one always worth considering in your interview. We could use a few different strategies to ensure high availability.

1. **Database Replication**: By using a database like Postgres that supports replication, we can create multiple identical copies of our database on different servers. If one server goes down, we can redirect to another. This adds complexity to our system design as we now need to ensure that our Primary Server can interact with any replica without any issues. This can be tricky to get right and adds operational overhead.
2. **Database Backup**: We could also implement a backup system that periodically takes a snapshot of our database and stores it in a separate location. This adds complexity to our system design as we now need to ensure that our Primary Server can interact with the backup without any issues. This can be tricky to get right and adds operational overhead.

Now, let's point our attention to the Primary Server.

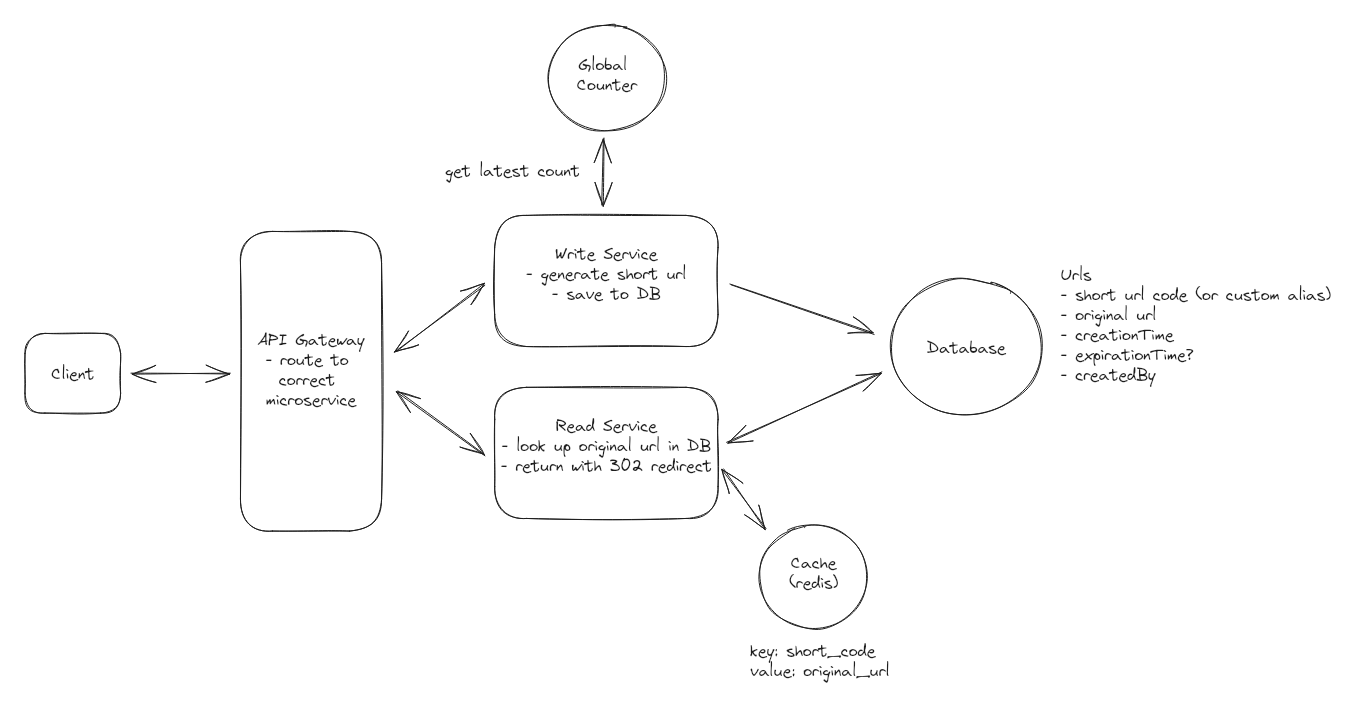
Coming back to our initial observation that reads are much more frequent than writes, we can scale our Primary Server by separating the read and write operations. This introduces a microservice architecture where the Read Service handles redirects while the Write service handles the creation of new short urls. This separation allows us to scale each service independently based on their specific demands.

Now, we can horizontally scale both the Read Service and the Write Service to handle increased load. Horizontal scaling is the process of adding more instances of a service to distribute the load across multiple servers. This can help us handle a large number of requests per second without increasing the load on a single server. When a new request comes in, it is randomly routed to one of the instances of the service.

But what about our counter?

Horizontally scaling our write service introduces a significant issue! For our short code generation to remain globally unique, we need a single source of truth for the counter. This counter needs to be accessible to all instances of the Write Service so that they can all agree on the next value.

We could solve this by using a centralized Redis instance to store the counter. This Redis instance can be used to store the counter and any other metadata that needs to be shared across all instances of the Write Service. Redis is single-threaded and is very fast for this use case. It also supports atomic increment operations which allows us to increment the counter without any issues. Now, when a user requests to shorten a url, the Write Service will get the next counter value from the Redis instance, compute the short code, and store the mapping in the database.



Final Design

But should we be concerned about the overhead of an additional network request for each new write request?

The reality is, this is probably not a big deal. Network requests are fast! In practice, the overhead of an additional network request is negligible compared to the time it takes to perform other operations in the system. That said, we could always use a technique called "counter batching" to reduce the number of network requests. Here's how it works:

1. Each Write Service instance requests a batch of counter values from the Redis instance (e.g., 1000 values at a time).
2. The Redis instance atomically increments the counter by 1000 and returns the start of the batch.
3. The Write Service instance can then use these 1000 values locally without needing to contact Redis for each new URL.
4. When the batch is exhausted, the Write Service requests a new batch.

This approach reduces the load on Redis while still maintaining uniqueness across all instances. It also improves performance by reducing network calls for counter values.

To ensure high availability of our counter service, we can use Redis's built-in replication features. Redis Enterprise, for example, provides automatic failover and cross-region replication. For additional durability, we can periodically persist the counter value to a more durable storage system.