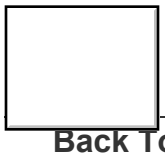



Back To Course Home

Grokking Modern System Design Interview for Engineers & Managers

0% completed

System Design Interviews

Introduction

Abstractions

Non-functional System Characteristics

Back-of-the-envelope Calculations

Building Blocks

Domain Name System

Load Balancers

Databases

Key-value Store

☐

Content Delivery Network (CDN)

☐

Sequencer

☐

Distributed Monitoring

☐

Monitor Server-side Errors

☐

Monitor Client-side Errors

☐

Distributed Cache

☐

Distributed Messaging Queue

☐

Pub-sub

☐

Rate Limiter

☐

Blob Store

☐

Distributed Search

☐

Distributed Logging

☐

Distributed Task Scheduler

Sharded Counters

Concluding the Building Blocks Discussion

Design YouTube

Design Quora

Design Google Maps

Design a Proximity Service / Yelp

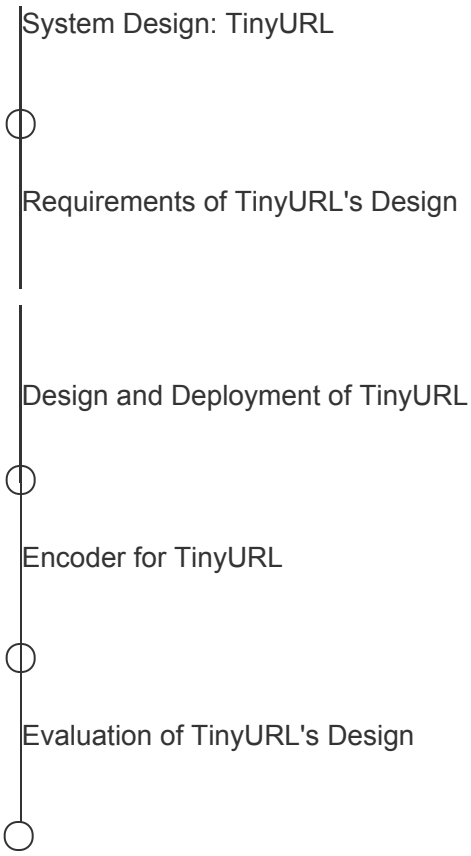
Design Uber

Design Twitter

Design Newsfeed System

Design Instagram

Design a URL Shortening Service / TinyURL



Design a Web Crawler

Design WhatsApp

Design Typeahead Suggestion

Design a Collaborative Document Editing Service / Google Docs

Spectacular Failures

Concluding Remarks

Course Certificate

Mark Course as Completed

Design and Deployment of TinyURL

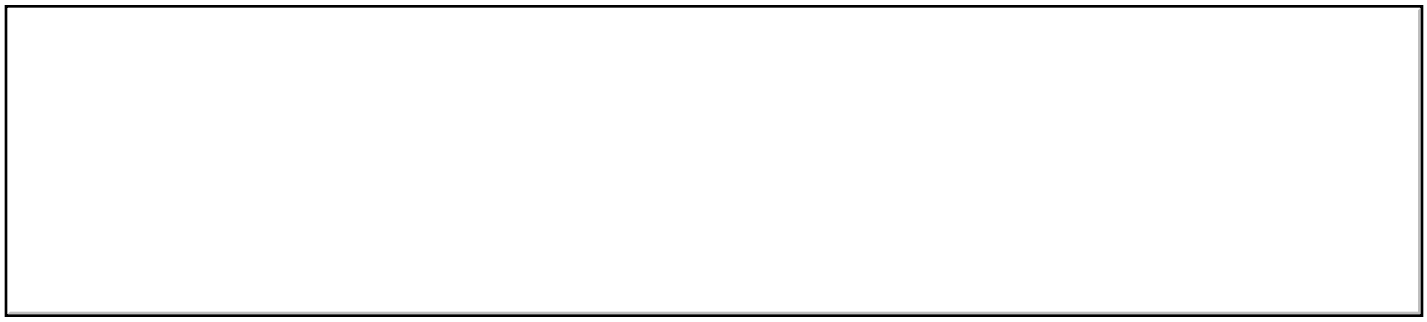
Deep-diving into the design and deployment of the URL shortening service.

We'll cover the following

System APIs#

To expose the functionality of our service, we can use REST APIs for the following features:

- Shortening a URL
- Redirecting a short URL
- Deleting a short URL



System API design overview

Shortening a URL#

We can create new short URLs with the following definition:

```
shortURL(api_dev_key, original_url, custom_alias=None, expiry_date=None
)
```

The API call above has the following parameters:

Parameter	Description
`api_dev_key`	A registered user account's unique identifier. This is useful in tracking a user's activity ar the system to control the associated services accordingly.
`original_url`	The original long URL that is needed to be shortened.
`custom_alias`	The optional key that the user defines as a customer short URL.
`expiry_date`	The optional expiration date for the shortened URL.

A successful insertion returns the user the shortened URL. Otherwise, the system returns an appropriate error code to the user.

Redirecting a short URL#

To redirect a short URL, the REST API's definition will be:

```
redirectURL(api_dev_key, url_key)
```

With the following parameters:

Parameter	Description
`api_dev_key`	The registered user account's unique identifier.
`url_key`	The shortened URL against which we need to fetch the long URL from the database.

A successful redirection lands the user to the original URL associated with the `url_key`.

Deleting a short URL#

Similarly, to delete a short URL, the REST API's definition will be:

```
deleteURL(api_dev_key, url_key)
```

and the associated parameters will be:

Parameter	Description
`api_dev_key`	The registered user account's unique identifier.
`url_key`	The shortened URL against which we need to fetch the long URL from the database.

A successful deletion returns a system message, `URL Removed`, conveying the successful URL removal from the system.

Design#

Let's discuss the main design components required for our URL shortening service. Our design depends on each part's functionality and progressively combines them to achieve different workflows mentioned in the functional requirements.

Components#

We'll explain the inner mechanism of different components within our system, as well as their usage as a part of the whole system below. We'll also highlight the design choices made for each component to achieve the overall functionality.

Database: For services like URL shortening, there isn't a lot of data to store. However, the storage has to be horizontally scalable. The type of data we need to store includes:

- User details.
- Mappings of the URLs, that is, the long URLs that are mapped onto short URLs.

Our service doesn't require user registration for the generation of a short URL, so we can skip adding certain data to our database. Additionally, the stored records will have no relationships among themselves other than linking the URL-creating user's details, so we don't need structured storage for record-keeping. Considering the reasons above and the fact that our system will be read-heavy, NoSQL is a suitable choice for storing data. In particular, MongoDB is a good choice for the following reasons:

1. It uses leader-follower protocol, making it possible to use replicas for heavy reading.
2. MongoDB ensures atomicity in concurrent write operations and avoids collisions by returning duplicate-key errors for record-duplication issues.

Quiz

Question

Why are NoSQL databases like Cassandra or Riak not good choices instead of MongoDB?

Show Answer

Short URL generator: Our short URL generator will comprise a building block and an additional component:

- A sequencer to generate unique IDs
- A Base-58 encoder to enhance the readability of the short URL

We built a sequencer in our building blocks section to generate 64-bit unique *numeric* IDs. However, our proposed design requires 64-bit *alphanumeric* short URLs in base-58. To convert the numeric (base-10) IDs to alphanumeric (base-58), we'll need a base-10 for the base-58 encoder. We'll explore the rationale behind these decisions alongside the internal working of the base-58 encoder in the next lesson.

Take a look at the diagram below to understand how the overall short URL generation unit will work.

Internal working of a short URL generator

Other building blocks: Beside the elements mentioned above, we'll also incorporate other building blocks like load balancers, cache, and rate limiters.

- **Load balancing**: We can employ Global Server Load Balancing (GSLB) apart from local load balancing to improve availability. Since we have plenty of time between a short URL being generated and subsequently accessed, we can safely assume that our DB is geographically consistent and that distributing requests globally won't cause any issues.
- **Cache**: For our specific read-intensive design problem, Memcached is the best choice for a cache solution. We require a simple, horizontally scalable cache system

with minimal data structure requirements. Moreover, we'll have a data-center-specific caching layer to handle native requests. Having a global caching layer will result in higher latency.

- **Rate limiter**: Limiting each user's quota is preferable for adding a security layer to our system. We can achieve this by uniquely identifying users through their unique `api_dev_key` and applying one of the discussed rate-limiting algorithms (see Rate Limiter from *Building Blocks*). Keeping in view the simplicity of our system and the requirements, the fixed window counter algorithm would serve the purpose, as we can assign a set number of shortening and redirection operations per `api_dev_key` for a specific timeframe.

Quiz

Question 1

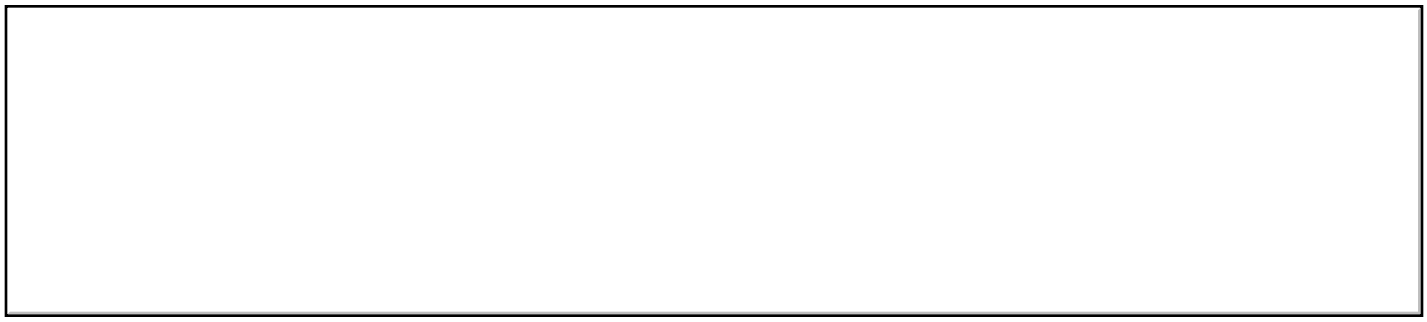
How will we maintain a unique mapping if redirection requests can go to different data centers that are geographically apart? Does our design assume that our DB is consistent geographically?

Show Answer

1 of 3

Design diagram#

A simple design diagram of the URL shortening system is given below.



A design diagram of the URL shortening service

Workflow#

Let’s analyze the system in-depth and how the individual pieces fit together to provide the overall functionality.

Keeping in view the functional requirements, the workflow of the abstract design above would be as follows.

1. **Shortening**: Each new request for short link computation gets forwarded to the short URL generator (SUG) by the application server. Upon successful generation of the short link, the system sends one copy back to the user and stores the record in the database for future use.

Quiz

Question 1

How does our system avoid duplicate short URL generation?

Show Answer

1 of 2

2. **Redirection**: Application servers, upon receiving the redirection requests, check the storage units (caching system and database) for the required record. If found, the application server redirects the user to the associated long URL.

Quiz

Question

How does our system ensure that our data store will not be a bottleneck?

Show Answer

3. **Deletion**: A logged-in user can delete a record by requesting the application server which forwards the user details and the associated URL's information to the database server for deletion. A system-initiated deletion can also be triggered upon an expiry time, as we'll see ahead.
4. **Custom short links**: This task begins with checking the eligibility of the requested short URL. The maximum length allowed is 11 alphanumeric digits. We can find the details on the allowed format and the specific digits in the next lesson. Once verified, the system checks its availability in the database. If the requested URL is available, the user receives a successful short URL generation message, or an error message in the opposite case.

The illustration below depicts how URL shortening, redirection, and deletion work.

URL shortening: The user initiates the request

Question

Upon successful allocation of a custom short URL, how does the system modify its records?

Show Answer

Requirements of TinyURL's Design

Next

Encoder for TinyURL

Mark as Completed

Report an Issue