Dilip Sharma

Confidential & Sensitive

URL Shortner

Design Document

Module.:	URL Shortner
Revision No.:	1.2
Date:	19th Sept 2023

DOCUMENT CONTROL

	NAME	POSITION
Author	Dilip Kumar Sharma	Tech Lead
Reviewed	TBD	TBD
Approved	TBD	TBD
Location	TBD	

REVISION HISTORY

REV	DATE	DESCRIPTION
1.0	20th Sept 2023	Document creation
1.1	21st Sept 2023	Added Architecture Diagram
1.2	22nd Sept 2023	Added Class Diagram

TERMS & ABBREVIATIONS

TERM	DEFINITION
DAO	Data Access Object

REFERENCED DOCUMENTS

REF NO	DOCUMENT NAME	SERVER LOCATION

Table of Contents

1 Introduction 6

1.1 Purpose 6

1.2 Scope 6

1.3 Audience 6

2 Design Overview 6

2.1 About 6

2.2 Requirements 6

2.2.1 Authentication 6

2.2.2 URL Expiry 6

2.2.3 Delete Url 6

2.2.4 Caching 6

2.2.5 Duplicate Url 6

2.2.6 Encoder 7

2.3 Assumptions 7

2.4 Constraints 7

2.5 Limitations 7

2.6 System Environment 7

3 Design Goals 7

3.1 Independent Component 7

3.2 Simplicity 7

3.3 Scalable 7

3.4 Flexible 8

3.5 Readable and Understandable 8

3.6 Highly Available 8

3.7 Low Latency 8

3.8 Unpredictability 8

3.9 Approach 8

4 Application Architecture 9

4.1 High Level Architecture Diagram 9

4.1.1 Client 9

4.1.2 Url Shortner Service 9

4.1.3 Encoder - Decoder 9

4.1.4 Cassandra Database 9

4.2 Complete Architecture Diagram 9

4.2.1 Client 10

4.2.2 Api Router 10

4.2.2.1 Auth Router 10

4.2.2.2 User Router 10

4.2.2.3 Url Router 10

4.2.3 Config 10
4.2.3.1 Schema 11
4.2.3.2 Dependency 11
4.2.3.3 Config Dependency 12
4.2.3.4 Database dependency 12
4.2.3.5 Auth Dependency 12
4.2.4 OAuth 12
4.2.5 Exceptions 12
4.2.6 Services 12
4.2.7 Utils 12
4.2.8 DAO 12
4.2.9 Cassandra Driver 12
4.3 Class diagram 12
4.3.1 Authentication 13
4.4 Sequence diagram 13

5 Others 14

5.1 To Do 14
5.2 References 14

1 Introduction

1.1 Purpose

This document details the technical design and implementation of Url Shortner.

Its main purpose is to detail the functionality which will be provided by each component and show how the various components interact in the design.

1.2 Scope

The scope of this document is limited to designing of Url Shortner component.

1.3 Audience

The intended audiences for this document are, Project manager, the project development teams, technical architects, database designers, testers.

2 Design Overview

2.1 About

URL shortening service is used to generate shorter aliases for long URLs. In this kind of web service if a user gives a long URL then the service returns a short URL and if the user gives a short URL then it returns the original long URL.

Deletion: Users should be able to delete a short link generated by our system, given the rights.

2.2 Requirements

2.2.1 Authentication

- ✓ Optional authentication with OAuth2 should be provided, though to generate short link authentication is not mandatory.

2.2.2 URL Expiry

- ✓ Each short URL should have an expiry date.
- ✓ For non authenticated users, this expiry date is shorter than their authenticated counter parts.

2.2.3 Delete Url

- ✓ Authenticated user will have ability to delete/inactive the shorter url from the system.

2.2.4 Caching

- ✓ Use caching mechanism for low latency.

2.2.5 Duplicate Url

- ✓ If generated short url is duplicate in database then regenerate it.

2.2.6 Encoder

- ✓ Use MD5 hash for encoding the long url into short url.

2.3 Assumptions

- Users
  - ✓ 1M users per day
  - ✓ Default Keep alive time – 1 Year
  - ✓ URL len – 7 Char

- Storage
  - ✓ Avg URL size – 2 KB – 2048 Bytes
  - ✓ Short URL Size – 7 Bytes
  - ✓ Expiry Date – 10 Bytes
  - ✓ Is Active – 1 Bytes

2.4 Constraints

- URL needs to be expired after given expiry date. This yet to be implemented.
- When multiple instances of shortner service run duplicate short url could be generated by different services. This yet to be implemented.

2.5 Limitations

- Not all edge cases, for url shortner service, have been covered in this document.

2.6 System Environment

- This design supports Docker, Docker Compose.
- This is implemented in Python 3.8.

3 Design Goals

3.1 Independent Component

- We need to bring uniformity, not only in python source code, but also in DB tables structure.

3.2 Simplicity

Source code and Database table structure should be less complex or say should be simple to understand.

- Simplifying the source code structure.
- Simplifying the database table structure.
- The short links generated by our system should be easily readable, distinguishable, and typeable.

3.3 Scalable

To be able to scale both up and down to support; -

- Varying number of users.
- Our system should be horizontally scalable with increasing demand.

3.4 Flexible

Ability of the application to adapt and evolve to accommodate new requirements without affecting the existing operations.

Flexibility in the application can be achieved by; -

- Use of Gang of Four design patterns.
- Use of SOLID principle of design patterns.
- Use of object-oriented programming principles.
- Designing database tables.

3.5 Readable and Understandable

Software is meant for modification/improvements. Fellow developers should be able to understand the code.

This could be achieved by; -

- Coding guidelines.
- UML diagrams.
- Sequence Diagrams
- Comments/Description of classes and methods used.
- Documentation (Doc string comments in Python), ReadME.

3.6 Highly Available

Our system should be highly available, because even a fraction of the second downtime would result in URL redirection failures.

Since our system's domain is in URLs, we don't have the leverage of downtime, and our design must have fault-tolerance conditions instilled in it.

3.7 Low Latency

The system should perform at low latency to provide the user with a smooth experience.

3.8 Unpredictability

From a security standpoint, the short links generated by our system should be highly unpredictable.

This ensures that the next-in-line short URL is not serially produced,

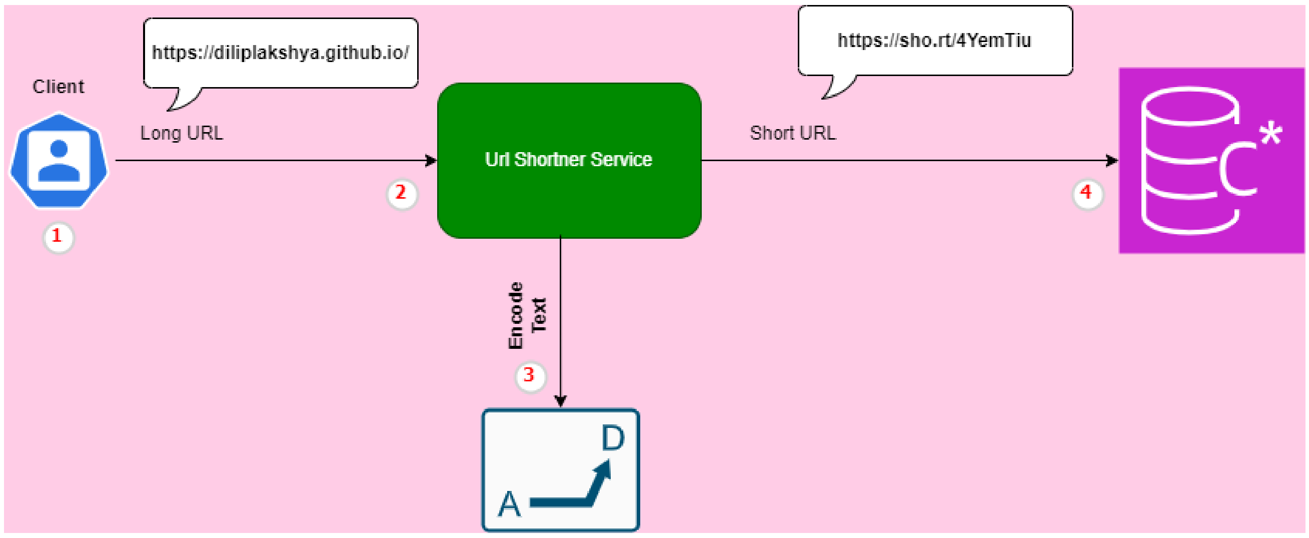
Eliminating the possibility of someone guessing all the short URLs that our system has ever produced or will produce.

3.9 Approach

- Expiry time: There must be a default expiration time for the short links, but users should be able to set the expiration time based on their requirements.
- Url should not be predictable.
- Need to design both python source code as well as DB table structure.
- Adding DAL between core logic and data.

4 Application Architecture

4.1 High Level Architecture Diagram



4.1.1 Client

- The user requesting URL Shortner service from browser or any rest api client.

4.1.2 Url Shortner Service

- Main service responsible for shortning the long url as well as returning long url when short url is passed.

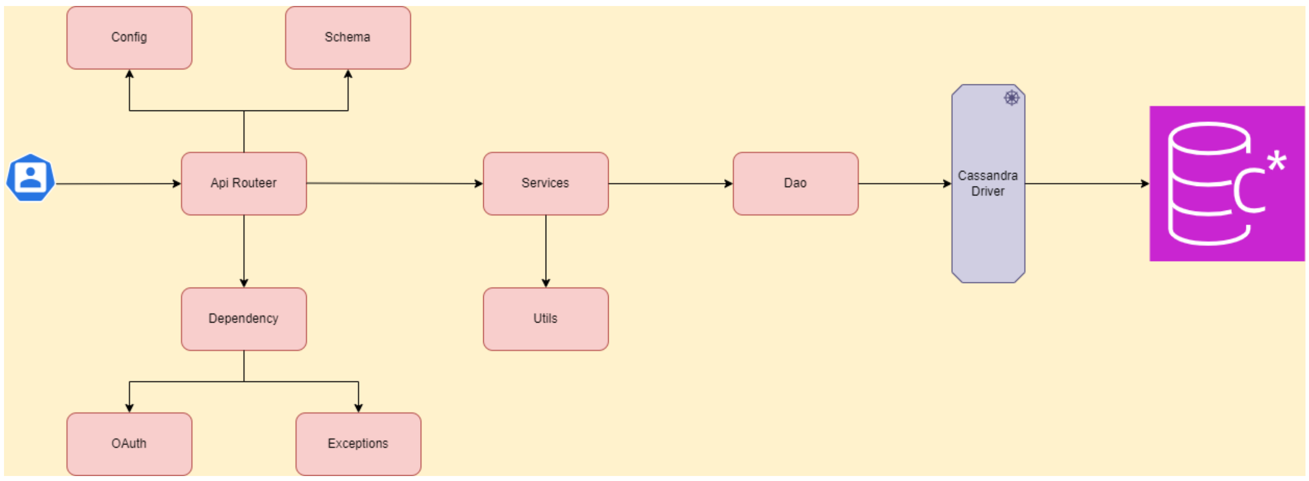
4.1.3 Encoder - Decoder

Responsible for encoding the long url text into short text.

4.1.4 Cassandra Database

Database to store long and short urls.

4.2 Complete Architecture Diagram



4.2.1 Client

Any rest api client.

4.2.2 Api Router

Fast Api router for accepting client's API request.

It is further divided into 3 parts; -

4.2.2.1 Auth Router

Responsible for handling user authentication.

4.2.2.2 User Router

It handles user related API operations. Like creating new user, deleting existing user, information about currently logged in user.

4.2.2.3 Url Router

It is responsible for creating short url from long url.

It fetch original long url when short url is provided.

It can mark existing short url as inactive.

4.2.3 Config

- Configuration files to hold Url Shortner's static information.

.env.app

Environment Variable	Ex. VALUE
API_HOST	0.0.0.0
API_PORT	5001
RELOAD	True
LOG_CONFIG_FILE	/app/src/config/development/logging.conf

LOGS_DIR	/tmp/url_shortner
LOG_FILE_NAME	api.log
JWT_ALGORITHM	HS256
TOKEN_EXPIRY_TIME	15
SECRET_KEY	I6YaDVSRPw5Ux+2paY4u4ToMKtZXQoBj
PASSWORD_LENGTH	8
BACKEND_CORS_ORIGINS	["http://localhost:5001", "http://127.0.0.1:5001"]
ALLOW_METHODS	["GET", "POST", "DELETE"]
ALLOW_HEADERS	["*"]
MAX_EXPIRES_IN_FOR_LOGGED_IN_USERS	365
MAX_EXPIRES_IN_FOR_NON_LOGGED_IN_USERS	30
CASSANDRA_HOST	shortner-development
CASSANDRA_KEYSPACE	shortner_app

logging.json

ATTRIBUTE	Ex. VALUE
Format	[%(levelname)s] %(asctime)s: [%(threadName)-17s] [%(filename)s:%(funcName)s:%(lineno)d] %(message)s
Level	10

Logging Level; -

LEVEL	NUMERIC VALUE
NOT SET	0
DEBUG	10
INFO	20
WARNING	30
ERROR	40
CRITICAL	50

4.2.3.1 Schema

Pydantic schema to be used for api input request and response.

4.2.3.2 Dependency

All api dependencies used by FastAPI.

It is further divided into 3 parts; -

4.2.3.3 Config Dependency

Responsible for providing config related dependency like creating FastApi config object which will be consumed by rest of the application.

4.2.3.4 Database dependency

Responsible for creating database related dependencies like creating database session object.

4.2.3.5 Auth Dependency

Responsible for providing authentication related dependencies.

4.2.4 OAuth

OAuth2 is used with basic password flow to allow user to login.

This is optional feature to be used and hence to generate short url user does not need to login first.

4.2.5 Exceptions

Custom exceptions are created.

DB Exception

Auth Exceptions

User Exceptions

4.2.6 Services

All business logic is written in service classes.

4.2.7 Utils

All utilities modules fall under this category which would be consumed by other modules in the application, like generating hashed password.

4.2.8 DAO

Data access object used for segregating python modules with database. So all DB related calls pass through DAO module. This module uses Cassandra python driver to communicate with Cassandra database.

4.2.9 Cassandra Driver

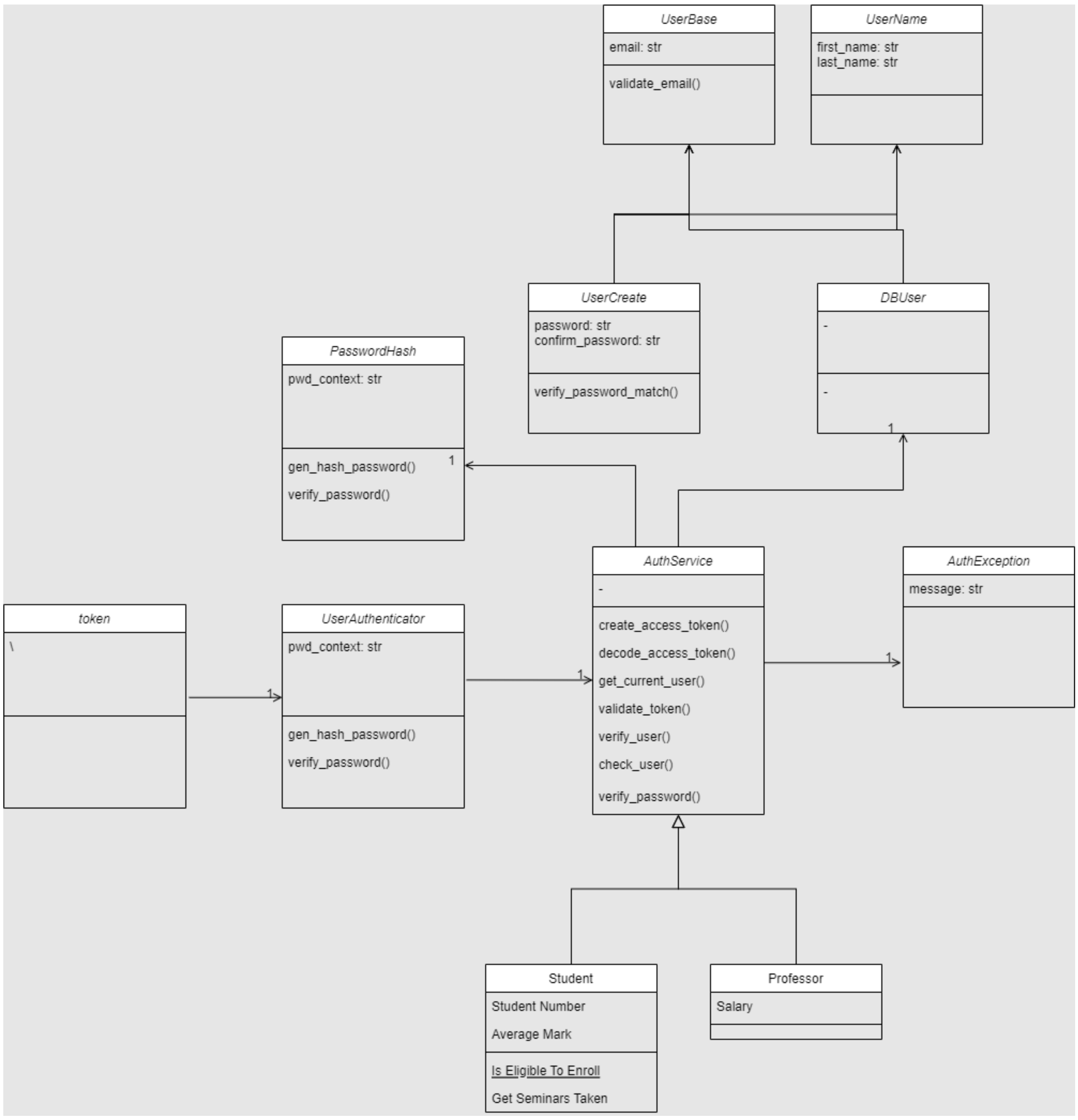
Python driver used to communicate with Cassandra for the purpose of doing CRUD operations.

Database

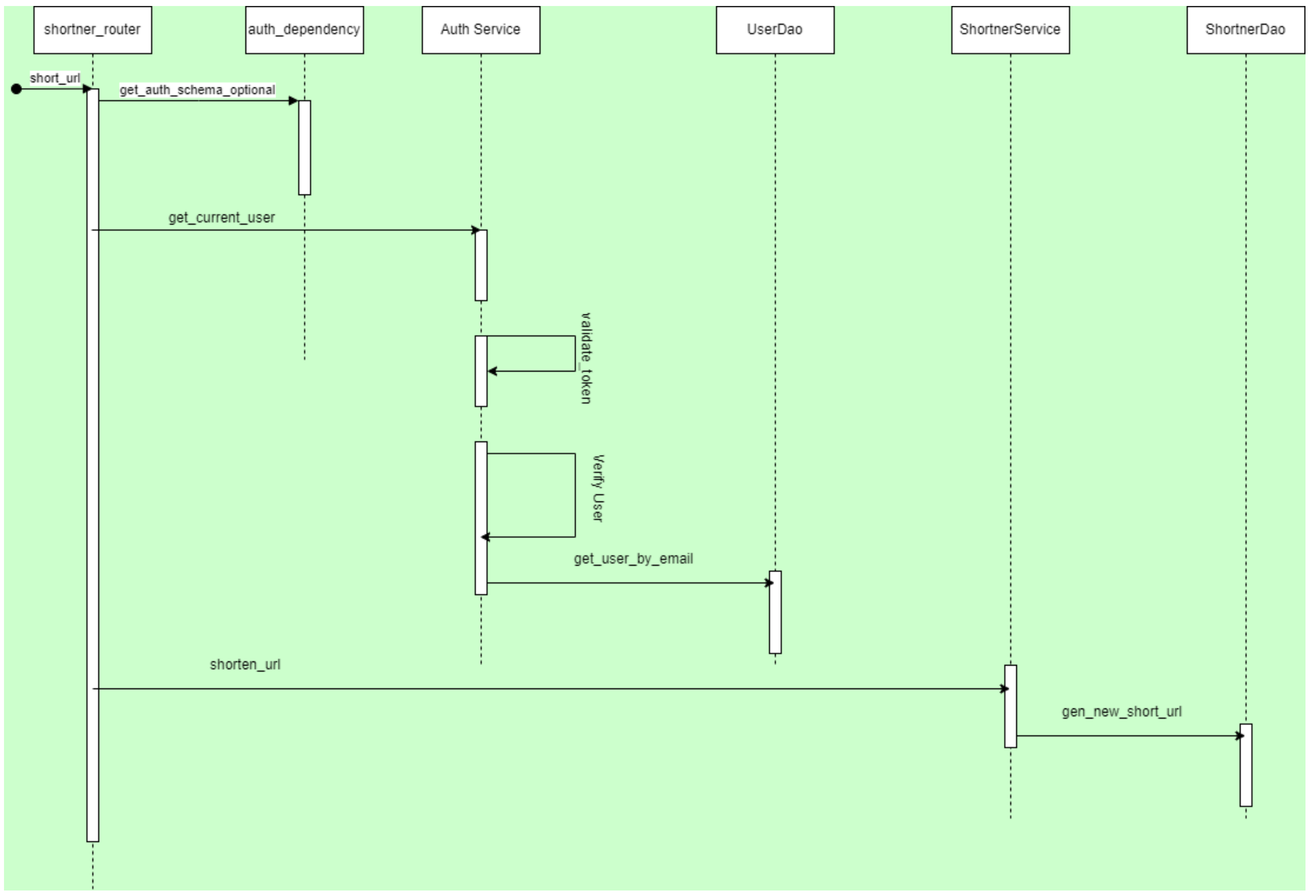
Datastax DSE Cassandra database is used for storing application data.

4.3 Class diagram

4.3.1 Authentication



4.4 Sequence diagram



5 Others

5.1 To Do

Edge cases for Url Shortner; -

TBD.

5.2 References

TBD