Requirements Gathering:

1. Realistic mini DNS service API
2. API handle DNS records like A records(IPv4) and CNAME records.
3. Service should reflect real world DNS constarints and behavior accurately.

DNS Record Types Supported:

1. A record – hostname to IPv4 addresses (one hostname can have multiple A records)
2. CNAME record – hostname alias to another hostname (only one CNAme per hostname) (no hostname can coexist)

Required API Endpoints

1. Add DNS Record
   -> POST /api/dns

2.Resolve Hostname

-> GET /api/dns/{hostname}

3. List DNS Records for hostname

-> GET /api/dns/{hostname}/records

4. Delete DNS Records

->DELETE /api/dns/{hostname}

DNS Implementation constraints

1. Multiple A records are allowed per hostname
2. Only 1 CNAME record per hostname and a hostname with a CNAMe cannot have other records.
3. CNAME chaining allowed (avoid circular references)
4. Validation:
   1. Prevent conflicting records (CNAME and A on same hostname)
   2. Prevent duplicate records
   3. Proper validation for hostname/IP formatting

5. Proper HTML Error Status Code and unit tests covering logic and edge cases

STEP 2: Advanced Constraints and requirements analysis

About DNS records:

https://www.site24x7.com/learn/dns-record-types.html

Points to note:

1. Right now we are supporting A records and CNAME Records only.
   Also one hostname can have multiple A records.
2. CNAME records: only one CNAME record per hostname.
3. Also CNAME and A name can't coexist.(Validation check)
4. CNAME chaining allowed

DNS Record structure:

hostname	string	DNS name
type	enum	It's either A or CNAME now. Also easy to extend for IPV6
value	String/ array of strings	Array of strings for A records because it allows multiple records, whereas CNAME only allows one
createdAt	timestamp	For future TTL implementation

Was also thinking of pointsTo but that breaks originality of a mini DNS service as originally, we do not use this structure and also we can rather cache this than store it as meta data.

DNS Error codes:

https://help.dnsfilter.com/hc/en-us/articles/4408415850003-DNS-return-codes.
Since we are going to be returning error codes, let's combine it with HTML Status codes like 200, 404, 403 and 500, 401-Unauthorised,etc.

HIGH LEVEL DESIGN:

MAIN OBJECTIVES:

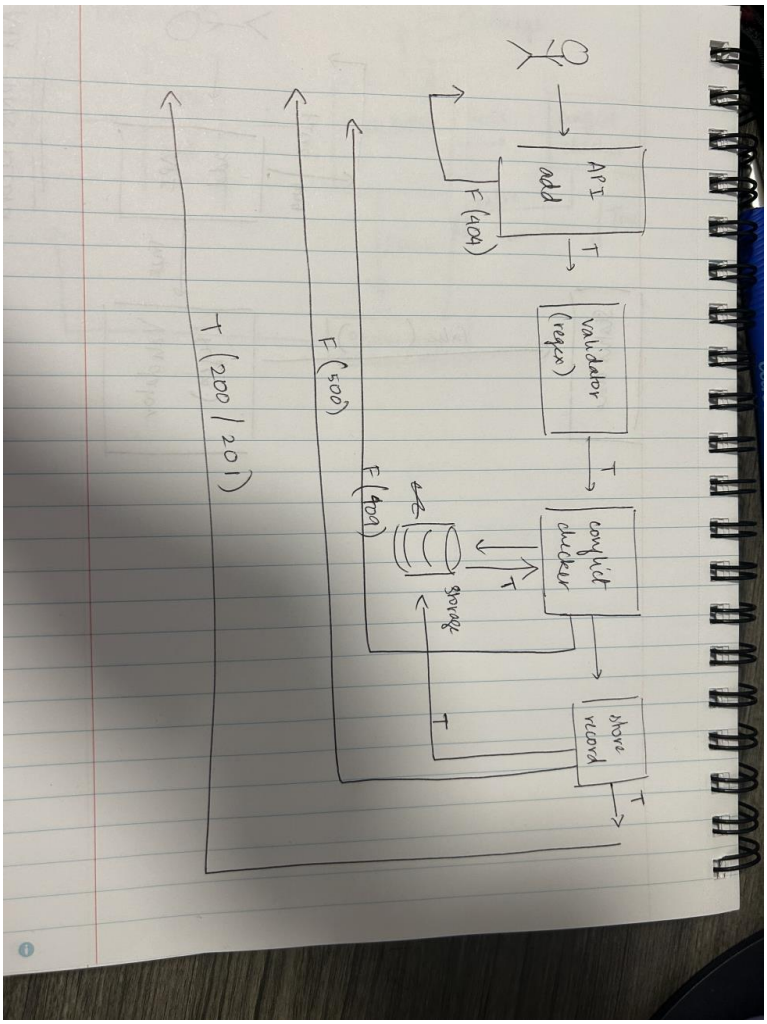
1. Maintable
2. Scalable
3. Resilient

https://www.geeksforgeeks.org/domain-name-system-dns-in-application-layer/

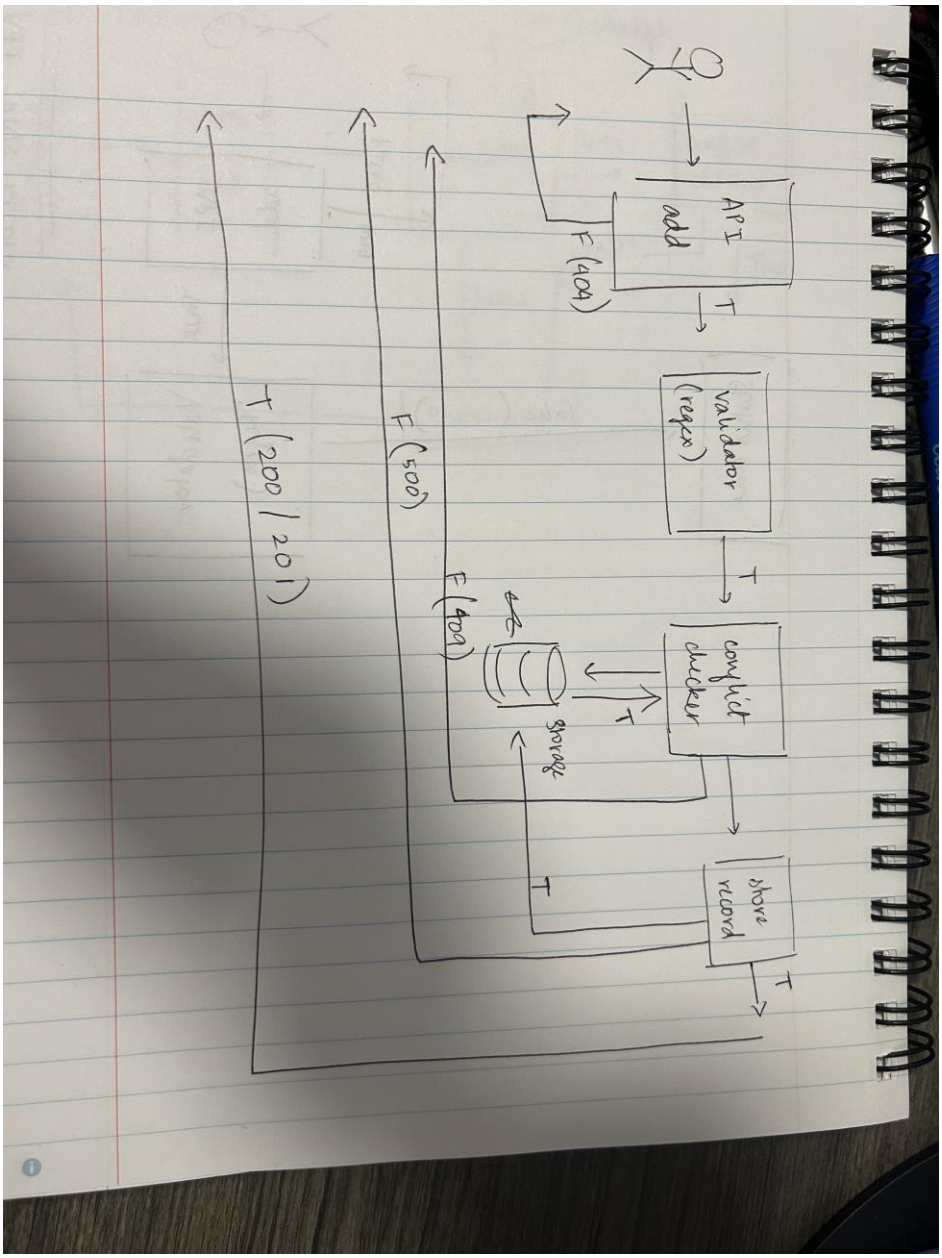
DNS service needs :

1. API Layer
2. Lookup
3. Resolver
4. Caching
5. Storage
6. Validator
7. Testing

System architecture for Add:



System architecture for Delete:



Let’s add TTL field, docker containerization, asyncio background task for cleanup, rate limiter using fast

Now technologies needed:

1. Backend Framework
2. Storage system
3. Authentication

4. Caching Layer
5. Rate Limiting
6. Containerization
7. Async Task Handling
8. Testing

API Framework: Fast API is better because it is ideal for async features unlike Flask

Data storage: we can go with postgres and basically structured db because dns rules are stricter and for circular CNAMEs or lookups join is better and bulk import/export is better.

Caching: Redis has good caching and also TTL expiration support

Authentication and rate limiting: Basic API key in header and slow rate in Fast API for rate limiting

Containerization: Docker (requested) and portable

Async handling: FastAPI

Testing: Pytest

Code Modularization:
Structure:

1. Need an api call for routing
2. Need a core function for config, error codes,logging and lifecycle (optional)
3. Auth folder for basic authentication and rate limiting
4. Models for building our records schema and db
5. Services module for handling bulk events, CRUD, ttl_cleanup, validator
6. Utils for basic hostname and record validation
7. Tests for running addition, deletion, db insertion, records recovery and resolver testing.
8. Dockerfile for containerization

We can always make a simpler one page module but the purpose fo modularizing this is to ensure that it can be scalable beyond , for instance we have separate files for record_schema and record_db to ensure we know the structure of the incoming Record Type and expand it further.

Explanation and reasoning behind each file:

1. Dns_routes:
   This is where the main routing occurs.
   We have integrated rate limiter approach here for instance if you keep sending more than 10 requests per minute we get a 429 response code.
   We work on add, deletion, records listing, bulk import, bulk export, and resolver operations here.
2. Authentication Module:
   1. Api kEY -> BASIC AUTHENTIACTION CAN BE DONE BY USING AN API KEY. Here the secret key is “supersecret”. We can always change the code in the .env file.
   2. Rate Limiter:We are using slowapi for rate limiting

3. Core Module:

1. config file:

This is where we have settings for api_key, db_url , redis_url and testing flag.the testing flag ensures that the background cleanup task doesn’t happen during testing leading to concurrency.

2. errors file:

This is the error codes standard file, here we have the possible errors listed and the appropraite messages. The reason behind having this file is to be able to expand to multiple languages messages in the future.

3. Lifecycle:

This is for logging the starting and ending of the FastAPI. This will be more useful in a production environment.

4. Logger:

Main component to start logging for testing and production.

4. Models Module:

This is the module where you can add more schemas if needed and make changes:

1. Record db: contains enum for recordtype and base dnsrecord structure.
2. Record schema:
   Contains individual validations for the records request schema and the discrimitator for each schema is it’s type value(Enum from record db)
3. Response schema: contains schema for how response records must look.

5.Services Module:

This is the module for bulk, crud and ttl cleanup operations basically anything crud related or manipuative lies here.

Bulk handler: handles bulk addition by separating the record in the json and finding operation clue an performing it. (has few errors now)

CRUD:

Where the deletion, insertion takes place

Resolver: we do cname chaining by using graph dfs and find the values.

Ttl_cleanup: checks the expiring time and starts periodic cleanup by using asyncio task for background cleaning. This is done in a separate thread and asynchronously because this should not affect the crud operations.

Validator: place where cname chaining is done and record type conflict is checked.

Storage:
db: where main db is initialised

Redis: here is how the cache is stored, invalidated and resolved.

Utilities module: for storing utility functions like hostname for validating hostname and record utils for validating records.

Additional requirements done:

1. Redis Cache
2. 2. Bulk import export
3. TTL expiry background
4. Docker containerization
5. Simple auth and rate limiting
6. MX TXT AAAA support
7. Centralised Logging Queries

Testing done for:

1. Bulk insertion, deletion
2. Add records

3. Delete records
4. List records
5. Resolve records
6. DB init