Computer Networks project report

BCSE III

<u>Section-A1</u>

<u>Group members</u>

Siddhart Mondal-002210501020

Sugata Laha-002210501040

Subham Mondal-002210501028

Soumyadeep Singh-002210501018

Task-Implement DNS protocol using TCP/UDP socket Code

What is DNS?

The Domain Name System (DNS) is the phonebook of the Internet. Humans access information online through domain names, like nytimes.com or espn.com. Web browsers interact through Internet Protocol (IP) addresses. DNS translates domain names to IP addresses so browsers can load Internet resources.

Each device connected to the Internet has a unique IP address which other machines use to find the device. DNS servers eliminate the need for humans to memorize IP addresses such as 192.168.1.1 (in IPv4), or more complex newer alphanumeric IP addresses such as 2400:cb00:2048:1::c629:d7a2 (in IPv6).

How does DNS work?

The process of DNS resolution involves converting a hostname (such as www.example.com) into a computer-friendly IP address (such as 192.168.1.1). An IP address is given to each device on the Internet, and that address is necessary to find the appropriate Internet device - like a street address is used to find a particular home. When a user wants to load a webpage, a translation must occur between what a user types into their web browser (example.com) and the machine-friendly address necessary to locate the example.com webpage.

In order to understand the process behind the DNS resolution, it's important to learn about the different hardware components a DNS query must pass between. For the web browser, the DNS lookup occurs "behind the scenes" and requires no interaction from the user's computer apart from the initial request.

There are 4 DNS servers involved in loading a webpage:

1. **DNS recursor** - The recursor can be thought of as a librarian who is asked to go find a particular book somewhere in a library. The DNS recursor is a server designed to receive queries from client machines through applications such as web browsers.

Typically the recursor is then responsible for making additional requests in order to satisfy the client's DNS query.

2. **Root nameserver** - The root server is the first step in translating (resolving) human readable host names into IP addresses. It can be thought of like an index in a library that points to different racks of books - typically it serves as a reference to other more specific locations.

3. **TLD nameserver** - The top level domain server (TLD) can be thought of as a specific rack of books in a library. This nameserver is the next step in the search for a specific IP address, and it hosts the last portion of a hostname (In example.com, the TLD server is "com").

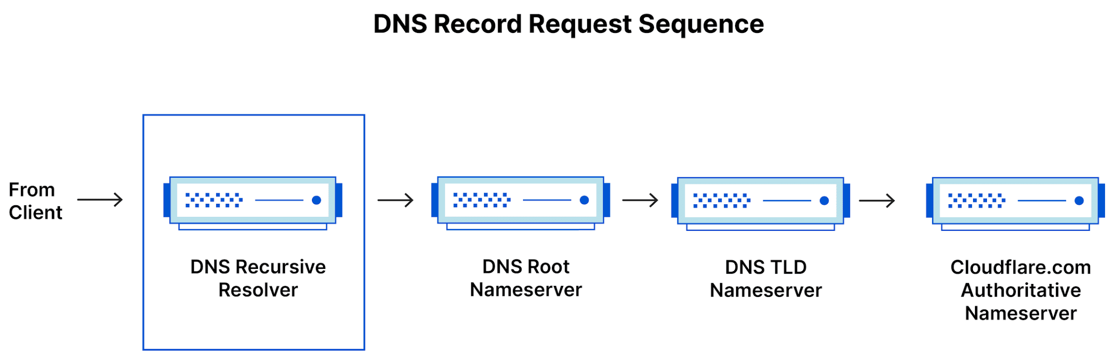
4. **Authoritative nameserver** - This final nameserver can be thought of as a dictionary on a rack of books, in which a specific name can be translated into its definition. The authoritative nameserver is the last stop in the nameserver query. If the authoritative name server has access to the requested record, it will return the IP address for the requested hostname back to the DNS Recursor (the librarian) that made the initial request.

What's the difference between an authoritative DNS server and a recursive DNS resolver?

Both concepts refer to servers (groups of servers) that are integral to the DNS infrastructure, but each performs a different role and lives in different locations inside the pipeline of a DNS query. One way to think about the difference is the recursive resolver is at the beginning of the DNS query and the authoritative nameserver is at the end.

Recursive DNS resolver

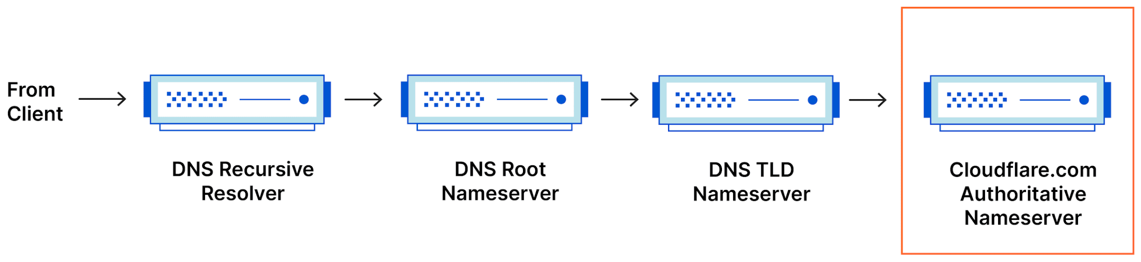
The recursive resolver is the computer that responds to a recursive request from a client and takes the time to track down the DNS record. It does this by making a series of requests until it reaches the authoritative DNS nameserver for the requested record (or times out or returns an error if no record is found). Luckily, recursive DNS resolvers do not always need to make multiple requests in order to track down the records needed to respond to a client; caching is a data persistence process that helps short-circuit the necessary requests by serving the requested resource record earlier in the DNS lookup.



Authoritative DNS server

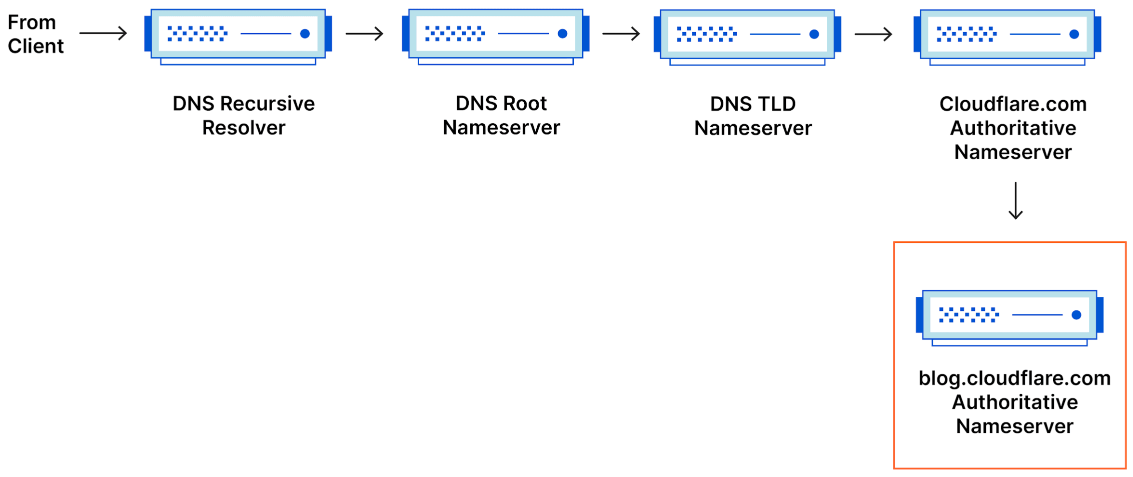
Put simply, an authoritative DNS server is a server that actually holds, and is responsible for, DNS resource records. This is the server at the bottom of the DNS lookup chain that will respond with the queried resource record, ultimately allowing the web browser making the request to reach the IP address needed to access a website or other web resources. An authoritative nameserver can satisfy queries from its own data without needing to query another source, as it is the final source of truth for certain DNS records.

DNS Record Request Sequence



It's worth mentioning that in instances where the query is for a subdomain such as foo.example.com or blog.cloudflare.com, an additional nameserver will be added to the sequence after the authoritative nameserver, which is responsible for storing the subdomain's CNAME record.

CNAME DNS Record Request Sequence


From Client
DNS Recursive Resolver
DNS Root Nameserver
DNS TLD Nameserver
Cloudflare.com Authoritative Nameserver
blog.cloudflare.com Authoritative Nameserver

There is a key difference between many DNS services and the one that Cloudflare provides. Different DNS recursive resolvers such as Google DNS, OpenDNS, and providers like Comcast all maintain data center installations of DNS recursive resolvers. These resolvers allow for quick and easy queries through optimized clusters of DNS-optimized computer systems, but they are fundamentally different than the nameservers hosted by Cloudflare.

Cloudflare maintains infrastructure-level nameservers that are integral to the functioning of the Internet. One key example is the f-root server network which Cloudflare is partially responsible for hosting. The F-root is one of the root level DNS nameserver infrastructure components responsible for the billions of Internet requests per day. Our Anycast network puts us in a unique position to handle large volumes of DNS traffic without service interruption.

What are the steps in a DNS lookup?

For most situations, DNS is concerned with a domain name being translated into the appropriate IP address. To learn how this process works, it helps to follow the path of a DNS lookup as it travels from a web browser, through the DNS lookup process, and back again. Let's take a look at the steps.

Note: Often DNS lookup information will be cached either locally inside the querying computer or remotely in the DNS infrastructure. There are typically 8 steps in a DNS lookup. When DNS information is cached, steps are skipped from the DNS lookup process which makes it quicker. The example below outlines all 8 steps when nothing is cached.

The 8 steps in a DNS lookup:

1. A user types ‘example.com’ into a web browser and the query travels into the Internet and is received by a DNS recursive resolver.

2. The resolver then queries a DNS root nameserver (.).

3. The root server then responds to the resolver with the address of a Top Level Domain (TLD) DNS server (such as .com or .net), which stores the information for its domains. When searching for example.com, our request is pointed toward the .com TLD.

4. The resolver then makes a request to the .com TLD.

5. The TLD server then responds with the IP address of the domain's nameserver, example.com.

6. Lastly, the recursive resolver sends a query to the domain's nameserver.

7. The IP address for example.com is then returned to the resolver from the nameserver.

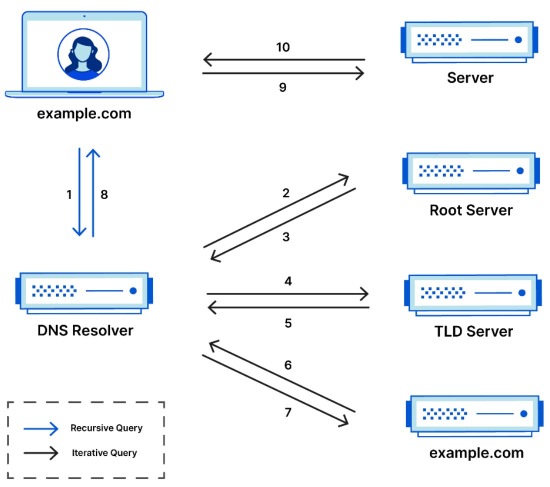
8. The DNS resolver then responds to the web browser with the IP address of the domain requested initially.

Once the 8 steps of the DNS lookup have returned the IP address for example.com, the browser is able to make the request for the web page:

9. The browser makes a HTTP request to the IP address.

10 The server at that IP returns the webpage to be rendered in the browser (step 10).

Complete DNS Lookup and Webpage Query

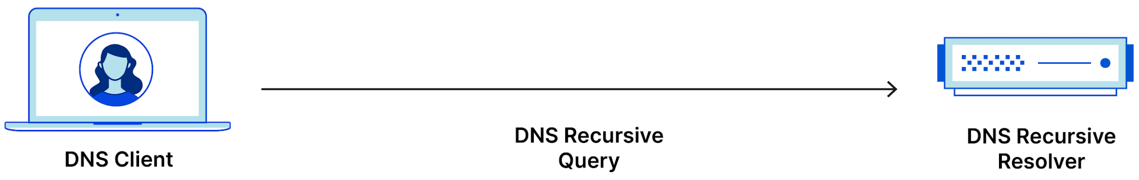


What is a DNS resolver?

The DNS resolver is the first stop in the DNS lookup, and it is responsible for dealing with the client that made the initial request. The resolver starts the sequence of queries that ultimately leads to a URL being translated into the necessary IP address.

Note: A typical uncached DNS lookup will involve both recursive and iterative queries.

It's important to differentiate between a recursive DNS query and a recursive DNS resolver. The query refers to the request made to a DNS resolver requiring the resolution of the query. A DNS recursive resolver is the computer that accepts a recursive query and processes the response by making the necessary requests.



What are the types of DNS queries?

In a typical DNS lookup three types of queries occur. By using a combination of these queries, an optimized process for DNS resolution can result in a reduction of distance traveled. In an ideal situation cached record data will be available, allowing a DNS name server to return a non-recursive query.

3 types of DNS queries:

1. Recursive query - In a recursive query, a DNS client requires that a DNS server (typically a DNS recursive resolver) will respond to the client with either the requested resource record or an error message if the resolver can't find the record.

2. Iterative query - in this situation the DNS client will allow a DNS server to return the best answer it can. If the queried DNS server does not have a match for the query name, it will return a referral to a DNS server authoritative for a lower level of the domain namespace. The DNS client will then make a query to the referral address. This process continues with additional DNS servers down the query chain until either an error or timeout occurs.

3. Non-recursive query - typically this will occur when a DNS resolver client queries a DNS server for a record that it has access to either because it's authoritative for the record or the record exists inside of its cache. Typically, a DNS server will cache DNS records to prevent additional bandwidth consumption and load on upstream servers.

What is DNS caching? Where does DNS caching occur?

The purpose of caching is to temporarily stored data in a location that results in improvements in performance and reliability for data requests. DNS caching involves storing data closer to the requesting client so that the DNS query can be resolved earlier and additional queries further down the DNS lookup chain can be avoided, thereby improving load times and reducing bandwidth/CPU consumption. DNS data can be cached in a variety of locations, each of which will store DNS records for a set amount of time determined by a time-to-live (TTL).

Browser DNS caching

Modern web browsers are designed by default to cache DNS records for a set amount of time. The purpose here is obvious; the closer the DNS caching occurs to the web browser, the fewer processing steps must be taken in order to check the cache and make the correct requests to an IP address. When a request is made for a DNS record, the browser cache is the first location checked for the requested record.

In Chrome, you can see the status of your DNS cache by going to chrome://net-internals/#dns.

Operating system (OS) level DNS caching

The operating system level DNS resolver is the second and last local stop before a DNS query leaves your machine. The process inside your operating system that is designed to handle this query is commonly called a “stub resolver” or DNS client. When a stub resolver gets a request from an application, it first checks its own cache to see if it has the record. If it does not, it then sends a DNS query (with a recursive flag set), outside the local network to a DNS recursive resolver inside the Internet service provider (ISP).

When the recursive resolver inside the ISP receives a DNS query, like all previous steps, it will also check to see if the requested host-to-IP-address translation is already stored inside its local persistence layer.

The recursive resolver also has additional functionality depending on the types of records it has in its cache:

1. If the resolver does not have the A records, but does have the NS records for the authoritative nameservers, it will query those name servers directly, bypassing several steps in the DNS query. This shortcut prevents lookups from the root and .com nameservers (in our search for example.com) and helps the resolution of the DNS query occur more quickly.

2. If the resolver does not have the NS records, it will send a query to the TLD servers (.com in our case), skipping the root server.

3. In the unlikely event that the resolver does not have records pointing to the TLD servers, it will then query the root servers. This event typically occurs after a DNS cache has been purged.

Code

index.js

```
import dgram from "dgram";

import dnsPacket from "dns-packet";

import dotenv from "dotenv";

import connectDB from "./db/index.js";

import { recordModel } from "./models/record.model.js";

let cache={};

dotenv.config();

// Create a UDP server

const server = dgram.createSocket('udp4');

// Connect to the database

connectDB();

const getDNSRecord=async (domainName)=>

{

  if(domainName in cache)

  {

    console.log("Found in cache !");

    return cache[domainName];
  }

  else

  {

      console.log("Cache miss!");

      try

      {

        const parts=domainName.split(".");

        const record=await
recordModel.find({name:parts[parts.length-1]});

        return record;

      }

      catch(error)

      {

        console.log("Error while fetching! ",error);

      }

  }

}

server.on('message', async (msg, rinfo) => {

  try {

    // Decode the incoming DNS query

    const incomingMsg = dnsPacket.decode(msg);
    // Log the query

    if (incomingMsg.questions && incomingMsg.questions[0]) {

      const domainName = incomingMsg.questions[0].name;

      console.log(`Query for: ${domainName}`);

      console.log("Not found in cache");

      const parts=domainName.split(".");

      // Prepare the DNS response

     const response = {

        type: 'response',

        id: incomingMsg.id,

        flags: dnsPacket.RECURSION_DESIRED |
dnsPacket.RECURSION_AVAILABLE | dnsPacket.AUTHORITATIVE_ANSWER, //
Proper flags

        questions: incomingMsg.questions,

        answers: [], // Default empty answers

        authority:[],

        additional:[]

      };



      try {

        // Search for the domain in the database
```

```
let record = await getDNSRecord(domainName);

if(record.length>1)

{

    response.authority=record;

}

else if (record.length==1)

{

  if(record[0].type=='A')

  {

    response.answers.push({

      type: record.type, // For example, 'A' for IPv4

      class: 'IN', // Internet class

      name: domainName,

      ttl: 300, // Time-to-live in seconds

      data: record.ip, // IP address from the database

    });

    cache[domainName]=record;

  }

  else if(record[0].type=='NS')

  {
```

```
  response.authority.push_back(

    {

      type:record[0].type,

      class:'IN',

      name:record[0].name,

      ttl:300,

      data:record[0].target

    }

  )

}

else if(record[0].type=='CNAME')

{

  response.authority.push(

    {

      type:record[0].type,

      name:record[0].name,

      ttl:300,

      data:record[0].target

    }

  )
```

```
          }
        } else {
          // If the domain is not found, set NXDOMAIN (Response
Code = 3)
          response.flags |= 3;
        }
      } catch (error) {
        console.error(`Error fetching record from DB:
${error.message}`);
        // Set NXDOMAIN if a DB error occurs
        response.flags |= 3;
      }


      // Encode the DNS response and send it
      const responseBuffer = dnsPacket.encode(response);
      server.send(responseBuffer, rinfo.port, rinfo.address, (err)
=> {
        if (err) {
          console.error(`Failed to send response:
${err.message}`);
        }
      });
```

```
    } else {

      console.warn("Received a malformed DNS query.");

    }

  } catch (error) {

    console.error(`Error processing DNS request: 
${error.message}`);

  }

});



// Start listening on the specified port

const port=process.env.PORT|| 10053;

console.log(port);

server.bind(port, () => {

  console.log(`DNS Server is listening on port ${port}`);

});
```

db/index.js

```
import mongoose from "mongoose";

import dotenv from "dotenv";
```

```
dotenv.config();



const connectDB=async ()=>

{

    try

    {

        const
connectionString=`${process.env.MONGODB_URL}/${process.env.DATABAS
E_NAME}`;

        const db=await mongoose.connect(connectionString);

        console.log("MongoDB connected! ",db.connection.host);

        return db.connection;

    }

    catch(error)

    {

        console.log("MongoDB connection failed! ",error);

        process.exit(1);

    }

}

export default connectDB;
```

models/record.model.js

```
import mongoose from "mongoose";

const recordSchema=new mongoose.Schema({

    name:

    {

        type:String,

        required:true,

    },

    type:

    {

        type:String,

        required:true

    },

    target:

    {

        type:String

    },

    text:

    {

        type:String
    },
    priority:
    {
        type:Number
    }
});
const recordModel=mongoose.model('Record',recordSchema);
export {recordModel};
```

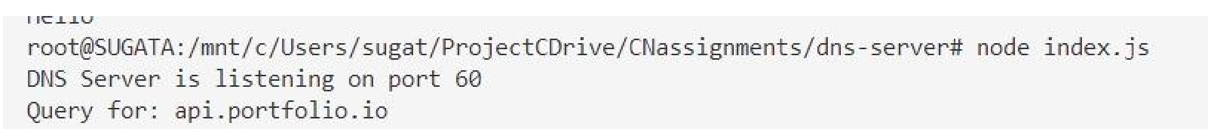
test.js

```
import connectDB from "./db/index.js";
import { recordModel } from "./models/record.model.js";
import { generatedData } from "./generatedData.js";
const connection=connectDB();
console.log(connection);
const insertRecords=async ()=>{
  try
  {
    await recordModel.insertMany(generatedData);
  }
  catch(error)
  {

    console.log(error);

  }

}

insertRecords();
```

generatedData.js

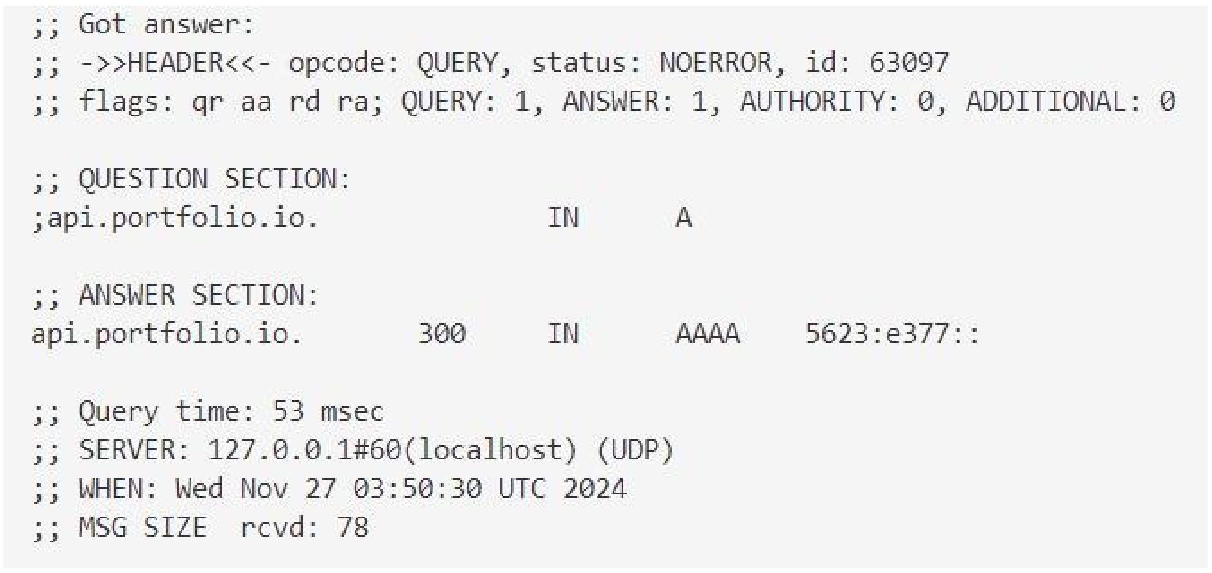
```
const generatedData=[

    {

      "name": "verma.com",

      "type": "AAAA",

      "ip": "2001:db8::921f:5da1:bb1d:0424:533a:fb05"

    },

    {

      "name": "raj.com",

      "type": "A",

      "ip": "192.0.2.58"

    },

//more data are there

]

export {generatedData};
```

Output



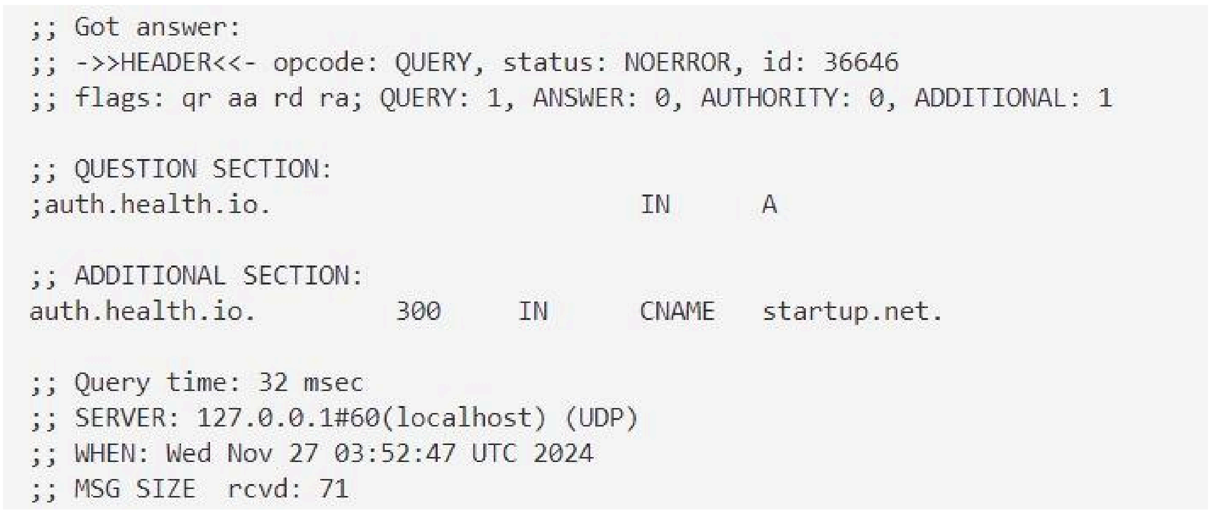
```
hello
root@SUGATA:/mnt/c/Users/sugat/ProjectCDrive/CNassignments/dns-server# node index.js
DNS Server is listening on port 60
Query for: api.portfolio.io
```

For A record



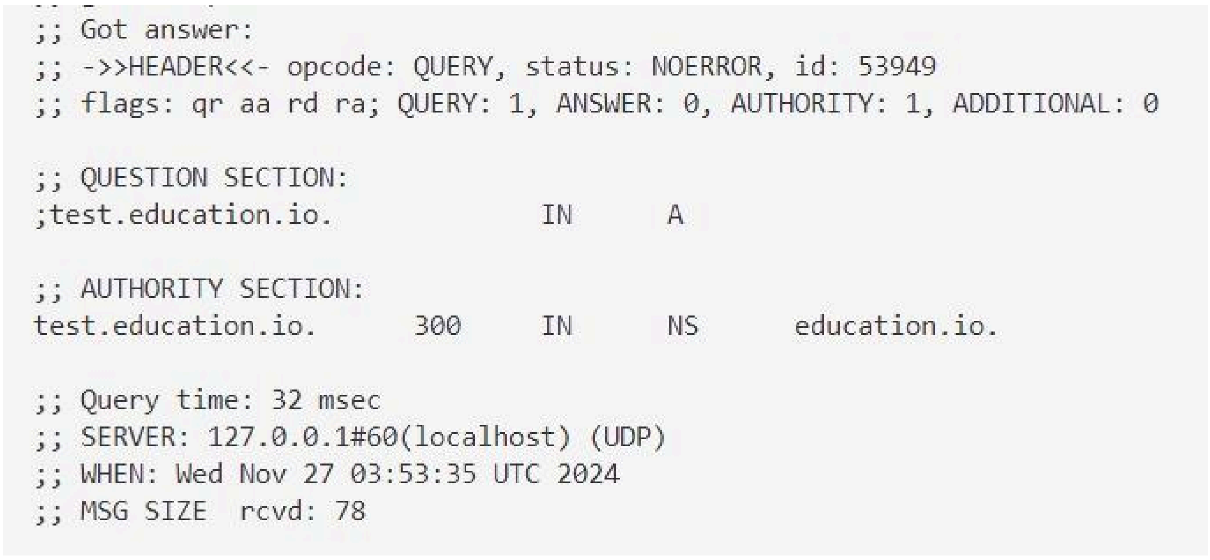
```
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 63097
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;api.portfolio.io.              IN      A

;; ANSWER SECTION:
api.portfolio.io.       300     IN      AAAA    5623:e377::

;; Query time: 53 msec
;; SERVER: 127.0.0.1#60(localhost) (UDP)
;; WHEN: Wed Nov 27 03:50:30 UTC 2024
;; MSG SIZE  rcvd: 78
```

For CNAME record



```
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 36646
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 0, ADDITIONAL: 1

;; QUESTION SECTION:
;auth.health.io.                        IN      A

;; ADDITIONAL SECTION:
auth.health.io.         300     IN      CNAME   startup.net.

;; Query time: 32 msec
;; SERVER: 127.0.0.1#60(localhost) (UDP)
;; WHEN: Wed Nov 27 03:52:47 UTC 2024
;; MSG SIZE  rcvd: 71
```

For NS record



```
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 53949
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 0

;; QUESTION SECTION:
;test.education.io.            IN      A

;; AUTHORITY SECTION:
test.education.io.      300     IN      NS      education.io.

;; Query time: 32 msec
;; SERVER: 127.0.0.1#60(localhost) (UDP)
;; WHEN: Wed Nov 27 03:53:35 UTC 2024
;; MSG SIZE  rcvd: 78
```

Summary and Explanation of the DNS Server Implementation

This set of code implements a custom **DNS server** that processes domain name queries, retrieves DNS records from a MongoDB database, and responds with appropriate information. It uses caching to optimize query responses and supports multiple DNS record types (e.g., A, NS, CNAME, AAAA). The implementation consists of several modular components, each performing a specific task.

Explanation of Each Component

1. Main Server Code (`index.js`)

This is the main script that creates a UDP-based DNS server. Below are its key functionalities:

1. **Setup:**
   - Imports necessary modules (`dgram` for UDP, `dns-packet` for DNS query parsing, `dotenv` for environment variables).
   - Connects to a MongoDB database using a function `connectDB` from `db/index.js`.
   - Sets up a cache object to store recently queried DNS records for faster response.
2. **getDNSRecord Function:**
   - Retrieves DNS records from the cache if available.

    - If not cached, queries the database (`recordModel`) to find the record and returns it.
3. **Handling DNS Queries:**
    - Decodes incoming DNS queries using the `dns-packet` library.
    - Checks the type of DNS record being queried and fetches the data from the database or cache.
    - Builds a response packet with appropriate flags and answers or returns an NXDOMAIN if the domain is not found.
4. **Supported Record Types:**
    - **A Record:** IPv4 address of a domain.
    - **AAAA Record:** IPv6 address of a domain.
    - **NS Record:** Name server for the domain.
    - **CNAME Record:** Canonical name for an alias.
5. **Response Encoding and Sending:**
    - Encodes the DNS response into a buffer and sends it back to the client using the `send` method of the UDP server.
6. **Error Handling:**
    - Logs errors encountered while querying the database or processing DNS requests.
    - Ensures the server continues to run in case of a minor error.

2. Database Connection Code (`db/index.js`)

This module handles the connection to the MongoDB database.

- **connectDB Function:**
    - Constructs the MongoDB connection URL using environment variables (`process.env.MONGODB_URL` and `DATABASE_NAME`).
    - Connects to the database using Mongoose and logs a success or failure message.

This modularity ensures the database connection logic is reusable across different scripts.

3. DNS Record Model (`models/record.model.js`)

Defines the schema and model for DNS records stored in MongoDB.

- **Schema Fields:**
    - `name`: The domain name (e.g., `example.com`).

    - `type`: The record type (e.g., `A`, `NS`, `CNAME`).
    - `target`: The value of the record, such as an IP address or alias.
    - `text`: Optional text associated with the record.
    - `priority`: Priority for MX records or other prioritized records.
- **recordModel:**
    - A Mongoose model to interact with the `Record` collection in the database.

4. Database Record Insertion (`test.js`)

This script is used to populate the MongoDB database with initial DNS records.

- **Generated Data:**
    - Contains predefined DNS records (e.g., IPv4/IPv6 addresses, aliases) in JSON format.
    - Examples:
        - `"name": "raj.com", "type": "A", "ip": "192.0.2.58"`
        - `"name": "verma.com", "type": "AAAA", "ip": "2001:db8::921f:5da1:..."`
    - Additional records can be added as needed.
- **insertRecords Function:**
    - Connects to the database and inserts the `generatedData` into the `Record` collection using `insertMany`.

Summary of Functionality

1. **Query Handling:** The server decodes DNS queries, checks for the requested domain in its cache or database, and sends a response with the matching DNS record.
2. **Caching:** Optimizes response times for frequently queried domains by avoiding repeated database lookups.
3. **Database-Driven Records:** Stores DNS records in MongoDB, allowing flexibility to add, update, or remove records dynamically.
4. **Error Handling:** Robust handling of errors during database queries or DNS processing to ensure server stability.
5. **Support for Multiple Record Types:** Handles IPv4 (A), IPv6 (AAAA), name servers (NS), and canonical names (CNAME).

Applications

- This DNS server can be used in small-scale environments to simulate or test DNS resolution.
- It demonstrates core concepts of DNS query-response cycles, database-driven backend services, and caching mechanisms.

Lab Report Notes

- **Key Concepts Demonstrated:**
  - Networking protocols (DNS query processing).
  - Database management (MongoDB integration with Node.js).
  - Performance optimization (cache implementation).
- **Challenges and Solutions:**
  - Handling multiple record types required detailed implementation logic.
  - Ensuring cache consistency and database synchronization.
  - Flag management for DNS responses.
- **Future Improvements:**
  - Add logging and monitoring for production-grade deployments.
  - Support additional DNS record types (e.g., MX, SRV).
  - Implement DNSSEC for enhanced security.

Conclusion

This project successfully implements a fully functional custom DNS server that integrates the fundamentals of networking protocols with backend database systems. The server handles DNS queries efficiently by decoding requests, retrieving the necessary records from a database or cache, and responding with the appropriate data. The modular design of the system ensures scalability, maintainability, and adaptability, making it suitable for educational and small-scale experimental purposes.

Through this implementation, several important technical concepts were explored and demonstrated:

1. **Networking Fundamentals:**
   - Understanding the DNS protocol and how it translates domain names into IP addresses.
   - Implementing a UDP-based server to handle real-time network requests and responses.
2. **Database Integration:**

    - Using MongoDB as the backend storage for DNS records, showcasing how databases can enhance dynamic and flexible data management in network services.
3. **Caching for Optimization:**
    - Implementing a caching mechanism to reduce latency and improve query response times for frequently accessed domains. This highlights the importance of performance optimization in real-world applications.
4. **Error Handling and Robustness:**
    - Incorporating mechanisms to handle malformed queries, database errors, and invalid requests ensures the server remains stable and responsive under various conditions.
5. **Support for Various Record Types:**
    - Implementing logic to process multiple DNS record types (A, NS, CNAME, AAAA) demonstrates a deep understanding of the DNS ecosystem and its diverse use cases.
6. **Dynamic Record Management:**
    - The ability to insert new records into the database dynamically paves the way for expanding the server's functionality, such as adding support for additional DNS record types like MX or TXT.

The project serves as a comprehensive example of combining network programming, database management, and software engineering principles. It bridges theoretical knowledge with practical application, providing hands-on experience in building a critical component of the internet infrastructure.

Reflections and Learning Outcomes

This project emphasized the importance of:

- **Protocol Understanding:** A strong grasp of the DNS protocol is essential to handle requests and responses accurately.
- **System Design:** Designing a modular, reusable architecture ensures that the server is easy to maintain and extend.
- **Performance Optimization:** Employing caching strategies demonstrated how computational and database overhead can be reduced.
- **Error Resilience:** Building systems that gracefully handle errors and edge cases is crucial for production-grade applications.

Moreover, the challenges encountered, such as handling complex query scenarios and ensuring the correctness of DNS responses, offered valuable insights into real-world problem-solving. Debugging database queries and fine-tuning caching logic further reinforced the importance of testing and iterative development in software projects.

Future Scope and Enhancements

This DNS server is a foundational implementation that can be expanded in multiple ways to make it more robust, secure, and feature-rich:

1. **Support for Additional DNS Features:**
   - Implement advanced record types like MX (mail exchange), SRV (service records), and TXT (text records).
   - Add support for wildcard DNS queries to handle subdomains dynamically.
2. **Security Improvements:**
   - Incorporate DNSSEC (DNS Security Extensions) to protect against DNS spoofing and ensure data integrity.
   - Implement rate limiting to prevent abuse and denial-of-service attacks.
3. **Performance Enhancements:**
   - Use an advanced caching library like Redis to improve performance for larger-scale deployments.
   - Add metrics and analytics for monitoring query patterns and optimizing server response times.
4. **Scalability:**
   - Implement clustering and load balancing to handle higher query volumes.
   - Use horizontal scaling by deploying multiple instances of the DNS server across distributed environments.
5. **User-Friendly Management:**
   - Develop an admin panel or API for managing DNS records, allowing users to add, update, or delete records without accessing the database directly.