GoodDNS: A Good DNS Server

November 30, 2023

Contents

1 Introduction

1.1 Overview

GoodDNS is a simple yet efficient DNS[5] (Domain Name System) server that can be used to resolve domain names to IP[8] (Internet Protocol) addresses. It is written in C# and utilizes only stock libraries. The server adheres strictly to the protocol as defined in RFC 1035[5]. This report will describe the implementation and development of GoodDNS, as well as the challenges faced during development.

1.2 Domain Name System

The Domain Name System[5] is a system described in RFC 1035[5] that is used to resolve domain names to IP addresses. The domain name system is hierarchical, each domain name server relies on higher level domain name

servers to resolve domain names. GoodDNS is a recursive DNS server, meaning that it will resolve domain names by querying other DNS servers. But it also supports defining authorative DNS records for lower level subdomains. This is a neat feature that allows the user to define custom DNS records for their own domain names.

1.3 DNS Records

The DNS[5] system uses DNS[5] records to store information about domain names. Here is a list of the most commonly used DNS records:

- **A**: Stores an IPv4 address
- **AAAA**: Stores an IPv6 address
- **CNAME**: Stores a canonical name for an alias
- **MX**: Stores a mail exchange record
- **NS**: Stores an authoritative name server
- **PTR**: Stores a pointer to a canonical name
- **SOA**: Stores the start of a zone of authority
- **SRV**: Stores service location
- **TXT**: Stores text information

Some noteworthy DNS records are the **A** record, which stores an IPv4 address, and the **AAAA** record, which stores an IPv6 address. These records are used to resolve domain names to IP addresses. The **NS** record is also noteworthy, as this is the record that is used to define authorative DNS servers on another domain name server. This is how the DNS system is hierarchical, each domain name server relies on higher level domain name servers to resolve domain names.

1.4 RFC 1035

RFC 1035[5] are the specifications that define how the Domain Name System operates. RFC stands for Request For Comments, and is a document that describes a standard or protocol. All commonly used protocols have an RFC document that describes how they operate.

1.5 UML

An UML[2] diagram is a diagram that describes the structure of a system. UML stands for Unified Modeling Language, and is a set of strict specifications that describe how UML diagrams should be structured. The specifications are designed by the Object Management Group[7] (OMG). UML diagrams are designed to be language agnostic, and can be used to describe the structure of any system. The UML diagram for GoodDNS can be found in *Figure 7*, this diagram describes the class relationships in GoodDNS. There are also many other types of UML diagrams, such as sequence diagrams, which describe the flow of a system.

1.6 SOLID Principles

The SOLID[10] priniciples is an acronym for five commonly adopted design principles in the design of object oriented software. The SOLID principles are:

- **S**ingle Responsibility Principle
- **O**pen/Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

The SOLID principles were first introduced by Robert C. Martin in his 2000 paper *Design Principles and Design Patterns*[9]. It is worth mentioning that the SOLID principles are not the only design principles that exist, but they were some of the first design principles to be introduced, and are still widely used today. Some competing design principles includes the CUPID[4], and GRASP[6] principles. This project loosely follows the SOLID principles as well as the CUPID[4] principles.

2 Domain Name System

This section will describe the Domain Name System and how it operates. We will also cover how parts of the Domain Name System are implemented in GoodDNS.

2.1 DNS Architecture

In RFC 1035[5] the Domain Name System is described as a hierarchical system of domain name servers. RFC 1035[5] provides us with a very detailed description of how the Domain Name System operates. The following are some illustrations of the Domain Name System architecture, as described in RFC 1035[5].

The domain name system consists of many domain name servers, each domain name server is responsible for a zone of authority. A zone of authority is a domain name, and all subdomains of that domain name. For example .com is a zone of authority, and example.com is a subdomain of .com. GoodDNS is designed to be the zone of authority for third level subdomains, such as example.com. This means that GoodDNS is responsible for resolving domain names such as www.example.com, but not example.com. This is because example.com is a second level subdomain, and GoodDNS is only responsible for third level subdomains see *Figure 1.*

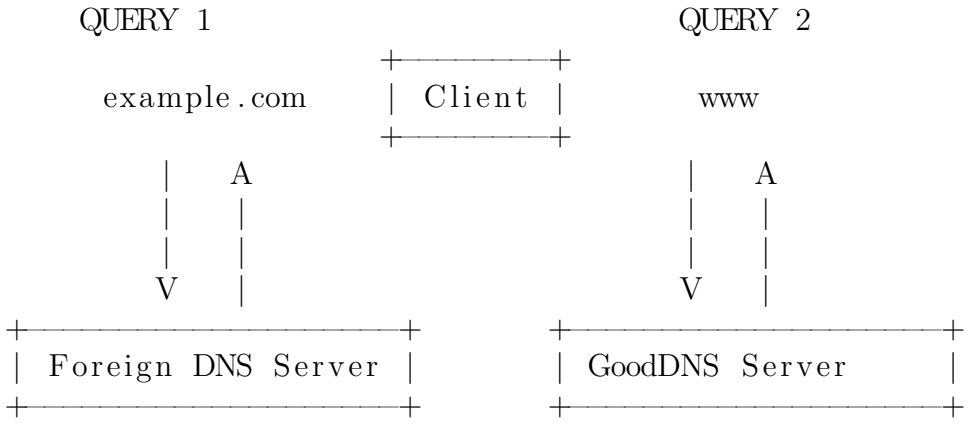

Figure 1: A foreign DNS server refering a client to GoodDNS

2.2 Recursive DNS

GoodDNS is a recursive DNS server, this means that if it is not the zone of authority for a domain name, it will query other DNS servers to resolve the domain name. This is what allows the DNS system to be hierarchical. This is also what allows GoodDNS to resolve domain names that it is not the zone

of authority for, by relaying the query to other DNS servers and caching the result. See *Figure 2*.

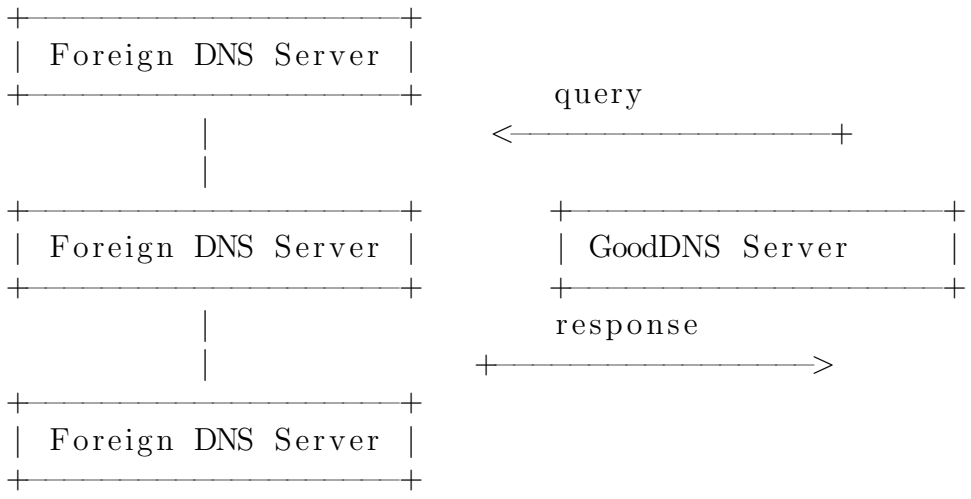

Figure 2: recursive DNS query from GoodDNS to foreign DNS servers

2.3 DNS packets

A DNS packet consists of a header, a question, an answer, an authority and an additional section. In this section we will describe the structure of the header, question and answer sections of a DNS packet. We are not going to cover the authority and additional sections, as these are not used by GoodDNS. see *Figure 3* for a quick overview of the DNS packet structure.

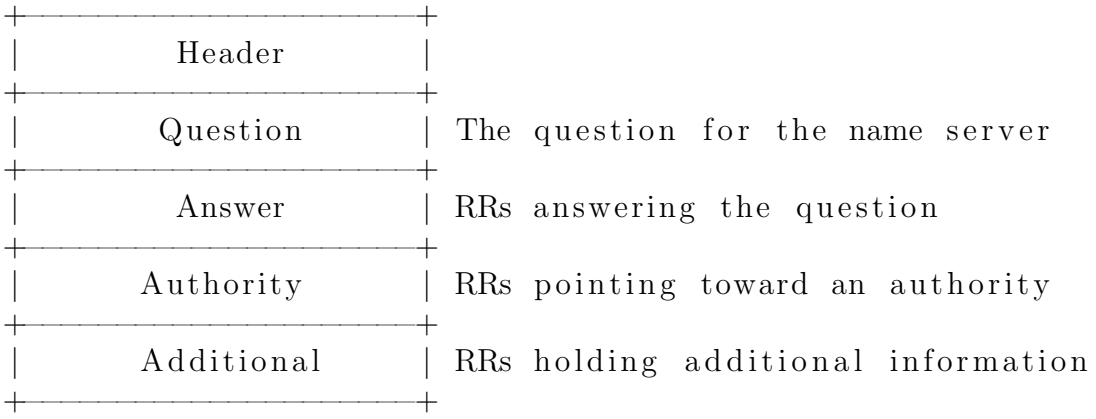

Figure 3: a quick overview of the DNS packet structure

2.4 DNS Header

The DNS header is the first section of a DNS packet. It contains the following fields as shown in *Figure 4*: **ID**: A 16 bit identifier assigned by the program that generates any kind of query. This identifier is copied the corresponding reply and can be used by the requester to match up replies to outstanding queries. **QR**: A one bit field that specifies whether this message is a query (0), or a response (1). **Opcode**: A four bit field that specifies kind of query in this message. **AA**: Authoritative Answer - this bit is valid in responses, and specifies that the responding name server is an authority for the domain name in question section. **TC**: TrunCation - specifies that this message was truncated due to length greater than that permitted on the transmission channel. **RD**: Recursion Desired - this bit may be set in a query and is copied into the response. If RD is set, it directs the name server to pursue the query recursively. **RA**: Recursion Available - this be is set or cleared in a response, and denotes whether recursive query support is available in the name server. **Z**: Reserved for future use. Must be zero in all queries and responses.
The rest of the fields represent the number of entries in the question, answer, authority and additional sections of the DNS packet.

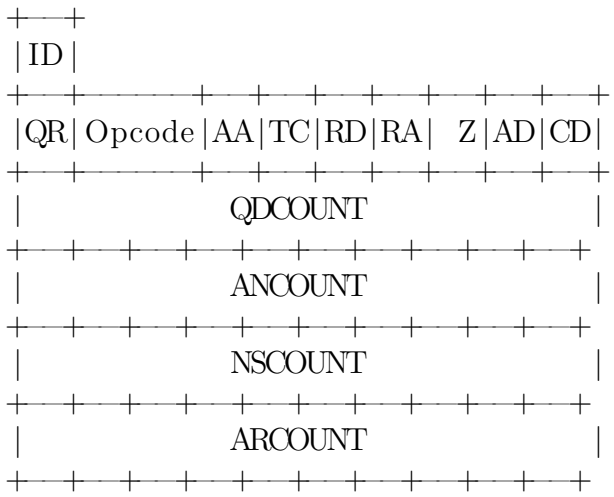

Figure 4: a quick overview of the DNS header structure

2.5 DNS Question

The DNS question is the second section of a DNS packet. It contains the following fields as shown in *Figure 4*: **QNAME**: a domain name represented as a sequence of labels, where each label consists of a length octet (4 bit) followed by that number of octets. **QTYPE**: a two octet code which specifies the type of the query. **QCLASS**: a two octet code that specifies the class of the query. see *Figure 5* for a quick overview of the DNS question structure.

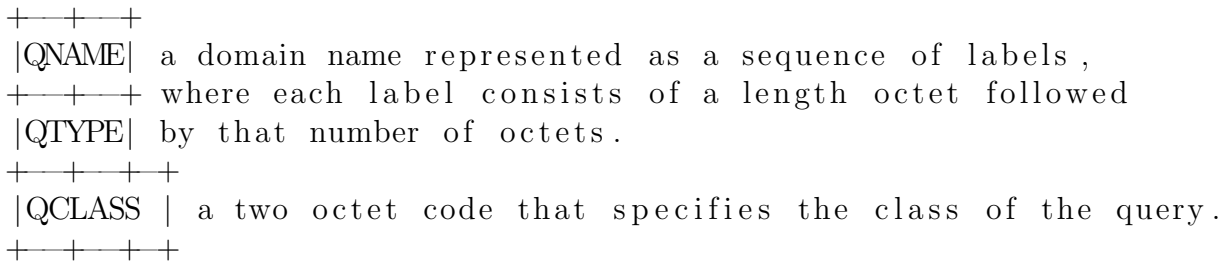

Figure 5: a quick overview of the DNS question structure

2.6 DNS Answer

The DNS answer is the third section of a DNS packet. It contains the following fields as shown in *Figure 6*: **NAME**: a domain name to which this resource record pertains. **TYPE**: two octets containing one of the RR (Resource Record) TYPE codes. **CLASS**: two octets which specify the class of the data in the RDATA field. **TTL**: a 32 bit unsigned integer that specifies the time interval that the resource record may be cached before the source of the information should again be consulted. **RDLENGTH**: an unsigned 16 bit integer that specifies the length in octets of the RDATA field. **RDATA**: a variable length string of octets that describes the resource. see *Figure 6* for a quick overview of the DNS answer structure.

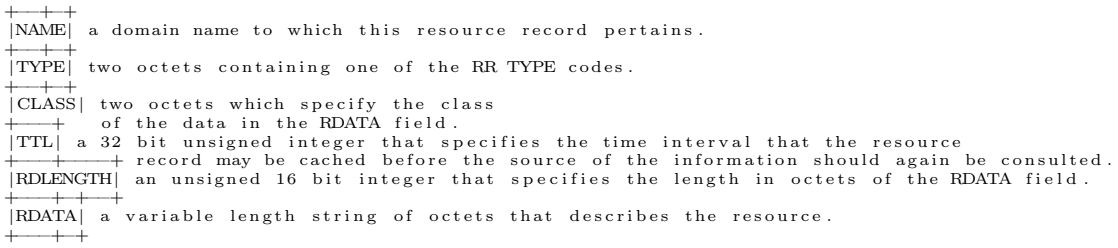

Figure 6: a quick overview of the DNS answer structure

3 Architecture

The general architecture of GoodDNS can be divided into three main components:

- Presentation Layer
- Application Layer
- Data Layer

3.1 Presentation Layer

The presentation layer is the layer that the user interacts with. In GoodDNS, the presentation layer consists of the settings class, and the logging class. The settings class is used to read settings specified by the user. Such as the port numbers for the TCP and UDP servers. The logging class is used to log information to the console and to the log file, such as when the server starts, or when a DNS query is received.

3.2 Application Layer

The application layer is the layer that contains the business logic of the application. In GoodDNS, the application layer consists of the following classes:

3.2.1 Packet

This class is used to represent a DNS packet. It contains methods for parsing and serializing DNS packets.

3.2.2 Question

When the packet parser reaches the first question in the DNS class, controll of the data pointer is passed to the question class.

3.2.3 Answers

When the packet parser reaches the first answer in the DNS class, controll of the data pointer is passed to the answer class.

3.2.4 Flagpole

The packet creates an instance of the flagpole class, and passes it all relevant flags. The flagpole class is used to store the flags of the DNS packet.

3.2.5 Server

This is an interface for the TCP and UDP servers. It contains methods for starting and stopping the server. Thus making the server logic protocol agnostic.

3.2.6 UDP Server

This class instanciates lots of UDP listeners, and creates new working threads for each user session.

3.2.7 TCP Server

This class instanciates lots of TCP listeners, and creates new working threads for each user session.

3.2.8 Universal Client

This class is used as a protocol agnostic interface for the packet recieved callback. Thus making the server logic protocol agnostic.

3.3 Data Layer

The data layer is responsible for caching, reading and recieving DNS records. In GoodDNS, the data layer consists of the following classes:

3.3.1 BIND

This class is used to read Zone[11] files. It contains methods for reading and parsing Zone files.

3.3.2 Cache

This class is used to cache DNS records. It contains methods for adding, removing and retrieving DNS records.

3.3.3 Record Requester

This class is creates and manages different transactions to other DNS servers. It contains methods for creating and managing transactions.

3.3.4 Transaction

The transaction contains the logic for sending and recieving DNS questions and answers. It is what allows the DNS server to be recursive.

The relationships between these classes can be seen in *Figure 7*.

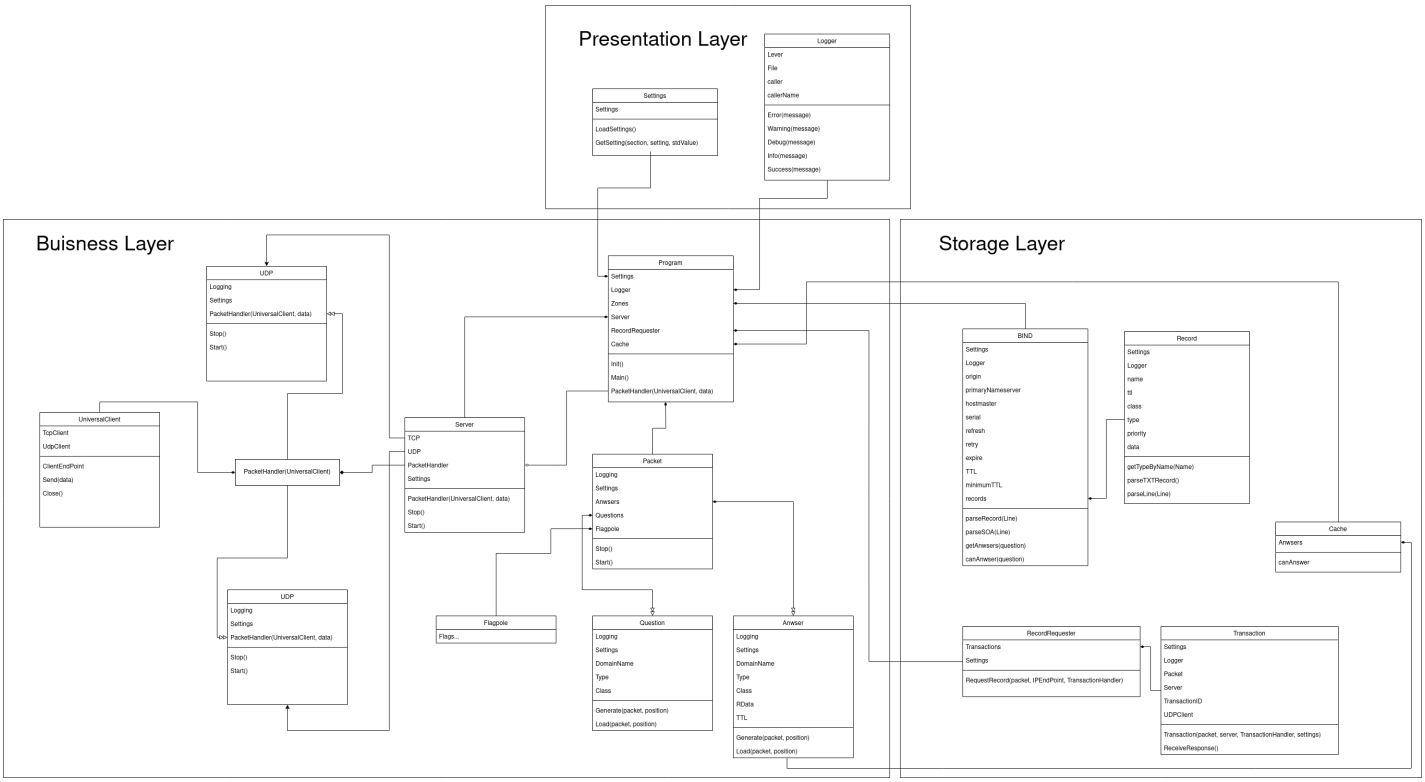

Figure 7: UML Diagram

3.4 The utilization of SOLID principles

All classes follow the single responsibility principle, and the open/closed principle. The adhereance to the single responsibility principle could be improved by abstracting the parsing and serializing of data into its own protocol agnostic template class, this would allow to easily define packet structures using a set of arbitrary instructions.

The program adheres to the Open Closed principle, this is achieved trough the callback system used by the TCP, UDP Server and Resource Requester. The program also theoretically adheres to the Liskov Substitution principle, but due to the program only implementing one byte protocol there is no need to inherit from abstract or allready existing classes.

The adhereance to the interface segregation principle can be seen in the use of the server interface, which is used to abstract the TCP and UDP servers. It can also be seen in the use of the Universal Client class, which is used to abstract the process of responding to a DNS query.

The dependency inversion principle is adhered to by the use of dependency injection. This allows for a loose coupling between the settings class and the rest of the program. The callbacks are also a sort of dependency injection which removes the higher level packet processing logic from the lower lever server logic.

3.5 Commonly used design patterns

The settings class is implemented following the singleton design pattern, a quite logical choice since there is no need for multiple instances of the settings class. The logger class does not adhere to the singleton design pattern, allowing different parts of the program to execute distinct levels of logging to different log files.

Within the program, there are also instances of factories, such as the question parsing methods within the packet class and the BIND (Zone file) parser, utilized for generating DNS records.

The program heavily relies on callbacks, this adheres to the observer design pattern. The combination of callbacks and lambda expressions enhances code readability and cleanliness.

Dependency injection is utilized to propagate the settings class throughout the program. Additionally, callbacks serve as a form of dependency injection, separating the higher-level packet processing logic from the lower-

level server logic. This modular approach contributes to a more maintainable and flexible codebase.

3.6 Event driven architecture

The programs is built around a simple yet solid event driven architecture, this allows the program to handle multiple clients simultaneously. The core mechanisme of this architecture is the creation of a working thread for each client. This contributes to the scalability of the program.

When a DNS query is received the program will dynamically initiate a new working thread, and pass the DNS query to this thread. This enables the program to handle multiple clients simultaneously. The working thread will then instansiates a new Universal Client, this client encapsulates all data needed to respond to the query, this is also what allows the rest of the program to be protocol agnostic.

The Universal Client will then be passed to a callback method, this callback method is what disconnects the higher level packet processing logic from the lower level server logic. The callback also provides the higher level processing its own working thread, this allows the program to handle multiple clients simultaneously. This is the core mechanisme of the event driven architecture. And this is what allows the program to scale so efficiently.

4 Development

4.1 Version Control

Version control systems are used to track changes in a set of files. This project relies on the version control system Git[1] to track changes in the source code. Git is a distributed version control system, meaning that each user has a local copy of the repository (A collection of files tracked by Git), and can make changes to this local copy. These changes can then be pushed to a Git server containing the main repository. This allows for multiple users to work on the same project

4.1.1 Branches

A branch is a copy of the repository. Branches allows us to simultaneously work on different features of the program without interfering with each other.

GoodDNS uses a system called feature branches, where each branch is named after the feature that is being implemented. When a feature is completed, the branch is merged into the main branch. Working versions of the main branch can be branched of to the release branch, which always contains a working version of the program. This system creates some challenges during major refactoring, as this will break all other branches. This is why major refactoring should be done on the main branch.

4.1.2 Testing

Integration testing is a method of testing different components of a system together. GoodDNS uses integration testing to test the following components:

- The UDP server
- The TCP server
- The BIND[11] class

The tests establish a DNS server, and then sends a DNS query to this server. The test then checks if the server responds with the correct DNS record. This is a very simple yet effective way of testing the system. The tests are configured to run automatically each time a commit is pushed to the Git server. More tests can easily be added due to the modular nature of the program.

4.2 Refactoring

The way the program handles the byte data found in the DNS queries were refactored multiple times during development. This area could still be improved like mentioned in the SOLID principles section. There were also a lot of styling changes during development, as the codebase was not very consistent in the beginning. The codebase is now much more consistent, and the adhereance to C# coding conventions[3] is more strict.

5 Usage

5.1 Settings

The server expects a configuration file named *settings.ini* to be present in the same directory as the executable. the settings file should look like this:

```
[logging]
path=./log.txt; the path to the log file.
logLevel=1; the log level, 0 is no logging, 1 is info logging,
;2 is debug logging... 5 is warning logging.
[server]
port=54321; this port is used for both TCP and UDP,
;note DNS usually uses port 53.
tcpThreads=10; the amount of threads allowed to be
;used by the TCP server.
udpThreads=10; the amount of threads allowed to be
;used by the UDP server.
[requester]
server=1.1.1.1; the DNS server to use for
;recursive DNS queries.
path=./zones; the path to the zone files.
```

All settings have a standard parameter and will run with the settings above if no settings file is found.

5.2 Zone files

The server expects zone files to be present in the directory specified by the *path* parameter in the settings file. A zone file should look like this:

```
$ORIGIN example.com.
@ SOA ns1.example.com.  (
    hostmaster.example.com. ; hostmaster email address
    2018080801 ; serial
    28800      ; refresh
    7200       ; retry
    864000     ; expire
    86400      ; minimum
)
www 60 A xxx.xxx.xxx.xxx
```

The zone file above defines the domain name *example.com* and the subdomain *www.example.com*.

References

[1] *About - Git*. https://git-scm.com/about. (Visited on 11/29/2023).

[2] *About the Unified Modeling Language Specification Version 2.5*. https://www.omg.org/spec/U (Visited on 11/29/2023).

[3] BillWagner. *.NET Documentation C# Coding Conventions - C#*. https://learn.microsoft.com us/dotnet/csharp/fundamentals/coding-style/coding-conventions. Aug. 2023. (Visited on 11/29/2023).

[4] *CUPID—for Joyful Coding*. https://dannorth.net/cupid-for-joyful-coding/. Feb. 2022. (Visited on 11/29/2023).

[5] *Domain Names - Implementation and Specification*. Request for Comments RFC 1035. Internet Engineering Task Force, Nov. 1987. DOI: 10.17487/RFC1035. (Visited on 11/29/2023).

[6] "GRASP (Object-Oriented Design)". In: *Wikipedia* (June 2023). (Visited on 11/29/2023).

[7] https://themeforest.net/user/dan_fisher. *OMG — Object Management Group*. https://www.omg.org/index.htm. (Visited on 11/29/2023).

[8] *Internet Protocol*. Request for Comments RFC 791. Internet Engineering Task Force, Sept. 1981. DOI: 10.17487/RFC0791. (Visited on 11/29/2023).

[9] Robert C Martin. "Design Principles and Design Patterns". In: (2000).

[10] "SOLID". In: *Wikipedia* (Nov. 2023). (Visited on 11/29/2023).

[11] "Zone File". In: *Wikipedia* (Aug. 2023). (Visited on 11/29/2023).