Details about the System

This section is focused on what other specifications could be useful to better know functionalities and limitations of the service.

I'll call slug the generated key to create a shortened url. For example:
http://www.longurltoshortened,com => http://cor.to/AA
AA is the slug

How the slug must be generated in case of the same long url is typed by the user?
The possibilities are 2:

- Generate a different slug
- Generate the same slug

For the first solution it doesn't need to check if the slug already exists for the long url. But the algorithm used, in case of hashing the long url for slug generation, will have to be based on the long url and another additional information in order to create different shortened urls for the same long url.
For the second solution, if the algorithm is based only on the long url, this is already guaranteed.

Is the custom slug functionality necessary?
If the user wants to insert a custom slug, a more complex system needs. Once the user inserts his slug, the service must check is the slug already exists and returns an error message to advise the user.

What happens on expired date if the user inserts an existed long url?
The problem appears only if we want to generate the same slug from the same long url. The expired date could be updated.
But if two users insert the same long urls, the second one updates the expired date of the shortened url that is related to both users. It's not good. Based on this last consideration, the best solution is to generate different slugs always.

If the system changes from anonymous to registered, things could be changed.

Could be useful to have the number of clicks for each shortened url?
From a point of view of the user this information determines the appeal of his shortened url.

Could be useful to check if the url exists?
In case, before the slug creation, it's necessary to check if the url exists, for example by pinging. But the user could want to generate a shortened url for his site before the site is online.

What happens if an attack creates many redirection calls or many creation requests?

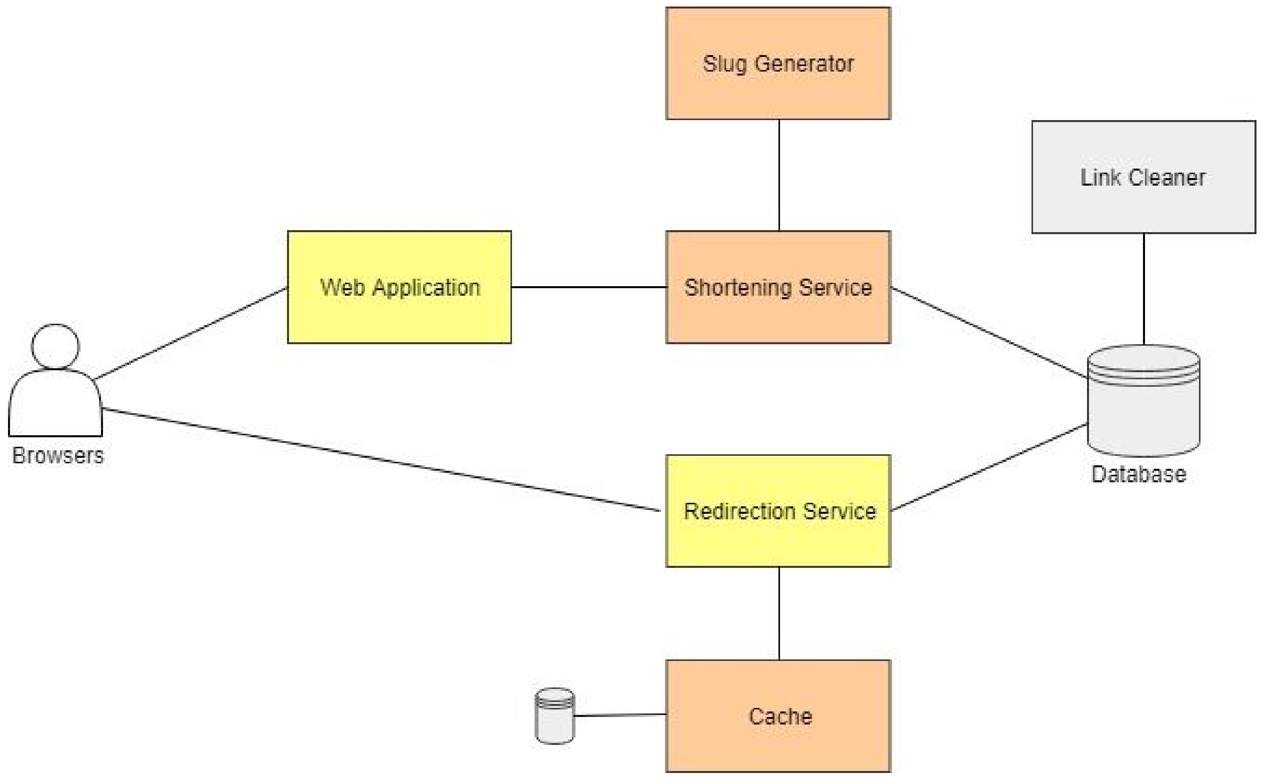
It’s possible to check the number of calls per second and block the ip from which the calls come, if the number is greater than a threshold.

Architecture

The architecture for the shortening service is distributed using the cloud.
In the following there is the list of all components in the system. The details for each one will be explain after.

- **Web Application**: it’s the web application for creating shortened urls by the user.
- **Web Services**
    - **Shortening Service**: it’s the web service that exposes the resource to create a new shortened url
    - **Redirection Service**: it’s the web service that exposes the resource to redirect from shortened url to long url
    - **Slug Generator**: it generates the slug to compose the shortened url from long url
- **Serverless**
    - **Link Cleaner**: it deletes shortened urls expired
- **Cache**: it manages the most used calls during redirection process, in order to avoid the database access
- **Database**: where all long url are stored together to its shortened url and additional useful information

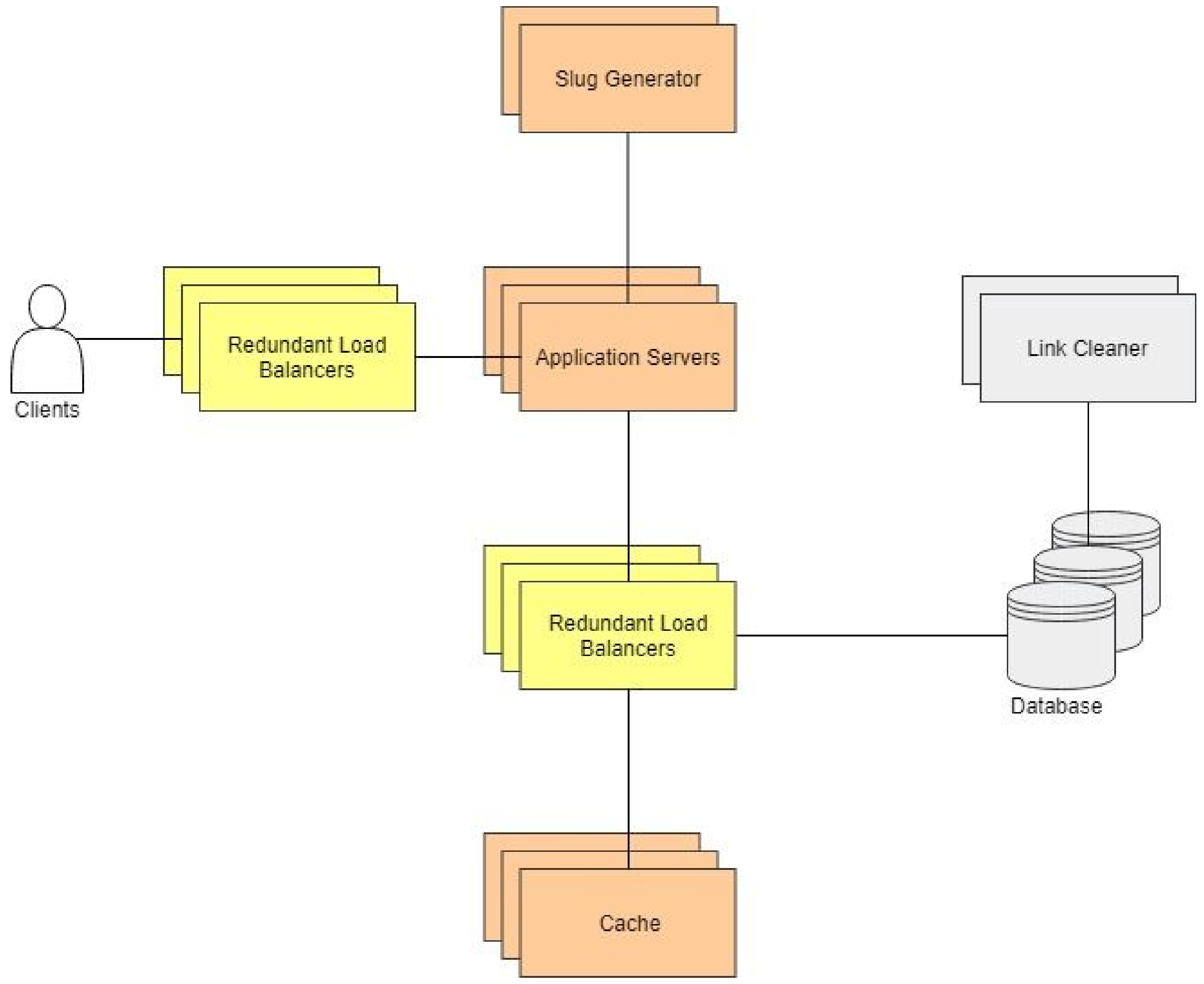
The following schema shows the interaction of the components



In the schema above it’s clear the **separation between Shorten Service and Redirection Service**. The reason is because the system is a read-heavy system. The difference, in term of load and performance, between creating shortened url and redirecting operation is very relevant. The redirection from shortened url to long url must be really fast. On the other side, the generation of a shortened url can be slower. The **Cache** component is useful to have **less latency** possible.
In addition the separation between two services allows to avoid the crash of all system if one service goes down. This aspect is related to the **Graceful Service Degradation**.
The cache has its own persistence in order to avoid the lost of data in case of crash.

Distribution

The following schema shows the distribution of the components in order to increase the stability of the system.



There are **multiple instances of the application servers**, managed by redundant load balancers, in order to distribute the network traffic load. The **load balancers are redundant** to remove a single point of failure.
The load balancer must also **skip from routing unhealthy instances**, if there are.

The **Slug Generator** is important but not crucial in term of performance for the system. Eliminate the single point of failure through a **failover cluster** is enough.

The same reasoning could be made for the **Link Cleaner**, also for it a **failover cluster** can be used.

For the **database** the **sharding method** could be applied for scaling out on multiple instances.

Resilience Software Design

Distribution systems implies critical points at the boundaries of theirs components. Besides bugs or bad behavior of each component, there are other problems related to communication among components.
One principle at the base of distributed systems is to design the architecture thinking to failures. The management of failures during communication becomes important.
To handle this kind of problems, the **Resilience Software Design** (RSD) is a must. The concept behind the RSD is that during the using of an application, the user does not notice if failure occurs or at least the user can continue to use the application with a defined reduced functional scope (**Graceful Degradation** of service).
For example, in order to avoid Cascading Failures caused by coupling of services, the architecture could be based on a service bus. In this way instead of a failure of the system caused by a call from service A to Service B that is down, a message is put in the bus and sent when the service B goes up again. The user will see a slower operation, but not an error of the system.

Based on the above concepts, there are two critical points in the system.

- The generation of the slug, made by a dedicated service
- the access to the cache during redirection.

In both case it’s not acceptable that the calling service waits for the answer of the called service and if the called service goes down the calling service goes down as well.
In order to avoid the cascading failure a **message queue** is useful.

The Shorten Service publish a message on the message queue and the Slug Generator listens for new messages and generate the new slug when a specific message arrived. It will publish a message to send the slug. The Shorten Service listens for messages and receive the slug when the message arrived.
In this way if the Slug Generator is down, the Shorten Service doesn’t go down as well. After the failover process the Shorten Service receives the message for new slug.
The user doesn’t receive an error message, but he maybe will notice a slowdown of the system.

For the cache the concept is the same, but in this case the async communication through a queue is already present in most caches (for example Redis).

Note
Before to choose an async or sync communication, it’s necessary to **balance costs and benefits**. Surely a system based on a message queue is more robust, but also more sophisticated and complex.
To balance costs and benefits are also crucial all redundant parts of the system (load balancers, multi instances, replication, etc.).

Deploy or Change Configuration

This is a critical point and often can cause an outage of some part of the system.
To solve this problem a load balancer is useful, in order to update only a subset of instances and point to updated instances.
If the system evolves, for example from anonymous to registered, it could be useful to test the new functionalities on different instances for each new and updated parts of the system (database, services, web application etc.).

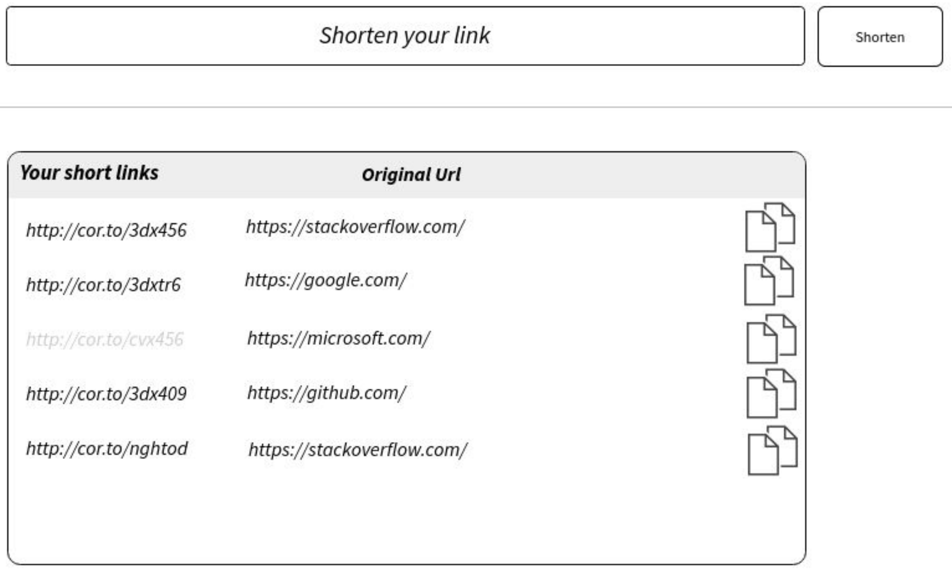
Components

In the following sections all components in the system are described.
In order to give some practical examples in the following list there are the technologies that can be used to develop the system:

- Web Application: Angular
- Web Services: Aspnet Core
- Cache: Redis
- Serverless: Azure Functions
- Database: MongoDB

Web Application

The client application to generate shortened url is relatively easy. It’s important that **the user can see what he can immediately do**. The textbox must be the most important object in the page, it must be in the center and big enough. Through the Shorten button or pressing Return key the new shortened url appears in the list as first element.



Your short links	Original Url
http://cor.to/3dx456	https://stackoverflow.com/
http://cor.to/3dxtr6	https://google.com/
http://cor.to/cvx456	https://microsoft.com/
http://cor.to/3dx409	https://github.com/
http://cor.to/nghtod	https://stackoverflow.com/

The list of shortened urls come under the textbox. The functions are the copy button and the original url.
Checks on input long url can be done on the client (and again on the server). Through regular expression is possible to validate the right format of the long url.
In order to see the list of shortened urls, it’s necessary that cookies are enabled.

The list must related to the browser, because the access is anonymous. The web application adds in a cookie a unique identifier and during the slug generation it will be stored in the database together to long and shortened url. At next accesses, the site will call the server to get the list of information related to the browser based on the unique identifier and populate the list.

More in the cut

To develop the client side application for shortened url generation, the choice could be Angular + Ngrx.
In order to manage an application based on Redux pattern the basic thing is to define the **state of the application**

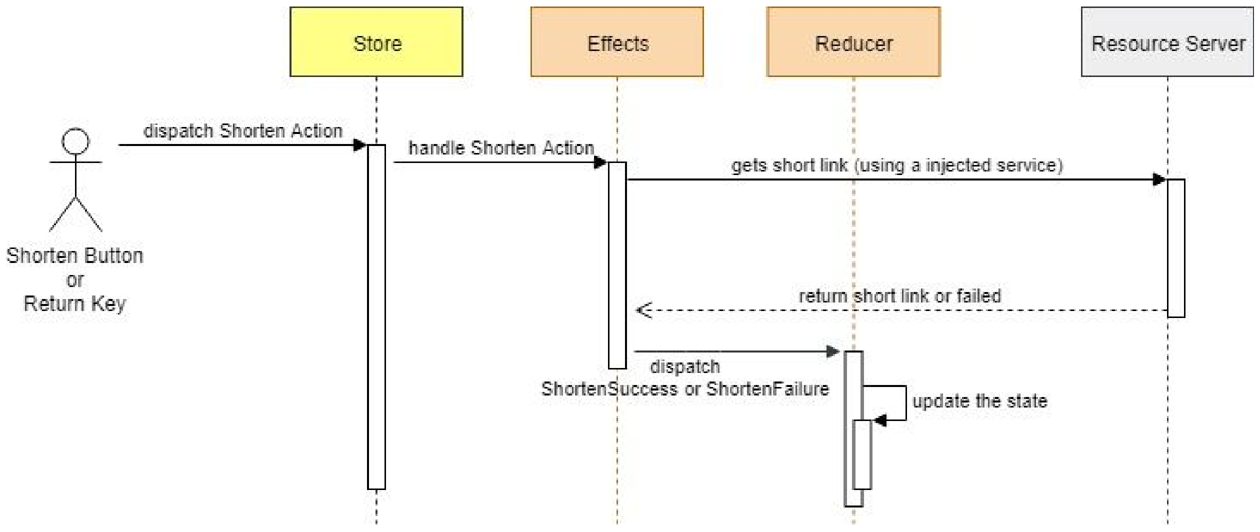
```
UrlInfo
        longUrl: string
        shortUrl: string
        selected: boolean

AppState
        busy: boolean
        errorMessage: string
        urls: UrlInfo[]
```

The **actions handled by reducers or effects** are:

- Shorten
- ShortenSuccess
- ShortenFailure
- GetUrls
- GetUrlsSuccess
- GetUrlsFailure
- UpdateUrls
- CopyToClipboard
- SelectUrl

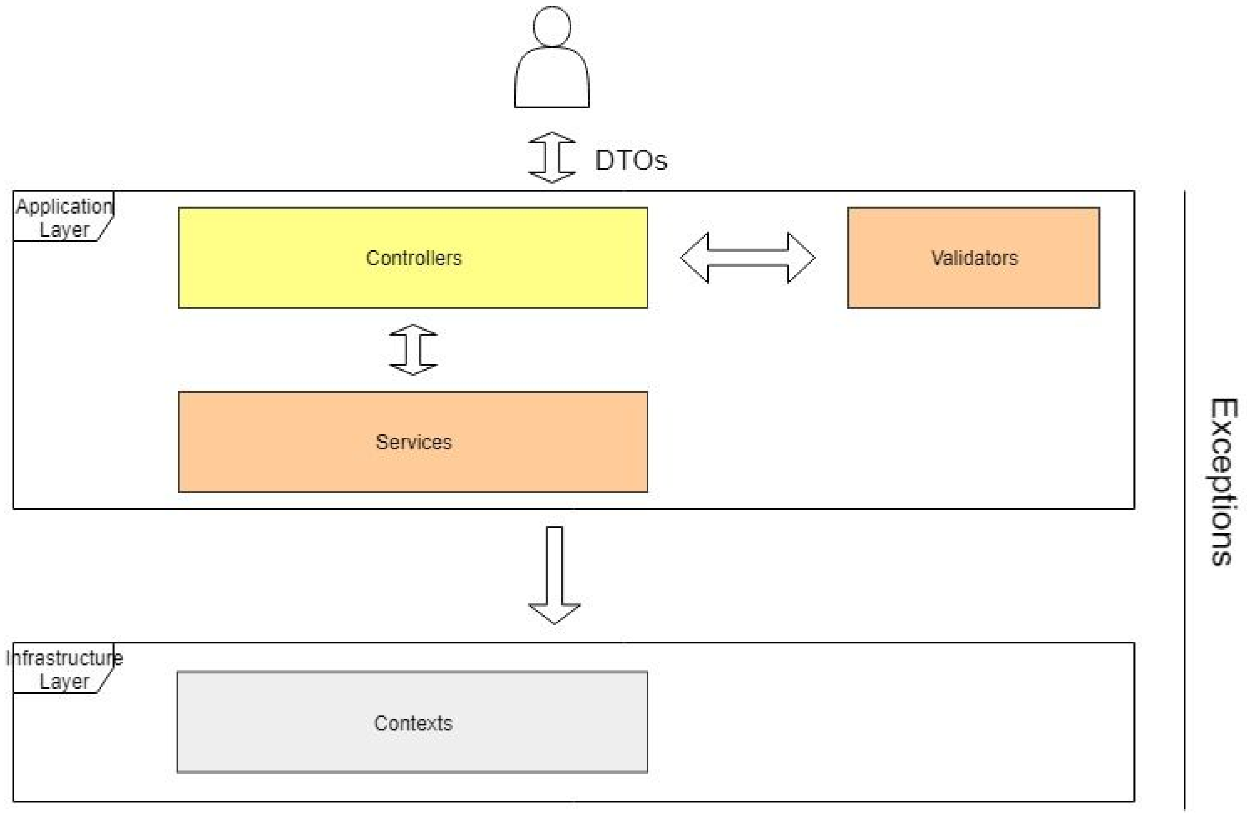
The following picture shows the sequence diagram related to press Shorten button (or press Return) by the user, handled by Ngrx



The architecture (the using of redux pattern) could seem complex for an easy application as this one, but the pattern allows the application to grow up in future without changes in the infrastructure.

Web Services

The following schema shows the layers on the server side



All web services in the system are relatively simple in terms of business rules.
Shorten Service must return a shortened url by a long url.

Redirection Service must redirect to long url from a shorten url.
Slug Generator must give a unique slug from a long url

In this case, the DDD approach seems to be an over architecture.
The only not linear things are:
- the calling of the Shorten Service to Slug Generator through a message queue
- the calling of the Redirection Service to Cache

A dedicated services in the application layer could be enough for these situations.

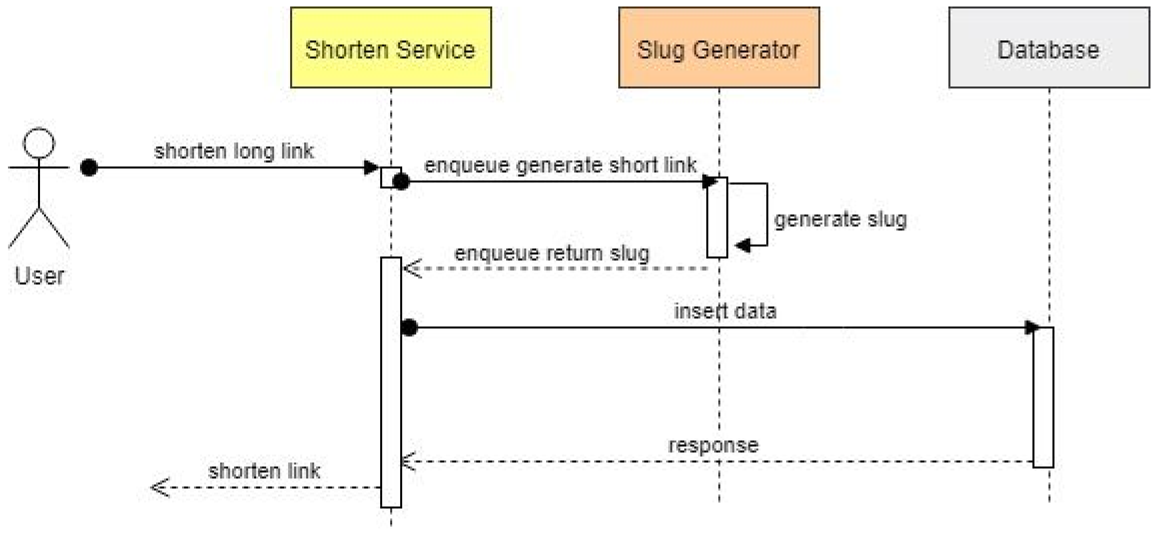
For the first case, it's necessary an assembler injected in the service to convert DTOs to message objects in queue and vice versa.
For the second situation the service must handle the Cache, querying and updating it.

The access to the database is made directly without passing through entities, working with json format.

The exception handling is made through an exception middleware inserted in the middleware pipeline of Asp.Net Core.
Every requests pass through the middleware pipeline from the first to the last middleware. Every responses pass through the same pipeline from the last to the first.

Shortening Service

The following picture shows the sequence diagram of the shortened url generation



The above schema is based on the Slug Generator generates always different slugs.
A sophisticated solution is to have a Slug Generator with a database with a enough numbers of pre-created unique slugs. Marking the used slugs ensures that a returned slug is unique.

In case to choose a different approach, to avoid slug duplication, before insert the data in the database, a check needs and in case the slug is already present, the service can call the Slug Generator again. An additional parameter could be used to generate a different slug than before. The process is done until a generated slug is unique.

In order to avoid cascading failure the communication between Shorten Service and Slug Generator is made through a message queue.

The only APIs for the basic functionality of the service for generation of shortened url are:

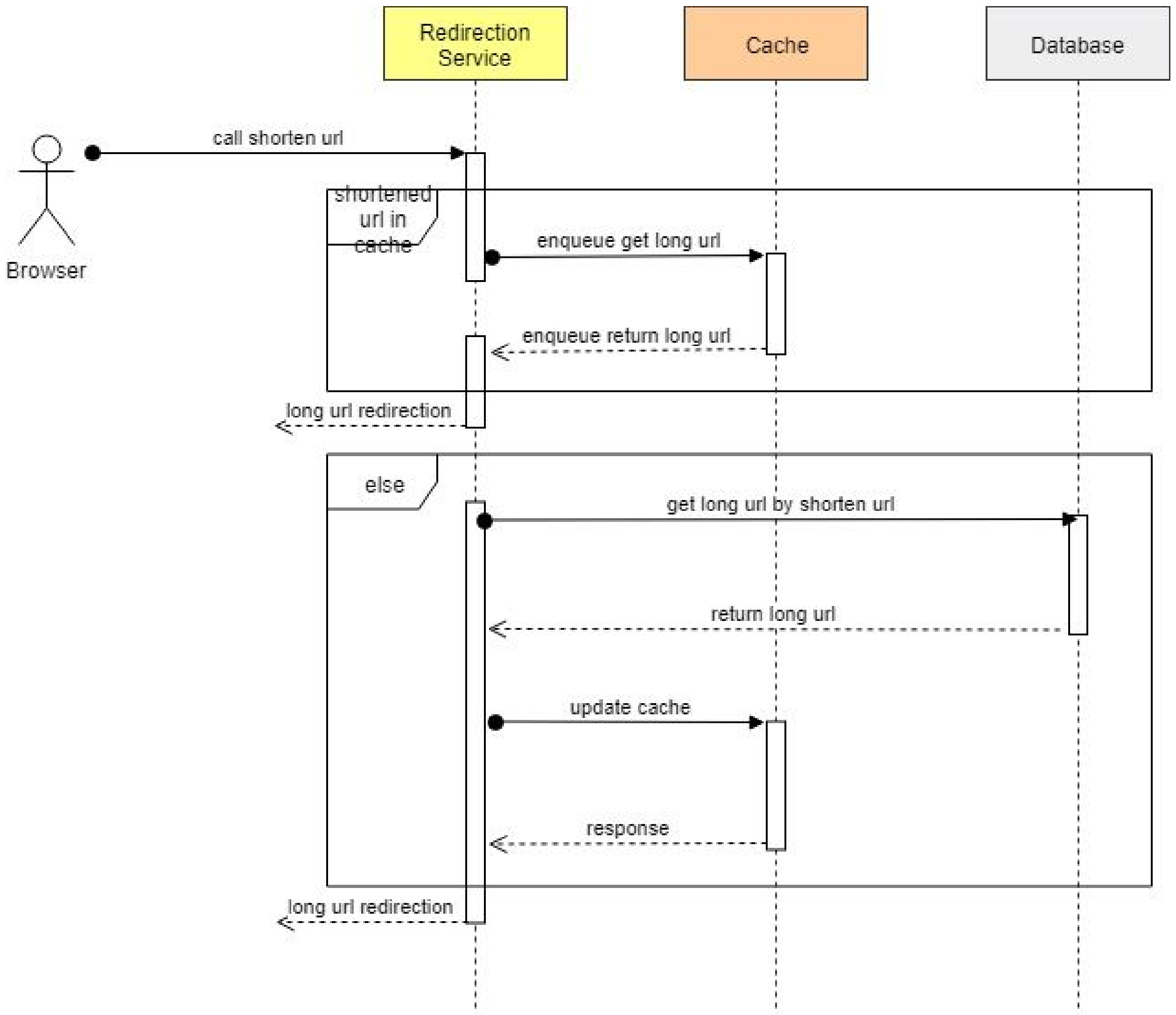
ShortenController
> *string Put(string longUrl) // Returns a shortened url from a long url*
> *List<UrlInfo> Get(string browserId) // Returns the list of url associated to the browser*

At the following endpoints:
http://cor.to/shorten PUT
http://cor.to/shorten GET

Once an action on the service is called the first thing to do is checking and validating of inputs. The format for longUrl has to be a valid url, otherwise an exception has to be thrown. Checking the presence of the browserId in the database otherwise an empty list must be returned.

Redirection Service

The following picture shows the sequence diagram of the long url redirection



RedirectionController

string Put(string shortenedUrl) // Redirects to long url from shortened url

At the following endpoint:
http://cor.to/redirection PUT

The status Code can be:

- 404 Not Found, in case of expired date or not present in the database
- 302 Found, in case of redirection is possible

A system Cache as Redis allows to use a queue to exchange messages. This native mechanism avoids the cascading failure.

Slug Generator

The generation of the slug is delegated to a specific service. In this way it will possible, for example, to change the algorithm and test separately the functionality.
This service can be implemented in two different ways:

- Generates a slug from the long url, through an algorithm for hashing based on long url input and other info to ensure the uniqueness of the generated slugs.
- Pre-generates unique slugs and stores them in a dedicated database and mark the used slugs.

In case of pre-generated slugs, it's important to clean the database associated to Slug Generator based on the value of the expired date. The cleaning is made by the Link Cleaner component.

Cache

One on the most heavy operation is the access to the database. If it could be irrelevant during the creation of shortened url, but it becomes a big problem during the redirect operation.
In order to reduce the latency, a cache service is useful.
Redis is one of the Cache System that it's possible to use. It has the possibility to persist key/value pairs and has a default **LRU (Least Recently Used) policy**. The policy could be changed, monitoring the number of accesses to the shortened urls and maintaining in cache the most used ones. For that an additional field (a click counter) in the database must be added or maybe it possible to use monitoring tool of the cloud.

This component is really critical and could be useful to have more Cache Service, for example one for each server running the Redirection Service.

As shown in Redirection Service sequence diagram, the Redirection Service knows both the Cache Service and the database. It's possible to delete the dependency between Redirection Service and the database, wrapping the access to the database and Redis.

Serverless

Performing a scheduled task, as cleaning up entries of a database, in the cloud could be done with functions. For example, in Azure it's possible to do that thanks to the Serverless Framework.
The Link Cleaner, for cleaning up the entries out of date is an example in which a cloud function is useful.

Link Cleaner

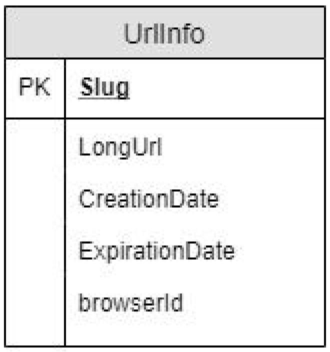
The cleaning up of expired shortened urls is done in two phases.

If a shortened urls is used to redirect to a long url and the shortened url is expired, the shortened url must be delete from the database. This activity is made during the normal user flow without an additional service.
But most of expired shortened urls will remain in the database.
In order to avoid the database growth, a dedicated service checks and deletes expired shortened url.
This service can't run always or everytime, but it must be scheduled during low traffic.
A typical implementation of this job could be made through a serverless.
This component is not critical for the system, but surely keeps clean the database.

In case of the Slug Generator has a own database with pre-generated slugs, the Link Cleaner must update also that, unmarking the expired slugs.

Database

The minimum fields to store in the database are the following



The slug value generated by the Slug Generator is the primary key.
The browserId field is used to get the list of all shortened urls generated by a browser.

There are no relationships with other tables so a NoSql db in this case could be better than a relational one. One possible database is MongoDB.

To handle the growing of stored data is possible to distribute data across multiple different servers. MongoDB using the horizontal scalability to achieve this, through the sharding method.

It's important to manage who can enter to the database and in which way:

- only read for redirection in case the deletion of expired shortened url is made only by Link Cleaner. Also write is the cleaning is made also during a redirection request if the url is expired.
- read and write for shortening

The access to the database could be wrap through a Web API. In this way the services don't know the database directly. The connection string is known only by the wrapper.

Moreover the concurrency is not a problem:

- The redirection is a read-only operation. The multiple access is not a problem.
- In case of generating new slugs for every shorten operation, also the insert operation is not a problem in terms of concurrency.

The only problem appears when different users request a redirection to a shortened url that is expired and the system goes to delete the entry. To avoid this the task to delete the expired entries can be left only to the Link Cleaner.

Testing

During the development of a distributed system there are more layers and more kinds of tests that must be done.

TDD

In order to avoid many mistakes during writing code, the TDD approach is the best choice to develop software.
Writing tests before code, helps to focus on the specific function that you want to develop in that moment.
There are many frameworks for all languages for doing tests.
On the client side, for Angular Jasmine and Karma are a must.
On the server side, for .NET technology XUnit is a choice.

Load and Performance on Service

When the server side of the system is ready, it becomes important to check the services stressing them with load and performance tests. A useful tool is JMeter that allows to simulate calls to services, specifying for example the numbers of calls in a time, users connected and other parameters.

Database

There are many tools to test the database.
In case of MongoDB, on of the choice is MongoDB Orchestration that is an HTTP server with a RESTful interface to MongoDB.

Chaos Testing

In a distributed systems other kind of tests are necessary in order to check the health of the infrastructure.
The Chaos Testing is a kind of test to add chaos in a group of systems, terminating one of the system in the group randomly. Tools for that are for example Chaos Monkey for AWS, Search Chaos Monkey for Azure or Kube Monkey in case of Kubernetes.

In order to be sure that the system is health, tests are necessary continually against common issues to make sure that all the system survives various failures.