URLShortener Documentation

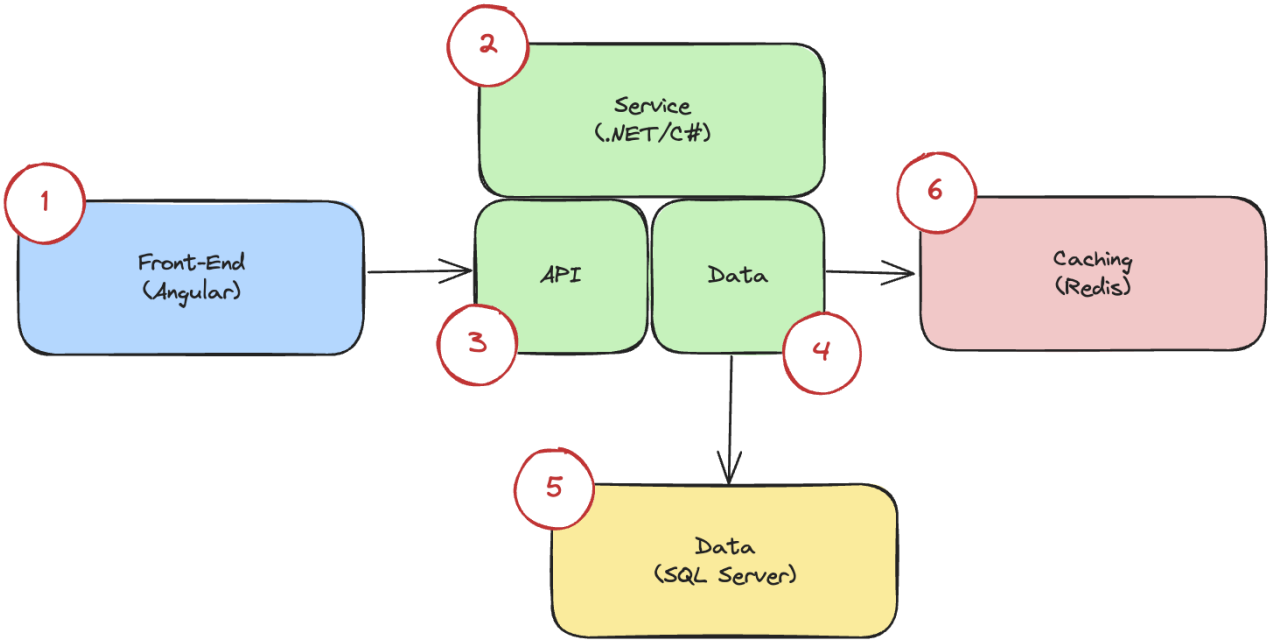
Introduction

URL Shortening service operates in the basis that a short domain name is owned by the URL Shortening service provider. The full URL is mapped to a shortened name and returned back to the user. When a client (eg: Web Browser) request the short-form of the URL, a HTTP re-direct is sent back to the client with the full-URL where the user is taken to the full URL based web-page.

1. The service provider owns the `short.en` domain and the DNS setup is already completed to point to the application service described in this document.
2. This service is used by a single user for the purpose of this exercise and no security, user control needed to be implemented in this exercise.
3. General caching (Redis discussed here, however a memory cache too should be sufficient)

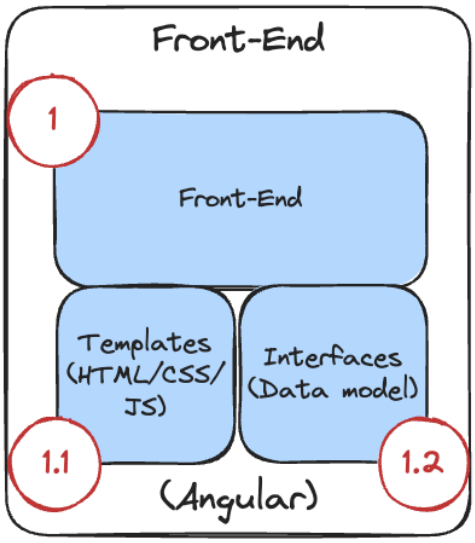
Approach

Designing and implementing a URL shortener service would involves several critical considerations and design choices, including scalability, performance, security, and user experience.



1. **Frontend**: Angular, modularised UI that maps the routes to the backend service. This component will allow the user to enter a full URL and receive a shortened URL
2. **Service**: DotNet service that will encompass two modules for API and Data
3. **API**: Service that would expose a RESTful service to the user interface. This will interact with the data module for data persistence needs. Also, when user sends a shortened URL, a redirect response to the full-URL is sent by this component.
4. **Data**: The caching and Object Relational Mapping service that would persist the data across the life of the service. This will optionally interact with the caching layer so that database calls can be minimised for frequent reads for the similar objects

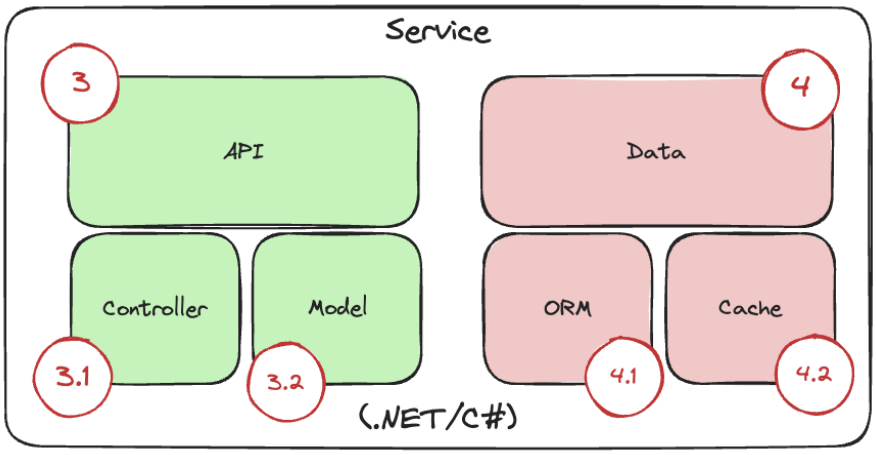
Front-end



A modular (component oriented) user interface is proposed for the ease of upgrading and management while providing a greater user experience and high performance.

1.1. The templates include the Javascript, HTML and CSS resources where the action (API calls to the backend) presentation (HTML for structure, CSS for branding/formatting)
1.2. Interfaces are the JSON object definitions (Typescript based) that would be passed between the UI and the API.

Service



API

3.1. Controller would expose the API end-points (including the Swagger documentation URLs and RESTful API
3.2. Model would hold the data binding from the user interface to the data later integration

Data

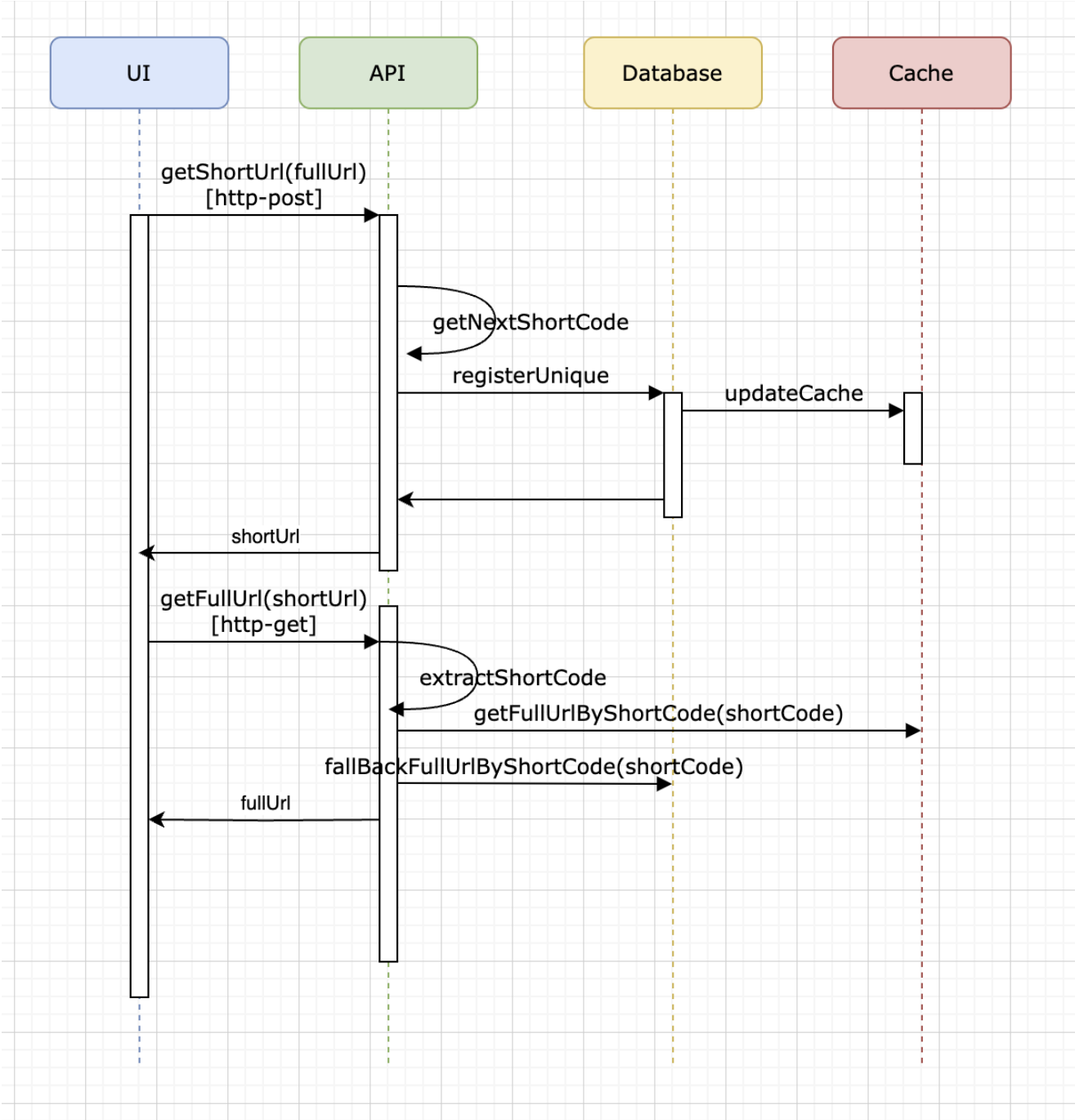
4.1. Object Relational mapping will have the classes for the business layer objects to interact with the database layer
4.2. Caching layer would have the classes for the business objects to interact with the Redis cache

NOTE: This cache will be the first call to receive the data on the read mode, and when not found, fall-back on the database, then refresh the cache.

UML Sequence Diagram

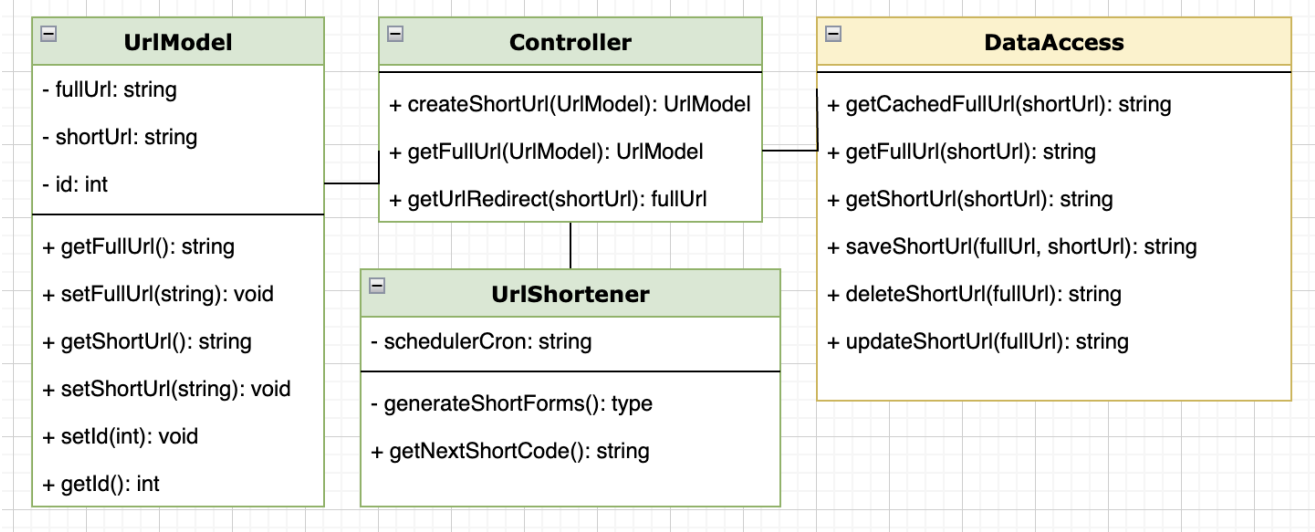
Initially, the UI will let the user enter a full URL, and send the payload (full URL to the API service). API service will validate the URL, and then find the next available short code from a list of available (Short form of the URLs are pre-generated so that the uniqueness of the URLs are maintained and run time overhead is reduced). API then will substitute the full URL to the short form and return to the UI.

On the request for the full URL, the short form of the URL will be looked up in the cache first, and then on the database. The full URL is returned by the API with a HTTP 302 (temporary re-direct, because if the user decided to change, this short URL should not be cached).



Class Diagram

URLModel is used to transport the URL mapping between the HTTP API Interface and the Controller, by the controller. The controller exposes the url-shortening API (for the front end) which lets the user create a short-form and retrieve the short form URL for a given full URL. Also given a short-URL, returns full URL retrieval with HTTP 302 redirect for the browser redirection.

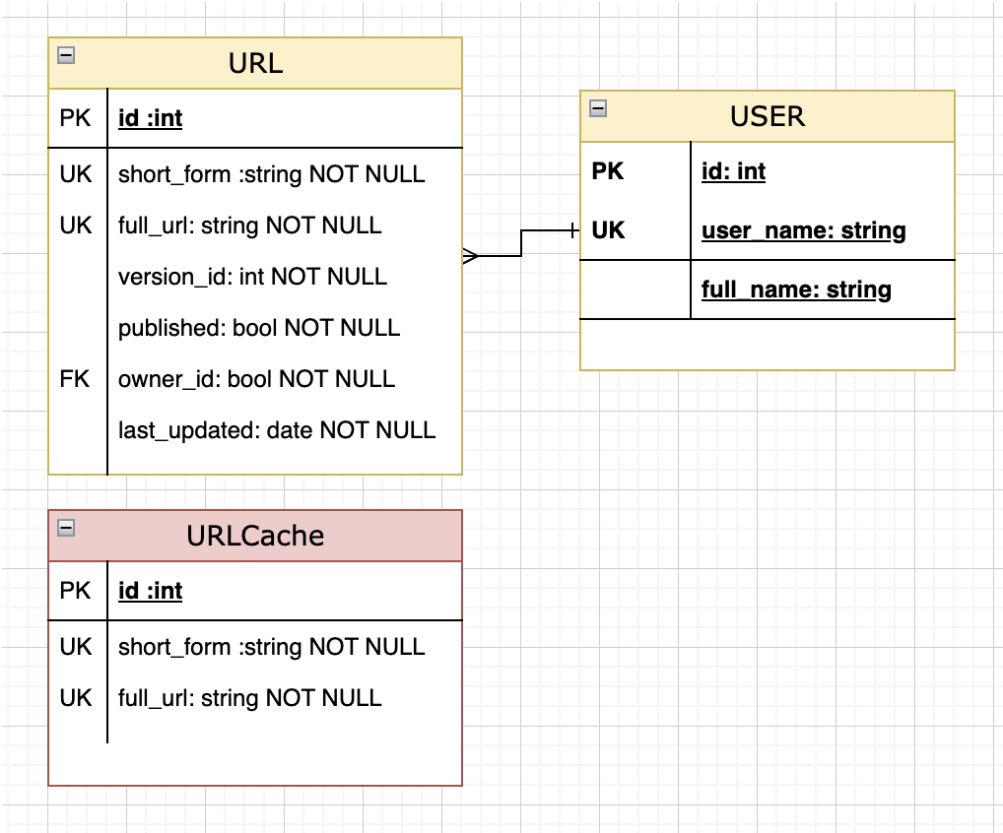


Assumption:

- No need to see the full list of currently available URLs. If that's required, another method in the controller, data access is required to return all the currently available URL mappings in the database. (ie: those having a full-url in the database)

ERD

Entity relationship diagram below discuss how the database and the cache is specified for the data-storage.



Database Table: URL

The shortForm is a pre-generated random, short, string value (eg: `sDyeWa`, `sYwPow`). Full URL can be assigned or removed for the short-form so that the short form URL can be re-used. The last_updated allows to have a grace period if the short form should not be re-used within a given time after un-assigning from a mail URL.

1. Primary Key is `id`
2. A Unique Key is set for `short_form` (shortened part of the url - eg: http://short.en/dYskqF url's `dYskqF` is stored in this field) so that shortened URL's uniqueness is maintained in the DB level
3. A given full URL should have only a single shortened URL. Hence, the `full_url` too is a Unique Key field.
4. `version_id` is to support the concurrency (not implemented in this exercise, however, version pattern can be applied to avoid dirty writes in the application level, preventing safety in a multi-user environment)
5. `published` holds the state whether the URL mapping is already in use or not. This allows a future enhance for the user to first list all the URLs and let them publish on a future date/time.
6. `last_updated` can be used for a scenario where there needs to be a grace period, such that a given short-form URL should not be re-used after its un-publishing until a given time window is elapsed (helps avoiding user confusion, and HTTP caching).

Database Table: USER

Future enhancement when a multi-user environment is introduced, the logged in user can be aligned to this table's username (along with a IDM id field) so that the `id` in this table can be mapped to the `owner_id` in the URL table. This results in URLs saving by multiple users.

URLCache

This is a store in the Redis cache so that the multi-reads can be high-performant. Premise is that the URL Shortening service is used to write once and read extremely high amounts of times. `id` is a unique record in the URLCache record store (in-built functionality can be used). The `short_form` is the Unique key (Key in the key-value store) and the `full_url` is the Value of that record.

Configuration

Configuration elements for the application.

1. Short URL Prefix: `short_form_prefix`
2. Server Port: `server_port`

Code smells

- **Long Methods**: Methods that are too long and handle multiple responsibilities can be hard to read, maintain, and test. This violates the Single Responsibility Principle (SRP).
- **Duplicated Code**: Copying and pasting code across the application increases the likelihood of bugs and makes future changes more difficult.
- **Large Classes**: Classes that do too many things are hard to maintain and understand. It's better to break them down into smaller, focused classes.
- **God Objects**: These are objects that know too much or do too much. They violate encapsulation and make your code harder to maintain.
- **Magic Numbers/Strings**: Using hardcoded values instead of constants or enums makes code less readable and harder to maintain.
- **Excessive Comments**: Comments should explain *why* something is done, not *what* is done. Over-commenting often points to poorly written code that needs refactoring.
- **Long Parameter Lists**: Methods with too many parameters are hard to read and maintain. Consider using objects to encapsulate the parameters.
- **Primitive Obsession**: Overuse of basic types (strings, ints, etc.) instead of creating specific types for your domain logic (like `URL` or `ID`).
- **Feature Envy**: When a method of one class seems to be overly interested in the details of another class, it usually means functionality is misplaced.
- **Switch Statements**: Large `switch` statements can often indicate that you should be using polymorphism or a pattern like Strategy or State.
- **Dead Code**: Unused or unnecessary code is clutter. It confuses the reader and increases the maintenance burden.
- **Temporary Field**: When an object has instance fields that are only set in certain scenarios, it can signal a design issue. The object might have too many responsibilities.
- **Inappropriate Intimacy**: When classes are too dependent on each other's implementation details, it reduces modularity and leads to tightly coupled code.
- **Refused Bequest**: When a subclass inherits methods and data it doesn't need from a parent class, it signals that inheritance might be misused. Composition might be better than inheritance in such cases.
- **Lazy Class**: A class that doesn't do much work, making its presence unnecessary, often indicates that its responsibilities can be merged elsewhere.
- **Too Many Getters and Setters**: Excessive use of getters and setters can break encapsulation. Instead of exposing internal data, consider using methods that encapsulate behavior.
- **Inconsistent Naming**: Not following a consistent naming convention for variables, methods, or classes makes code harder to understand.

Docker installation

Assumption:

- Docker desktop or compatible containerisation is available in the host machine.

Sample Docker compose

```
version: '3.8'  # Use the latest version of Docker Compose

services:
  api:
    image: mcr.microsoft.com/dotnet/aspnet:6.0  # Official .NET runtime
image for running the API
    container_name: urlshortener-api
    ports:
      - "5000:80"  # Expose the API on port 5000
    volumes:
      - ./api:/app  # Mount the API source code directory to the container
      - ./data:/app/Data  # Mount the local folder for SQLite database
storage
    working_dir: /app  # Set the working directory to the API folder in
the container
    command: ["dotnet", "UrlShortener2.dll"]  # Start the API by running
the .NET application
    depends_on:
      - angular  # Ensure the Angular UI starts first

  angular:
    image: node:16  # Official Node.js image for building the Angular UI
    container_name: urlshortener-ui
    volumes:
      - ./ui:/app  # Mount the Angular project directory
    working_dir: /app  # Set working directory to Angular project folder
    command: ["npm", "start"]  # Start the Angular application
    ports:
      - "4200:4200"  # Expose the Angular development server on port 4200
    environment:
      - CHOKIDAR_USEPOLLING=true  # Enable polling for file changes
(necessary when using mounted volumes)

volumes:
  data:
    driver: local  # SQLite database will be stored in the local volume
```

SQL Server installation

This is just for future reference, need to include it in the docker-compose after testing/completing with SQLite.

```
# docker pull mcr.microsoft.com/mssql/server

docker run -e "ACCEPT_EULA=Y" -e "MSSQL_SA_PASSWORD=P@ssword001" -e
"MSSQL_PID=Evaluation" -p 1433:1433  --name sqlpreview --hostname
sqlpreview -d mcr.microsoft.com/mssql/server
```