Problem Statement

- Long URLs can be difficult to share, especially on platforms with character limits like Twitter.
- Transform horrible links into branded links.

Features

- Returns a URL that is shorter than the original
- Shortened URL should allow redirects to the original URL
- Supports custom alias for short URLs
- URLs can be provided with expiry date time
- Deletion of expired URLs on a periodic basis

Assumptions

- Short URLs do not have to be human-readable.
- Custom short URLs must be unique and not conflict with existing URLs.
- Expired URLs will no longer redirect and be stored.
- The service will be a RESTful API.
- Short URLs will contain only alphanumerics with small case letters

Architecture/High-Level Components Interaction

The architecture of the URL shortener service consists of the following components:

- **Client**: The client is the interface that the user interacts with to create, view, and manage shortened URLs. This can be a web application, a mobile application, or a browser extension.

- **Application server**: The server-side application is responsible for handling requests from the client-side application, processing data, and returning responses to the client-side application. The server-side application includes the Django web framework and is responsible for handling the following:
    - **URL Shortening**: The server-side application accepts a long URL from the client-side application, generates a short URL for it, and saves it to the database.
    - **URL Redirection**: The server-side application accepts a short URL from the client-side application, looks up the corresponding long URL in the database, and redirects the user to the original long URL.
    - **URL Deletion**: The server-side application periodically deletes expired URLs from the database.

- **Database**: The database stores the URLs and their associated data. The server-side application interacts with the database to store and retrieve URLs. The Django web framework uses an Object-Relational Mapping (ORM) to abstract away the details of the database and provide an object-oriented interface for interacting with the data.

- **Background worker**: The background worker performs tasks asynchronously to improve performance and reliability. In this project, it is responsible for deleting expired URLs from the database.

Interaction between the components:

1. The client requests the application server to create or view a shortened URL.
2. The application server generates a shortened URL, stores it in the database, and returns the URL to the client.
3. When a user clicks on a shortened URL, the client requests the application server with the short URL.

4. The application server retrieves the original URL from the database and redirects the user to the original URL.
5. The background worker periodically checks for expired URLs and deletes them from the database.

This architecture ensures that the service is scalable, reliable, and efficient. It allows for separating concerns between the different components, making it easier to maintain and evolve the service over time.

Data Model

This is the **Url** model with all its attributes and methods.

- ***url_id***: a unique auto-incrementing integer value that acts as the primary key of the Url model.
- ***original_url***: a required URL field that stores the original URL entered by the user.
- ***short_url***: a unique string field with a maximum length of 15 characters that store the shortened URL generated by the system.
- ***created_at***: a date and time field that records the time the URL was created. AutoField while creation
- ***expires_at***: a date and time field that specifies the expiration time of the URL. This field is optional and can be left blank. If set, the URL will be deleted automatically once the expiration time is reached.

- Meta: an inner class that sets the ordering of Url objects by their creation date, with the most recent first.
- __str__: a method that returns a string representation of the Url object, showing the original URL and the shortened URL.
- expired: a boolean property that returns True if the URL has an expiration date set and that date has passed, otherwise False.

APIs

- /shorten_url
  Generates a short URL for the specified original URL.

- /<str:short_url>
  Redirects to the original URL corresponding to the specified short URL.

  Look into the code for more details. Prepare the shorten_url POST API very carefully.

Testing and Production Level Details (very important)

- Unit testing is performed on the Url model to ensure it functions as expected, including testing the string representation and expiration property.
- The ***ShortenUrlAPIView*** is tested to ensure it successfully creates a shortened URL and handles invalid URLs and custom aliases appropriately.
- The ***RedirectShortenUrlAPIView*** is tested to ensure it successfully redirects to the original URL and appropriately handles invalid shortened URLs.
- The test cases cover a variety of scenarios, including valid and invalid inputs, existing custom aliases, and expired URLs.
- Using test cases ensures that the application functions as intended and helps catch and fix any bugs or issues before they reach production.

Also, talk about some Production Level details:

To make the project production-ready, we need to consider the following aspects:

- **Deployment**: The project needs to be deployed on a production server that is scalable and reliable. We can use platforms like AWS, Google Cloud, or Heroku to deploy the project.

- **Security**: Security is a critical aspect of any production system. We need to make sure that the project is secure from unauthorized access, data breaches, and other security threats. We can use SSL/TLS certificates to secure the communication between the client and server, and also use security mechanisms like firewalls, intrusion detection and prevention systems, and regular security audits.

- **Monitoring**: We need to monitor the project's performance, availability, and other metrics to ensure that it is running smoothly. We can use monitoring tools like Prometheus, Grafana, or Nagios to monitor the project.

- **Scaling**: The project needs to be designed in a way that it can handle a large number of requests without any performance issues. We can use load balancers, auto-scaling groups, and other scaling mechanisms to ensure that the project can handle traffic spikes.

- **Testing**: We need to ensure that the project is thoroughly tested before it goes live. We can use automated testing tools like pytest or Django's built-in testing framework to test the project's functionality, performance, and security.

- **Continuous Integration/Continuous Deployment (CI/CD)**: We can use CI/CD tools like Jenkins, GitLab CI/CD, or Travis CI to automate the process of building, testing, and deploying the project. This ensures that any changes made to the codebase are automatically tested and deployed to production, reducing the risk of human error.

Future Plans

- Implementing more advanced analytics tracking to gather insights on usage patterns, traffic sources, and other key metrics that can help inform future improvements.
- Adding additional security measures, such as user authentication and authorization, to protect against potential misuse or unauthorized access to the system.
- Building out additional features, such as the ability to edit or delete existing URLs or to view a user's full history of shortened URLs.
- Investigating additional methods for generating short URLs that could offer greater efficiency, security, or flexibility.
- Scaling up the system to handle larger volumes of traffic and data and optimizing performance to minimize latency and improve overall user experience.