## Project Name:

## URL Shortener

Name: YEDUVAKA LIKHITHA

Designation: Python Development Intern

Intern ID:CC44573

Batch: August 2023

Mail ID: [20981a4914@raghuenggcollege.in](mailto:20981a4914@raghuenggcollege.in)

# Context Introduction and Usage:

In today's digital world, URLs (Uniform Resource Locators) are the primary means of accessing web resources. However, some URLs can be quite lengthy and cumbersome, making them challenging to share or remember. URL shorteners are tools that convert long URLs into shorter, more manageable links, making them easier to share on social media, emails, and other platforms.

The URL shortener project aims to develop a web application that takes long URLs as input and generates shortened URLs. Users can then use these shortened links to access the original long URLs. The project will use Python as the programming language, Flask as the web framework, and SQLite as the database.

# Objective:

The objective of the URL shortener project is to build a functional and user-friendly web application that allows users to shorten long URLs and share them more conveniently. The application should store the mappings between the shortened URLs and their corresponding long URLs in a database. When a user accesses a shortened URL, the application should redirect them to the original long URL.

# Project Prerequisities:

Before starting the URL shortener project, the following prerequisites should be in place:

1.Python Knowledge: Familiarity with Python programming is essential as the entire project will be implemented using Python.

2.Web Development Fundamentals: Basic understanding of web development concepts, including HTML, CSS, and JavaScript, is helpful for designing the frontend of the application.

3.Flask Framework: Knowledge of the Flask web framework is necessary, as it will be used to create the URL shortener web application.

4.SQLite Database: Understanding of how to work with SQLite databases in Python is required, as the application will use it to store URL mappings.

5.HTML/CSS/JS Basics: Basic knowledge of HTML, CSS, and JavaScript is necessary to design the user interface of the URL shortener.

6.IDE (Integrated Development Environment): Choose one of the preferred IDEs like Visual Studio Code, PyCharm, or Thonny for development.

7.Virtual Environment: Set up a virtual environment to manage project dependencies and avoid conflicts with the system-wide Python environment.

8.Package Management: Knowledge of package managers like `pip` is essential to install the required Python packages (e.g., Flask, SQLite, etc.).

IDE Selection:

You can use any of the following IDEs for this project:

\* Visual Studio Code (VS Code), PyCharm, Thonny

These IDEs provide a user-friendly interface, code highlighting, debugging tools, and other features that will make the development process smoother and more efficient.

With these prerequisites in place, you are ready to embark on your URL shortener project using Python and the specified software requirements.

# Project File Structure:

Step 1: Prepare Your Environment

Step 2: Set up your python URL Shortener

Step 3: Tidy up your Code

Step 4: Manage Your URLs

Step 5: Conclusion

## Step 1: Prepare Your Environment

# shortener\_app/config.py

from pydantic import BaseSettings

class Settings(BaseSettings):

env\_name: str = "Local"

base\_url: str = "http://localhost:8000"

db\_url: str = "sqlite:///./shortener.db"

def get\_settings() -> Settings:

settings = Settings()

print(f"Loading settings for: {settings.env\_name}")

return settings

In line 3, you’re importing [pydantic](https://pydantic-docs.helpmanual.io/). You installed pydantic automatically when you installed fastapi with pip. pydantic is a library that uses type annotation to validate data and manage settings.

* The Settings class that you define in line 5 is a **subclass** of BaseSettings. The BaseSettings class comes in handy to define environment variables in your application. You only have to define the variables that you want to use, and pydantic takes care of the rest.

>>> from shortener\_app.config import get\_settings

>>> get\_settings().base\_url

Loading settings for: Local

'http://localhost:8000'

>>> get\_settings().db\_url

Loading settings for: Local

'sqlite:///./shortener.db'

* To show a message once your settings are loaded, you create get\_settings() in lines 10 to 13. The get\_settings() function returns an instance of your Settings class and will provide you with the option of **caching** your settings. But before you investigate why you need caching, run get\_settings() in the [interactive Python interpreter](https://realpython.com/interacting-with-python/)

# shortener\_app/config.py

from functools import lru\_cache

from pydantic import BaseSettings

class Settings(BaseSettings):

env\_name: str = "Local"

base\_url: str = "http://localhost:8000"

db\_url: str = "sqlite:///./shortener.db"

@lru\_cache

def get\_settings() -> Settings:

settings = Settings()

print(f"Loading settings for: {settings.env\_name}")

return settings

# shortener\_app/main.py

from fastapi import FastAPI

app = FastAPI()

@app.get("/")

def read\_root():

return "Welcome to the URL shortener API :)"

* You import lru\_cache from Python’s functools module in line 3. The @lru\_cache [decorator](https://realpython.com/primer-on-python-decorators/) allows you to cache the result of get\_settings() using the LRU strategy. Run the command below to see how the caching works

>>> from shortener\_app.config import get\_settings

>>> get\_settings().base\_url

Loading settings for: Local

'http://localhost:8000'

>>> get\_settings().db\_url

'sqlite:///./shortener.db'

* To load your external .env file, adjust your Settings class in your config.py file:

# shortener\_app/config.py

# ...

class Settings(BaseSettings):

env\_name: str = "Local"

base\_url: str = "http://localhost:8000"

db\_url: str = "sqlite:///./shortener.db"

class Config:

env\_file = ".env"

# ...

* When you add the Config class with the path to your env\_file to your settings, pydantic loads your environment variables from the .env file. Test your external environment variables by running the commands below

from shortener\_app.config import get\_settings

get\_settings().base\_url

get\_settings().db\_url

# Source code for step1:

from functools import lru\_cache

from pydantic import BaseSettings

class Settings(BaseSettings):

env\_name: str = "Local"

base\_url: str = "http://localhost:8000"

db\_url: str = "sqlite:///./shortener.db"

class Config:

env\_file = ".env"

@lru\_cache

def get\_settings() -> Settings:

settings = Settings()

print(f"Loading settings for: {settings.env\_name}")

return settings

# Step 2: Set Up Your Python URL Shortener

### **Create Your FastAPI App:**

# shortener\_app/main.py

from fastapi import FastAPI

app = FastAPI()

@app.get("/")

def read\_root():

return "Welcome to the URL shortener API :)"

* Start by creating the base models for your API request and response bodies in a schemas.py file:

# shortener\_app/schemas.py

from pydantic import BaseModel

class URLBase(BaseModel):

target\_url: str

class URL(URLBase):

is\_active: bool

clicks: int

class Config:

orm\_mode = True

class URLInfo(URL):

url: str

admin\_url: str

* Head back to main.py to use the URLBase schema in a POST endpoint:

# shortener\_app/main.py

import validators

from fastapi import FastAPI, HTTPException

from . import schemas

# ...

def raise\_bad\_request(message):

raise HTTPException(status\_code=400, detail=message)

# ...

@app.post("/url")

def create\_url(url: schemas.URLBase):

if not validators.url(url.target\_url):

raise\_bad\_request(message="Your provided URL is not valid")

return f"TODO: Create database entry for: {url.target\_url}"

* Prepare Your SQLite Database

# shortener\_app/database.py

from sqlalchemy import create\_engine

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.orm import sessionmaker

from .config import get\_settings

engine = create\_engine(

get\_settings().db\_url, connect\_args={"check\_same\_thread": False}

)

SessionLocal = sessionmaker(

autocommit=False, autoflush=False, bind=engine

)

Base = declarative\_base()

* While database.py contains information about your database connection, the models.py file will describe the content of your database. To continue, create models.py:

# shortener\_app/models.py

from sqlalchemy import Boolean, Column, Integer, String

from .database import Base

class URL(Base):

\_\_tablename\_\_ = "urls"

id = Column(Integer, primary\_key=True)

key = Column(String, unique=True, index=True)

secret\_key = Column(String, unique=True, index=True)

target\_url = Column(String, index=True)

is\_active = Column(Boolean, default=True)

clicks = Column(Integer, default=0)

* With the database model in place, you can now link your app to the database. For now, you’ll add most of the code to communicate with your database to main.py:

# shortener\_app/main.py

import secrets

import validators

from fastapi import Depends, FastAPI, HTTPException

from sqlalchemy.orm import Session

from . import models, schemas

from .database import SessionLocal, engine

app = FastAPI()

models.Base.metadata.create\_all(bind=engine)

def get\_db():

db = SessionLocal()

try:

yield db

finally:

db.close()

# ...

@app.post("/url", response\_model=schemas.URLInfo)

def create\_url(url: schemas.URLBase, db: Session = Depends(get\_db)):

if not validators.url(url.target\_url):

raise\_bad\_request(message="Your provided URL is not valid")

chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

key = "".join(secrets.choice(chars) for \_ in range(5))

secret\_key = "".join(secrets.choice(chars) for \_ in range(8))

db\_url = models.URL(

target\_url=url.target\_url, key=key, secret\_key=secret\_key

)

db.add(db\_url)

db.commit()

db.refresh(db\_url)

db\_url.url = key

db\_url.admin\_url = secret\_key

return db\_url

* So far, the database contains the table that you defined in models.py, but it doesn’t contain any data.

>>> from shortener\_app.database import SessionLocal

Loading settings for: Development

>>> db = SessionLocal()

>>> from shortener\_app.models import URL

>>> db.query(URL).all()

[]

* Once you’ve used the POST endpoint of your app, verify that the request created the database entries accordingly:

>>> db.query(URL).all()

[<shortener\_app.models.URL object at 0x104c65cc0>, ...]

* Forward a Shortened URL

# shortener\_app/main.py

# ...

from fastapi import Depends, FastAPI, HTTPException, Request

from fastapi.responses import RedirectResponse

# ...

def raise\_not\_found(request):

message = f"URL '{request.url}' doesn't exist"

raise HTTPException(status\_code=404, detail=message)

# ...

@app.get("/{url\_key}")

def forward\_to\_target\_url(

url\_key: str,

request: Request,

db: Session = Depends(get\_db)

):

db\_url = (

db.query(models.URL)

.filter(models.URL.key == url\_key, models.URL.is\_active)

.first()

from functools import lru\_cache

from pydantic import BaseSettings

class Settings(BaseSettings):

env\_name: str = "Local"

base\_url: str = "http://localhost:8000"

db\_url: str = "sqlite:///./shortener.db"

class Config:

env\_file = ".env"

@lru\_cache

def get\_settings() -> Settings:

settings = Settings()

print(f"Loading settings for: {settings.env\_name}")

return settings if db\_url:

return RedirectResponse(db\_url.target\_url)

else:

raise\_not\_found(request)

# Source code for step2:

from functools import lru\_cache

from pydantic import BaseSettings

class Settings(BaseSettings):

env\_name: str = "Local"

base\_url: str = "http://localhost:8000"

db\_url: str = "sqlite:///./shortener.db"

class Config:

env\_file = ".env"

@lru\_cache

def get\_settings() -> Settings:

settings = Settings()

print(f"Loading settings for: {settings.env\_name}")

return settings

# Step 3: Tidy Up Your Code

* Start by having a look at the current state of create\_url():

# shortener\_app/main.py

@app.post("/url", response\_model=schemas.URLInfo)

def create\_url(url: schemas.URLBase, db: Session = Depends(get\_db)):

if not validators.url(url.target\_url): raise\_bad\_request(message="Your provided URL is not valid")

chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

key = "".join(secrets.choice(chars) for \_ in range(5))

secret\_key = "".join(secrets.choice(chars) for \_ in range(8))

db\_url = models.URL(

target\_url=url.target\_url, key=key, secret\_key=secret\_key

)

db.add(db\_url)

db.commit()

db.refresh(db\_url)

db\_url.url = key

db\_url.admin\_url = secret\_key

return db\_url

* Next, look at forward\_to\_target\_url(). Again, take a moment to find the problems on your own first. Then read on to learn what the problems are:

# shortener\_app/main.py

@app.get("/{url\_key}")

def forward\_to\_target\_url(

url\_key: str,

request: Request,

db: Session = Depends(get\_db)

):

db\_url = (

db.query(models.URL)

.filter(models.URL.key == url\_key, models.URL.is\_active)

.first()

)

if db\_url:

return RedirectResponse(db\_url.target\_url)

else:

raise\_not\_found(request)

* Refactor your code

# shortener\_app/keygen.py

import secrets

import string

def create\_random\_key(length: int = 5) -> str:

chars = string.ascii\_uppercase + string.digits

return "".join(secrets.choice(chars) for \_ in range(length))

* By modularizing the random string creation into its own function, you can test it conveniently in the Python interpreter:

>>> from shortener\_app.keygen import create\_random\_key

>>> create\_random\_key()

'81I5B'

>>> create\_random\_key(length=8)

'WAT3K9VQ'

* create a crud.py file. Your crud.py file will contain functions that perform actions to **Create, Read, Update, and Delete (CRUD)** items in your database. Go ahead and add create\_db\_url():

# shortener\_app/crud.py

from sqlalchemy.orm import Session

from . import keygen, models, schemas

def create\_db\_url(db: Session, url: schemas.URLBase) -> models.URL:

key = keygen.create\_random\_key()

secret\_key = keygen.create\_random\_key(length=8)

db\_url = models.URL(

target\_url=url.target\_url, key=key, secret\_key=secret\_key

)

db.add(db\_url)

db.commit()

db.refresh(db\_url)

return db\_url

* First, [define a function](https://realpython.com/defining-your-own-python-function/) that tells , if a key already exists in your database:

# shortener\_app/crud.py

# ...

def get\_db\_url\_by\_key(db: Session, url\_key: str) -> models.URL:

return (

db.query(models.URL)

.filter(models.URL.key == url\_key, models.URL.is\_active)

.first()

)

* Now you can create a function that makes sure you generate a unique key. Move back to keygen.py and add create\_unique\_random\_key():

# shortener\_app/keygen.py

# ...

from sqlalchemy.orm import Session

from . import crud

# ...

def create\_unique\_random\_key(db: Session) -> str:

key = create\_random\_key()

while crud.get\_db\_url\_by\_key(db, key):

key = create\_random\_key()

return key

* With this function in place, update your create\_db\_url() function in crud.py:

# shortener\_app/crud.py

# ...

def create\_db\_url(db: Session, url: schemas.URLBase) -> models.URL:

key = keygen.create\_unique\_random\_key(db)

secret\_key = f"{key}\_{keygen.create\_random\_key(length=8)}"

db\_url = models.URL(

target\_url=url.target\_url, key=key, secret\_key=secret\_key

)

db.add(db\_url)

db.commit()

db.refresh(db\_url)

return db\_url

* There are two advantages to creating secret\_key like this:
* The key prefix indicates which shortened URL secret\_key belongs to.
* You’re not hitting the database again when creating another random string.
* Head back to main.py and update create\_url() to use crud.create\_db\_url():

# shortener\_app/main.py

# ...

from . import crud, models, schemas

# ...

@app.post("/url", response\_model=schemas.URLInfo)

def create\_url(url: schemas.URLBase, db: Session = Depends(get\_db)):

if not validators.url(url.target\_url):

raise\_bad\_request(message="Your provided URL is not valid")

db\_url = crud.create\_db\_url(db=db, url=url)

db\_url.url = db\_url.key

db\_url.admin\_url = db\_url.secret\_key

return db\_url

* Next, leverage the creation of get\_db\_url\_by\_key() and update forward\_to\_target\_url():

# shortener\_app/main.py

# ...

@app.get("/{url\_key}")

def forward\_to\_target\_url(

url\_key: str,

request: Request,

db: Session = Depends(get\_db)

):

if db\_url := crud.get\_db\_url\_by\_key(db=db, url\_key=url\_key):

return RedirectResponse(db\_url.target\_url)

else:

raise\_not\_found(request)

# Source code for step3:

from functools import lru\_cache

from pydantic import BaseSettings

class Settings(BaseSettings):

env\_name: str = "Local"

base\_url: str = "http://localhost:8000"

db\_url: str = "sqlite:///./shortener.db"

class Config:

env\_file = ".env"

@lru\_cache

def get\_settings() -> Settings:

settings = Settings()

print(f"Loading settings for: {settings.env\_name}")

return settings

# Step 4: Manage Your URLs:

* Start by creating get\_db\_url\_by\_secret\_key() in your crud.py file:

# shortener\_app/crud.py

# ...

def get\_db\_url\_by\_secret\_key(db: Session, secret\_key: str) -> models.URL:

return (

db.query(models.URL)

.filter(models.URL.secret\_key == secret\_key, models.URL.is\_active)

.first()

)

* Your get\_db\_url\_by\_secret\_key() function checks your database for an active database entry with the provided secret\_key. If a database entry is found, then you return the entry. Otherwise, you return None.
* You work with the returned data in get\_url\_info() in main.py:

# shortener\_app/main.py

# ...

@app.get(

"/admin/{secret\_key}",

name="administration info",

response\_model=schemas.URLInfo,

)

def get\_url\_info(

secret\_key: str, request: Request, db: Session = Depends(get\_db)

):

if db\_url := crud.get\_db\_url\_by\_secret\_key(db, secret\_key=secret\_key):

db\_url.url = db\_url.key

db\_url.admin\_url = db\_url.secret\_key

return db\_url

else:

raise\_not\_found(request)

* In line 5, you’re defining a new API endpoint at the /admin/{secret\_key} URL. You also give this endpoint the name "administration info", making it easier to refer to it later. As response\_model, you expect a URLInfo schema in line 8.
* After you get the database entry of crud.get\_db\_url\_by\_secret\_key() in line 13, you assign it to db\_url and check it right away. You’re using an assignment expression for the if statement of this line.

# shortener\_app/main.py

# ...

from starlette.datastructures import URL

# ...

from .config import get\_settings

# ...

def get\_admin\_info(db\_url: models.URL) -> schemas.URLInfo:

base\_url = URL(get\_settings().base\_url)

admin\_endpoint = app.url\_path\_for(

"administration info", secret\_key=db\_url.secret\_key

)

db\_url.url = str(base\_url.replace(path=db\_url.key))

db\_url.admin\_url = str(base\_url.replace(path=admin\_endpoint))

return db\_url

# ...

* In get\_admin\_info(), you go even one step further than just receiving the .url and .admin\_url attributes. You also leverage the URL class from the [starlette](https://www.starlette.io/) package that comes with FastAPI. To create base\_url in line 14, you pass in base\_url from your settings to initialize the URL class. After that, you can use the .replace() method to construct a full URL.

# shortener\_app/main.py

# ...

@app.post("/url", response\_model=schemas.URLInfo)

def create\_url(url: schemas.URLBase, db: Session = Depends(get\_db)):

if not validators.url(url.target\_url):

raise\_bad\_request(message="Your provided URL is not valid")

db\_url = crud.create\_db\_url(db=db, url=url)

return get\_admin\_info(db\_url)

# ...

@app.get(

"/admin/{secret\_key}",

name="administration info",

response\_model=schemas.URLInfo,

)

def get\_url\_info(

secret\_key: str, request: Request, db: Session = Depends(get\_db)

):

if db\_url := crud.get\_db\_url\_by\_secret\_key(db, secret\_key=secret\_key):

return get\_admin\_info(db\_url)

else:

raise\_not\_found(request)

* Update Your Visitor Count
* **Note:** The methods .commit() and .refresh() are from db, not db\_url.
* When you forward to a target URL, you call the update\_db\_clicks() function that you just created. Therefore, you need to adjust the forward\_to\_target\_url() function in main.py:
* Insert the crud.update\_db\_clicks() function call in line 12. Every time a friend uses your shortened URL, the click count increases. You can use the number of clicks to see how often a link was visited.
* Python URL shortener app is great for sharing a link with friends. Once your friends have visited the link, you may want to delete the shortened URL again.
* Just like with the update\_db\_clicks() function, you start by creating a new function in crud.py:
* The code that you wrote focused on getting the app working first. But you didn’t stop there. You took your time to inspect your codebase and spotted opportunities to **refactor your code**. You’re now in a great position to think about extending your Python URL shortener.
* The code that you wrote focused on getting the app working first. But you didn’t stop there. You took your time to inspect your codebase and spotted opportunities to **refactor your code**. You’re now in a great position to think about extending your Python URL shortener.
* The code that you wrote focused on getting the app working first. But you didn’t stop there. You took your time to inspect your codebase and spotted opportunities to **refactor your code**. You’re now in a great position to think about extending your Python URL shortener.
* The body of delete\_url() probably looks familiar by now. You’re using an assignment expression (:=) in line 9 to assign db\_url the return of crud.deactivate\_db\_url\_by\_secret\_key() in line 10. If a database entry with the provided secret\_key exists and was deactivated, then you return the success message in line 11. Otherwise, you trigger raise\_not\_found() in line 13.

# Step 5: Conclusion:

* I Have Built a web app with **FastAPI** to create and manage shortened URLs. With your URL shortener, you can now convert long URLs into tiny, shareable links. When someone clicks your shortened URL, then your URL shortener app will forward them to the targeted URL.

**In this tutorial, you learned how to:**

* Create a **REST API** with FastAPI
* Run a development web server with **Uvicorn**
* Model an **SQLite** database
* Investigate the auto-generated **API documentation**
* Interact with the database with **CRUD actions**
* The code that you wrote focused on getting the app working first. But you didn’t stop there. You took your time to inspect your codebase and spotted opportunities to **refactor your code**. You’re now in a great position to think about extending your Python URL shortener.

# Final Source code:

import validators

from fastapi import Depends, FastAPI, HTTPException, Request

from fastapi.responses import RedirectResponse

from sqlalchemy.orm import Session

from starlette.datastructures import URL

from . import crud, models, schemas

from .database import SessionLocal, engine

from .config import get\_settings

app = FastAPI()

models.Base.metadata.create\_all(bind=engine)

def get\_db():

db = SessionLocal()

try:

yield db

finally:

db.close()

def get\_admin\_info(db\_url: models.URL) -> schemas.URLInfo:

base\_url = URL(get\_settings().base\_url)

admin\_endpoint = app.url\_path\_for(

"administration info", secret\_key=db\_url.secret\_key

)

db\_url.url = str(base\_url.replace(path=db\_url.key))

db\_url.admin\_url = str(base\_url.replace(path=admin\_endpoint))

return db\_url

def raise\_bad\_request(message):

raise HTTPException(status\_code=400, detail=message)

def raise\_not\_found(request):

message = f"URL '{request.url}' doesn't exist"

raise HTTPException(status\_code=404, detail=message)

@app.get("/")

def read\_root():

return "Welcome to the URL shortener API :)"

@app.post("/url", response\_model=schemas.URLInfo)

def create\_url(url: schemas.URLBase, db: Session = Depends(get\_db)):

if not validators.url(url.target\_url):

raise\_bad\_request(message="Your provided URL is not valid")

db\_url = crud.create\_db\_url(db=db, url=url)

return get\_admin\_info(db\_url)

@app.get("/{url\_key}")

def forward\_to\_target\_url(

url\_key: str, request: Request, db: Session = Depends(get\_db)

):

if db\_url := crud.get\_db\_url\_by\_key(db=db, url\_key=url\_key):

crud.update\_db\_clicks(db=db, db\_url=db\_url)

return RedirectResponse(db\_url.target\_url)

else:

raise\_not\_found(request)

@app.get(

"/admin/{secret\_key}",

name="administration info",

response\_model=schemas.URLInfo,

)

def get\_url\_info(

secret\_key: str, request: Request, db: Session = Depends(get\_db)

):

if db\_url := crud.get\_db\_url\_by\_secret\_key(db, secret\_key=secret\_key):

return get\_admin\_info(db\_url)

else:

raise\_not\_found(request)

@app.delete("/admin/{secret\_key}")

def delete\_url(

secret\_key: str, request: Request, db: Session = Depends(get\_db)

):

if db\_url := crud.deactivate\_db\_url\_by\_secret\_key(

db, secret\_key=secret\_key

):

message = (

f"Successfully deleted shortened URL for '{db\_url.target\_url}'"

)

return {"detail": message}

else:

raise\_not\_found(request)

# Sample Input and Output:

URL Shortener

1. Shorten a URL

2. Retrieve original URL

3. Exit

Enter your choice (1/2/3): 1

Enter the long URL: https://www.example.com/this-is-a-long-url-that-needs-shortening

Shortened URL: http://yourdomain.com/abc123

URL Shortener

1. Shorten a URL

2. Retrieve original URL

3. Exit

Enter your choice (1/2/3): 2

Enter the shortened URL: http://yourdomain.com/abc123

Original URL: https://www.example.com/this-is-a-long-url-that-needs-shortening

URL Shortener

1. Shorten a URL

2. Retrieve original URL

3. Exit

Enter your choice (1/2/3): 3

Exiting...





