*Follow this guide to create a headless CMS in just ten steps.*

Say you want to start a blog or showcase your products and services on a website. One option is to build everything from scratch using HTML, CSS, and JavaScript. This would require building databases, an admin login, and interfaces so that you can log in and update the information. As this takes a lot of effort, you should probably use a proven Content Management System (CMS) if you’re not familiar with programming. A CMS provides all the tools required to manage content and apply different layouts to a website out-of-the-box.

There are many different [CMSs available](https://nordicapis.com/api-based-cms-buyers-guide/), such as WordPress, Wix, Contentful, or Squarespace, which can be used to generate websites and web applications. A CMS provides visual editing interfaces, templates, custom code, and other content management capabilities, all from a single environment. WordPress, one of the most powerful options, boasts many plugins to extend behavior and introduce powerful eCommerce abilities.

So, now the next question arises: what is a **headless CMS**?

## What Is a Headless CMS?

So, we have been using CMSs for about 20 years now. However, the world has changed a lot. Today’s mobile era has seen much growth into novel frontiers, such as IoT, bots, digital assistance, and VR. However, traditional CMS wasn’t built for these cutting-edge platforms.

Today, content must be displayed on a variety of devices in different formats. Since traditional CMS was not developed for this purpose, we require a new type of client-agnostic content management system. This is where headless CMS comes in.

A headless CMS focuses solely on the backend process of managing the content. It doesn’t control how the frontend presentation looks. Instead, a headless CMS uses an API to provide content to the end channels. In this way, a headless CMS is detached from the client (the “head”) and is thus headless. Using APIs to separate concerns means your content is deliverable to any platform; it could be an Angular website, mobile application, or even a smartwatch.

### Pros of Headless CMS

* It is more secure when compared with traditional CMS
* It is also smaller in size
* Faster than the conventional CMS
* It allows you to choose any languages for frontend development
* It enables you to publish your content to different platforms

### Cons of Headless CMS

* You have to manage two parts at the same time — the backend and frontend
* You’ll need a completely different infrastructure to maintain the frontend
* It can be more expensive to maintain than a traditional CMS

### Some Popular Headless CMS

* Directus
* Prismic
* Kentico Kontent
* Bloomreach
* Magnolia

Now that we understand what a headless CMs is, let’s learn how to create one. In this walkthrough, we’ll create a headless CMS using Flask and MySQL.

## Prerequisites

* Python
* MySQL
* Flask
* Code Editor like VS Code
* MySQL
* SQLalchemy

## Step – 1: The Setup

Open your terminal and create a new folder using the command mkdir. We’re using cms as the folder name:

mkdir cms

Now open the folder:

cd cms  
virtualenv .  
pipenv install flask flask-sqlalchemy flask-cors

Now we must create different files and folders inside the root folder. For the sake of this walkthrough, create this structure of five folders and one file:

* Blog
* Login
* Tag
* blog\_tags
* User
* \_\_init\_\_.py

Now create some Python files as well. The final folder structure has to be like this:

* Blog  
  – blog\_model.py  
  – blog\_routes.py
* Login  
  – login\_route.py
* Tag  
  – tag\_model.py
* blog\_tags  
  – blog\_tag\_table.py
* User  
  – user\_model.py

Once you’ve created the project structure, now install Flask and virtualenv. We have already covered how to set up the Flask in a previous article. You can check it out [here](https://nordicapis.com/how-to-create-an-api-using-the-flask-framework/) and then continue with the next steps.

Let’s install flask-sqlalchemy. On your terminal, paste the below command:

python3 -m pip install flask-sqlalchemy

What exactly is flask-sqlalchemy? Well, it is a Flask extension that adds support for sqlalchemy and simplifies many MySQL tasks. It uses Object Relational Mapping (ORM), making it easier for you to run queries without writing down the raw SQL statements.

### Initializing

Once all the dependencies are installed, it’s time to write some code. So, open cms/\_\_init\_\_.py in your code editor and paste the below code:

from flask import Flask  
from flask\_sqlalchemy import SQLAlchemy  
from flask\_cors import CORS  
  
db = SQLAlchemy()  
  
def create\_app():  
 app = Flask(\_\_name\_\_)  
 app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///flaskdatabase.db'  
 app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False  
 CORS(app)  
  
 db.init\_app(app)  
  
 return app

### Explanation

Here we have created a function called create\_app(), which basically initializes our application and database at the same time, which we can use anywhere in the code.

## Step – 2: Database Setup

In this step, we’ll be creating tables for our headless CMS. These tables will store all the data that we’ll publish through our CMS. We’ll start with the blog table which will have columns: id, title, text, date\_of\_publish, image, tags. If you want, you can add some more columns if required, but for now, for the sake of simplicity, we’re just creating the basic columns which are there in most CMSs.

Now open Blog/blog\_model.py and paste the below code:

from cms import db  
from datetime import datetime  
from cms.blog\_tags.blog\_tag\_table import tag\_blog  
  
tags=db.relationship('Tag',secondary=tag\_blog,backref=db.backref('blogs\_associated',lazy="dynamic"))  
  
class Blog(db.Model):  
 id = db.Column(db.Integer,primary\_key=True)  
 title=db.Column(db.String(50),nullable=False)  
 text=db.Column(db.Text,nullable=False)  
 image= db.Column(db.String,nullable=False)  
 date\_of\_publish = db.Column(db.DateTime, default=datetime.utcnow)  
  
 @property  
 def serialize(self):  
 return {  
 'id': self.id,  
 'title': self.title,  
 text: self.text,  
 image: self.image,  
 date\_of\_publish: self.date\_of\_publish,  
 }

### Explanation

Here we are importing different modules like db for database connections, and datetime for timestamps. We have created a model Blog and have defined all the fields in it. There’s a function called serialize(self), which is used to return all the data in JSON form.

You might have noticed that we haven’t defined tags here, right? This is because tags is a foreign key that will come from a completely different table called tags. Basically, one post can have many tags, and one tag can be associated with multiple blog posts.

Now open cms/Tag/tag\_model.py and paste the below code:

from cms import db  
  
class Tag(db.Model):  
 id=db.Column(db.Integer,primary\_key=True)  
 name=db.Column(db.String(20))  
 @property  
 def serialize(self):  
 return {  
 'id': self.id,  
 'name': self.name   
 }

### Explanation

We have defined the model with id and name columns since tags don’t require more than these columns.

Once the model is done, it’s time to create a table also, so open the file cms/blog\_tags/blog\_tag\_table.py, and paste the below code:

from cms import db  
  
tag\_blog = db.Table('tag\_blog',  
 db.Column('tag\_id',db.Integer,db.ForeignKey('tag.id'), primary\_key=True),  
 db.Column('blog\_id', db.Integer,db.ForeignKey('blog.id'),primary\_key=True)  
)

### Explanation

This table holds the relationship between the blog table and tags table where tag\_id is mapped with blog\_id.

## Step –3: Adding Blueprints

Now we have to edit the file Blog/blog\_routes.py and add the below lines of code:

blogs= Blueprint('blogs',\_\_name\_\_)

Open cms/\_\_init\_\_.py and add the below code:

from cms.Blog.blog\_routes import blogs  
app.register\_blueprint(blogs)

### Explanation

We’re adding blueprints here. It helps us to break the application into small components that can be reused anywhere. Here we have defined blogs as the blueprint.

## Step –4: Creating a Route for Publishing a Blog Post

Now open your blog\_routes.py and add the below code:

from flask import Blueprint,request,jsonify,make\_response  
from flask\_jwt\_extended import jwt\_required  
from cms import db  
from cms.Blog.blog\_model import Blog  
from cms.Tag.tag\_model import Tag  
  
blogs= Blueprint('blogs',\_\_name\_\_)  
@blogs.route('/add\_post',methods=["POST"])  
def create\_blog():  
 data = request.get\_json()  
  
 new\_blog=Blog(title=data["title"],content=data["content"],image=data["image"])  
  
 for tag in data["tags"]:  
 current\_tag=Tag.query.filter\_by(name=tag).first()  
 if(current\_tag):  
 current\_tag.blogs\_associated.append(new\_blog)  
 else:  
 new\_tag=Tag(name=tag)  
 new\_tag.blogs\_associated.append(new\_blog)  
 db.session.add(new\_tag)  
   
  
 db.session.add(new\_blog)  
 db.session.commit()  
  
 blog\_id = getattr(new\_blog, "id")  
 return jsonify({"id": blog\_id})

### Explanation

We have created a route /blog\_post, which invokes the function that will create create\_blog(). This function is basically used to create a blog post and accepts title, text, image, and tags. We’re running a loop where it can accept multiple tags, and if a tag doesn’t exist, then it’ll create a new tag and associate it with the blog post.

## Step –5: Creating Route to Fetch the Blog Posts

We will create two different routes to fetch the blog posts. One route will fetch all the blog posts while the other one will be used to fetch blog posts on the basis of id. This can be used to search the blog posts when a user opens the full blog post. Now open blog\_routes.py and paste the below code:

@blogs.route('/blogs',methods=["GET"])  
def get\_all\_blogs():  
 blogs= Blog.query.all()  
 serialized\_data = []  
 for blog in blogs:  
 serialized\_data.append(blog.serialize)  
  
 return jsonify({"all\_blogs": serialized\_data})

### Explanation

We have defined a route /blogs, which runs a SELECT query using the ORM and returns a JSON containing all the blog posts and their data under the all\_blogs key.

Now to fetch the blog post with specific id paste the below code:

@blogs.route('/blog/<int:id>',methods=["GET"])  
def get\_single\_blog(id):  
 blog = Blog.query.filter\_by(id=id).first()  
 serialized\_blog = blog.serialize  
 serialized\_blog["tags"] = []  
  
 for tag in blog.tags:  
 serialized\_blog["tags"].append(tag.serialize)  
  
 return jsonify({"single\_blog": serialized\_blog})

### Explanation

We have defined another route /blog, which accepts an integer value and returns all the blog data in a JSON under the single\_blog key.

## Step –6: Creating a Route to Delete a Blog Post

So far we have covered how to create a blog. Now here’s how to delete a blog post. In the blog\_routes.py, paste the below code:

@blogs.route('/delete\_post/<int:id>', methods=["DELETE"])  
def delete\_post(id):  
 blog = Blog.query.filter\_by(id=id).first()  
 db.session.delete(blog)  
 db.session.commit()  
  
 return jsonify("Blog was deleted"),200

### Explanation

Here we have defined a route /delete\_post, which accepts the id of the blog post and runs the delete query for the associated blog id.

## Step –7: Creating a Route to Update a Blog Post

To update a blog post we’ll use the PUT method here which will take a blog id as input parameter. So in blog\_routes.py add the below code:

@blogs.route('/update\_post/<int:id>', methods=["PUT"])  
def update\_post(id):  
 data = request.get\_json()  
 blog=Blog.query.filter\_by(id=id).first\_or\_404()  
  
 blog.title = data["title"]  
 blog.text=data["text"]  
 blog.image=data["image"]  
  
 updated\_blog = blog.serialize  
  
 db.session.commit()  
 return jsonify({"blog\_id": blog.id})

### Explanation

We have added one more route that is /update\_post, which uses the PUT method and runs an UPDATE query on the passed blog id.

## Step –8: Adding Admin User and Login Route

Now we have defined all the paths that can perform CRUD operations on the blog post. But we also have to prevent unauthorized access, right? This will ensure that no unauthorized person can update or add a blog post.

So we’ll first create a user model which will used to store the user info. So, open the file cms/User/user\_model.py, and paste the below code:

from cms import db  
  
class User(db.Model):  
 id=db.Column(db.Integer,primary\_key=True)  
 email=db.Column(db.String(120),nullable=False)  
 password=db.Column(db.String(120),nullable=False)

Here we’ll have only one admin user, so we don’t need to create a route for that. So in your \_\_init\_\_.py file paste the below code:

@click.command(name='add\_admin')   
 @with\_appcontext  
 def add\_admin():  
 admin=User(email="ADMIN EMAIL",password="YOUR PASSWORD STRING")  
 admin.password = generate\_password\_hash(admin.password,'sha256',salt\_length=12)  
 db.session.add(admin)  
 db.session.commit()  
  
 app.cli.add\_command(add\_admin)

And on the top, add the below code:

import click  
from flask.cli import with\_appcontext  
from werkzeug.security import generate\_password\_hash

### Explanation

Here we are storing email and password for the admin user. Now we can not store it as plain text, so we’re using SHA256 hashing.

Once the above part is done, we have to create a route for admin login. So to do that, we have to open the Login/login\_route.py file and paste the below code:

from flask import Blueprint,request,jsonify  
from cms.User.user\_model import User  
from flask\_jwt\_extended import create\_access\_token  
from werkzeug.security import check\_password\_hash   
  
login=Blueprint('login', \_\_name\_\_)  
  
@login.route('/login', methods=["POST"])  
def log\_in():  
 request\_data = request.get\_json()  
  
 user=User.query.filter\_by(email=request\_data["email"]).first()  
 if user:  
 if check\_password\_hash(user.password,request\_data["password"]):  
 jwt\_token=create\_access\_token(identity=user.email)  
 return jsonify({"token":jwt\_token})  
 else:  
 return "Invalid email or password",400

### Explanation

Here we have defined a route /login, which will take email and password. Once both of them are correct, it’ll return a JWT token you can use to make the next requests.

## Step –9: Implementing JWT

Now we have added an admin user, and the login part is also done. But to make it more secure, we’ll implement JWT on add\_post and update\_post to prevent unauthorized access. On your terminal, paste the below code:

pipenv install flask-jwt-extended

This will install JWT, which you can use to implement JWT.

Now, open \_\_init\_\_.py and paste the below code:

app.config['JWT\_SECRET\_KEY']=ADD YOUR SECRET STRING HERE  
jwt=JWTManager(app)

Now open file Blog\_routes.py and update the below routes:

@blogs.route('/delete\_post/<int:id>', methods=["DELETE"])  
@jwt\_required  
def delete\_post(id):  
 blog = Blog.query.filter\_by(id=id).first()  
 db.session.delete(blog)  
 db.session.commit()  
  
 return jsonify("Blog was deleted"),200

Similarly, add the same @jwt\_required below the line @blogs.route('/add\_post',methods=["POST"])

## Step –10: Finalizing the Setup

In the end, your cms/\_\_init\_\_.py should look something like this:

from flask import Flask  
from flask\_sqlalchemy import SQLAlchemy  
from flask\_cors import CORS  
import click  
from flask.cli import with\_appcontext  
from flask\_jwt\_extended import JWTManager  
from werkzeug.security import generate\_password\_hash  
  
db = SQLAlchemy()  
  
def create\_app():  
 app = Flask(\_\_name\_\_)  
 app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///flaskdatabase.db'  
 app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False  
 CORS(app)  
 db.init\_app(app)  
 app.config['JWT\_SECRET\_KEY']='YOUR\_SECRET\_KEY'  
 jwt=JWTManager(app)  
  
 from cms.Blog.blog\_routes import blogs  
 app.register\_blueprint(blogs)  
  
 from cms.User.user\_model import User  
  
 from cms.Login.login\_route import login  
 app.register\_blueprint(login)  
  
 from cms.Tag.tag\_model import Tag  
  
   
 @click.command(name='create\_admin')   
 @with\_appcontext  
 def create\_admin():  
 admin=User(email="ANY\_EMAIL",password="ANY\_PASSWORD")  
 admin.password = generate\_password\_hash(admin.password,'sha256',salt\_length=12)  
 db.session.add(admin)  
 db.session.commit()  
  
 app.cli.add\_command(create\_admin)  
  
   
  
 return app

Once you’re done with everything, you simply need to run the below command to make the database working. So paste it on your terminal:

python  
from cms import db,create\_app  
db.create\_all(app=create\_app())

This will create a file flaskdatabase.db, which contains all the tables. Now to run the API server, use the command:

set FLASK\_APP=cms/\_\_init\_\_.py  
set FLASK\_DEBUG=1  
set FLASK\_ENV=development  
flask create\_admin  
flask run

And Bingo!!! You’re ready to make the API calls to create blog posts. You can clone ready-made code from [this repository](https://github.com/vyomsrivastava/cms-backend).

## Final Words

In this article, we have covered a lot of ground, like how to make SQL connections, how to run SQL queries using SQLalchemy, implementation of JWT, and more, all to generate a headless CMS. You can say that this is a kind of all-in-one article, which can help you to start picking up some advanced concepts in Python-Flask.