**Building a Full Stack App with Flask, React, MySQL**

[[Sandyjtech](https://medium.com/@sandyjtech?source=post_page-----a78fcc235ff0--------------------------------)](https://medium.com/@sandyjtech?source=post_page-----a78fcc235ff0--------------------------------)

[Sandyjtech](https://medium.com/@sandyjtech?source=post_page-----a78fcc235ff0--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F162c03a82582&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40sandyjtech%2Fbuilding-a-full-stack-app-with-flask-react-mysql-a78fcc235ff0&user=Sandyjtech&userId=162c03a82582&source=post_page-162c03a82582----a78fcc235ff0---------------------post_header-----------)

6 min read

·

Sep 2

14

*Creating a full-stack app is a daunting task and I remember feeling overwhelmed and lost during my first attempt. However, after putting in a lot of hard work and sleepless nights, I was eventually able to complete the assignment. As a result, I have decided to create a tutorial on how to create a full-stack app, so that others can benefit from my experience.*



Welcome to my comprehensive tutorial on building a dynamic full-stack web application using Flask, React.js, and MySQL. By the end of this guide, you’ll have a solid foundation for creating powerful web applications from scratch.

**1. Setting Up Your Development Environment**

Before diving into development, ensure you have the necessary tools installed:

* [**Node.js**](https://nodejs.org/)**:** A JavaScript runtime for building the front end.
* [**Python**](https://www.python.org/)**:** Required for creating the backend with Flask.
* A code editor of your choice **(e.g.,**[**Visual Studio Code**](https://code.visualstudio.com/)**)**.

**2. Backend Setup with Flask**

Let’s begin by creating a RESTful API with Flask. Follow these steps:

**Project Setup:**

Create a new project directory and set up a virtual environment:

python -m venv venv  
source venv/bin/activate # Linux/Mac  
venv\Scripts\activate # Windows

**Install Flask:**

pip install Flask

Install Flask Extensions: Install necessary extensions for our app:

pip install Flask-RESTful Flask-SQLAlchemy

**3. Define Your Flask App Structure**

Organize your project structure:

your\_app/  
├── app/  
│ ├── models.py  
│ ├── resources.py  
│ ├── auth.py  
│ └── ...  
├── app.py  
└── venv/

**4. Model Structure**

Define your database models models.py using SQLAlchemy. For instance:

class User(db.Model):  
 id = db.Column(db.Integer, primary\_key=True)  
 username = db.Column(db.String(80), unique=True, nullable=False)  
 password = db.Column(db.String(120), nullable=False)  
 images = db.relationship('Image', backref='user')class Image(db.Model):  
 id = db.Column(db.Integer, primary\_key=True)  
 title = db.Column(db.String(200), nullable=False)  
 url = db.Column(db.String(200), nullable=False)  
 user\_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)

**5. Database Migrations and Seeding**

It's recommended to use Flask-Migrate to manage database migrations, and seeding is a useful technique for populating the database with initial data.

flask db init  
flask db revision --autogenerate -m "created new tables"  
flask db upgrade head

**6. RESTful API Design**

Define your routes using Flask-RESTful in app.py:

from flask import Flask, request, jsonify  
from flask\_restful import Api, Resource, reqparse  
from flask\_sqlalchemy import SQLAlchemy

app = Flask(\_\_name\_\_)  
app.config['SQLALCHEMY\_DATABASE\_URI'] = 'mysql://username:password@localhost/db\_name'  
db = SQLAlchemy(app)  
api = Api(app)# Define resources here (Users, UsersByID)if \_\_name\_\_ == '\_\_main\_\_':  
 app.run(debug=True)

**7. Handling Sessions and Cookies**

Session management and cookies play a crucial role in web applications to maintain user state across multiple requests. Here’s how to handle sessions and cookies in Flask:

from flask import Flask, request, jsonify, session

app = Flask(\_\_name\_\_)  
app.secret\_key = 'your\_secret\_key'@app.route('/login', methods=['POST'])  
def login():  
 data = request.json  
 username = data.get('username')  
 password = data.get('password') # Replace with your authentication logic  
 if username == 'user' and password == 'password':  
 session['username'] = username # Set a session variable  
 return jsonify(message='Login successful'), 200  
 else:  
 return jsonify(message='Login failed'), 401@app.route('/logout', methods=['POST'])  
def logout():  
 session.pop('username', None) # Clear the session variable  
 return jsonify(message='Logged out'), 200@app.route('/protected', methods=['GET'])  
def protected():  
 if 'username' in session:  
 return jsonify(message='Protected data'), 200  
 else:  
 return jsonify(message='Unauthorized access'), 401if \_\_name\_\_ == '\_\_main\_\_':  
 app.run(debug=True)

**8. Authentication and Authorization**

Authentication verifies the identity of a user, while authorization determines what a user is allowed to do within the application. Here’s an example of authentication and authorization in Flask:

from flask import Flask, jsonify, session, request  
from flask\_restful import Api, Resource  
from flask\_bcrypt import Bcrypt  
  
app = Flask(\_\_name\_\_)  
app.secret\_key = 'your\_secret\_key'  
bcrypt = Bcrypt(app)  
api = Api(app)  
  
# Mock user data for testing (replace with your database logic)  
users = {  
 'user': {  
 'password\_hash': bcrypt.generate\_password\_hash('password').decode('utf-8'),  
 'id': 1  
 }  
}  
  
class AuthResource(Resource):  
 def post(self):  
 data = request.get\_json()  
 username = data.get('username')  
 password = data.get('password')  
  
 if username in users and bcrypt.check\_password\_hash(users[username]['password\_hash'], password):  
 session['user\_id'] = users[username]['id']  
 return jsonify(message='Login successful'), 200  
 else:  
 return jsonify(message='Login failed'), 401  
  
 def delete(self):  
 if 'user\_id' in session:  
 session.pop('user\_id', None)  
 return jsonify(message='Logged out'), 200  
 else:  
 return jsonify(message='No active session'), 401  
  
 def get(self):  
 if 'user\_id' in session:  
 return jsonify(message='Protected data'), 200  
 else:  
 return jsonify(message='Unauthorized access'), 401  
  
api.add\_resource(AuthResource, '/auth')  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 app.run(debug=True)

**9. Password Management and Hashing**

Storing passwords securely is crucial to safeguard user data. Here’s how you might use bcrypt to hash passwords in Flask:

from flask import Flask, jsonify, request  
from flask\_restful import Api, Resource  
from flask\_bcrypt import Bcrypt  
  
app = Flask(\_\_name\_\_)  
bcrypt = Bcrypt(app)  
api = Api(app)  
  
# Mock user data for testing (replace with your database logic)  
users = {}  
  
class RegisterResource(Resource):  
 def post(self):  
 data = request.get\_json()  
 username = data.get('username')  
 password = data.get('password')  
  
 if username in users:  
 return jsonify(message='Username already exists'), 400  
  
 hashed\_password = bcrypt.generate\_password\_hash(password).decode('utf-8')  
 users[username] = {'password\_hash': hashed\_password}  
  
 return jsonify(message='User registered'), 201  
  
class LoginResource(Resource):  
 def post(self):  
 data = request.get\_json()  
 username = data.get('username')  
 password = data.get('password')  
  
 if username not in users:  
 return jsonify(message='User not found'), 404  
  
 hashed\_password = users[username]['password\_hash']  
  
 if bcrypt.check\_password\_hash(hashed\_password, password):  
 return jsonify(message='Login successful'), 200  
 else:  
 return jsonify(message='Login failed'), 401  
  
api.add\_resource(RegisterResource, '/register')  
api.add\_resource(LoginResource, '/login')  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 app.run(debug=True)

**10. Frontend Setup with React.js**

Setting up your React app:

npm install -g create-react-app  
npx create-react-app my-app  
cd my-app

**11. Form Handling with Formik and Yup**

Install Formik and Yup:

npm install formik yup

Create a form with Formik and Yup in LoginForm.js:

import React from 'react';  
import { useFormik } from 'formik';  
import \* as Yup from 'yup';

const LoginForm = () => {  
 const formik = useFormik({  
 initialValues: {  
 username: '',  
 password: '',  
 },  
 validationSchema: Yup.object({  
 username: Yup.string().required('Username is required'),  
 password: Yup.string().required('Password is required'),  
 }),  
 onSubmit: (values) => {  
 // Handle form submission and API requests here  
 },  
 }); return (  
 <div>  
 <form onSubmit={formik.handleSubmit}>  
 {/\* Form fields and validation messages \*/}  
 </form>  
 </div>  
 );  
};export default LoginForm;

**12. Connecting Frontend and Backend**

In this section, I have simplified the process of connecting your Flask backend with your React frontend and provided a basic example:

**1. Build React Production Build:**

First, create a production build of your React app by running the following command in your client directory:

npm run build --prefix client

This will generate optimized static HTML, JavaScript, and CSS files in the client/build directory.

**2. Update Flask for Static Files:**

In your Flask app.py file, configure Flask to serve static files from the React build directory:

from flask import Flask, render\_template

app = Flask(\_\_name\_\_, static\_url\_path='', static\_folder='client/build')@app.route('/')  
def index():  
 return render\_template('index.html')

This code sets up Flask to **serve**static files from the client/build directory and renders the React app on the root route /.

**3. Run Flask Server:**

Start your Flask server, and it will now serve your React app:

gunicorn --chdir server app:app

**4. Making API Requests:**

In your React components, you can make API requests to your Flask backend using the Fetch API or Axios. Here’s a simplified example using Fetch API:

import React, { useState, useEffect } from 'react';

function App() {  
 const [users, setUsers] = useState([]); useEffect(() => {  
 fetch('/api/users') // Replace with your Flask API endpoint  
 .then((response) => response.json())  
 .then((data) => setUsers(data))  
 .catch((error) => console.error(error));  
 }, []); return (  
 <div>  
 <h1>User List</h1>  
 <ul>  
 {users.map((user) => (  
 <li key={user.id}>{user.username}</li>  
 ))}  
 </ul>  
 </div>  
 );  
}export default App;

Replace '/api/users' with the actual URL of your Flask backend endpoint.

**13. Deployment (Simplified):**

To deploy your Flask-React app, you can use platforms like Render, Vercel, or Heroku:

* ***Render:*** Create a web service on Render, connect your GitHub repository, and configure build and start commands as mentioned in your blog post.
* ***Vercel:*** For the React front-end, connect your GitHub repository with Vercel, and it will automatically detect and build your React app. For the Flask backend, you can deploy it separately on platforms like Heroku.
* ***Heroku:*** Deploy your Flask backend on Heroku following Heroku’s deployment guide. For the React front-end, you can deploy it on Vercel or other suitable platforms.

With this simplified approach, you can effectively connect your Flask backend with your React frontend and deploy the application for public access.

**Conclusion**

Congratulations! You have successfully built a full-stack app using Flask, and React, and integrated advanced features such as migrations, seeding, cookies, sessions, Yup, and Formik. I wrote this tutorial with the hope of assisting you in creating a full-stack app. It took me some time to grasp these concepts, so I hope this tutorial will make the process easier for you.