




DZone > Database Zone > Flask 101: Adding, Editing, and Displaying Data

Flask 101: Adding, Editing, and Displaying Data

by Mike Driscoll  MVB · Dec. 15, 17 · Database Zone · Tutorial

Enable Your 'Work from Home' Workforce with VDI



Watch How to Enable a 'Work from Home' Workforce with Cloud-Based VDI to learn about different types of virtual desktop infrastructure and select the option that best meets your needs.

Last time, we learned how to add a search form to our music database application. Of course, we still haven't added any data to our database, so the search form doesn't actually do much of anything except tell us that it didn't find anything. In this tutorial, we will learn how to actually add data, display search results, and edit entries in the database.

Let's get started!

Adding Data to the Database

Let's start by coding up our new album form. Open up the `forms.py` file we created in the last tutorial and add the following class:

```
1 class AlbumForm(Form):
2     media_types = [('Digital', 'Digital'),
3                   ('CD', 'CD'),
4                   ('Cassette Tape', 'Cassette Tape')]
5
6     artist = StringField('Artist')
7     title = StringField('Title')
8     release_date = StringField('Release Date')
9     publisher = StringField('Publisher')
10    media_type = SelectField('Media', choices=media_types)
```

This defines all the fields we need to create a new `Album`. Now we need to open `main.py` and add a function to handle what happens when we want to create the new

album.

```

1 # main.py
2
3 from app import app
4 from db_setup import init_db, db_session
5 from forms import MusicSearchForm, AlbumForm
6 from flask import flash, render_template, request, redirect
7 from models import Album
8
9 init_db()
10
11
12 @app.route('/', methods=['GET', 'POST'])
13 def index():
14     search = MusicSearchForm(request.form)
15     if request.method == 'POST':
16         return search_results(search)
17
18     return render_template('index.html', form=search)
19
20
21 @app.route('/results')
22 def search_results(search):
23     results = []
24     search_string = search.data['search']
25
26     if search.data['search'] == '':
27         qry = db_session.query(Album)
28         results = qry.all()
29
30     if not results:
31         flash('No results found!')
32         return redirect('/')
33     else:
34         # display results
35         return render_template('results.html', table=table)
36
37
38 @app.route('/new_album', methods=['GET', 'POST'])
39 def new_album():
40     """
41     Add a new album
42     """
43     form = AlbumForm(request.form)
44     return render_template('new_album.html', form=form)
45
46
47 if __name__ == '__main__':
48     app.run()

```

Here we add an import to import our new form at the top and then we create a new function called `new_album()`. Then we create an instance of our new form and pass it to our

`render_template()` function which will render a file called `new_album.html`. Of course, this HTML file doesn't exist yet, so that will be the next thing we need to create. When you save this new HTML file, make sure you save it to the `templates` folder inside of your `musicdb` folder.

Once you have `new_album.html` created, add the following HTML to it:

```
1<doctype html>
2<title>New Album - Flask Music Database</title>
3<h2>New Album</h2>
4
5{% from "_formhelpers.html" import render_field %}
6<form method=post>
7    <dl>
8        {{ render_field(form.artist) }}
9        {{ render_field(form.title) }}
10       {{ render_field(form.release_date) }}
11       {{ render_field(form.publisher) }}
12       {{ render_field(form.media_type) }}
13    </dl>
14    <p><input type=submit value=Submit>
15</form>
```

This code will render each field in the form and it also creates a **Submit** button so we can save our changes. The last thing we need to do is update our `index.html` code so that it has a link that will load our new album page. Basically, all we need to do is add the following:

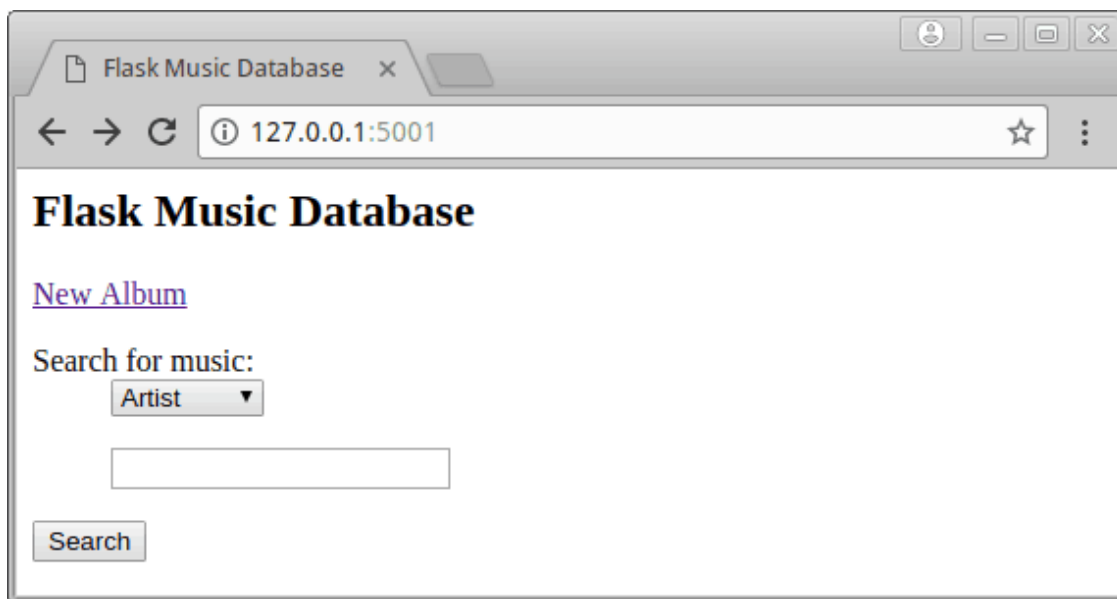
```
1<a href="{{ url_for('.new_album') }}"> New Album </a>
```

So the full change looks like this:

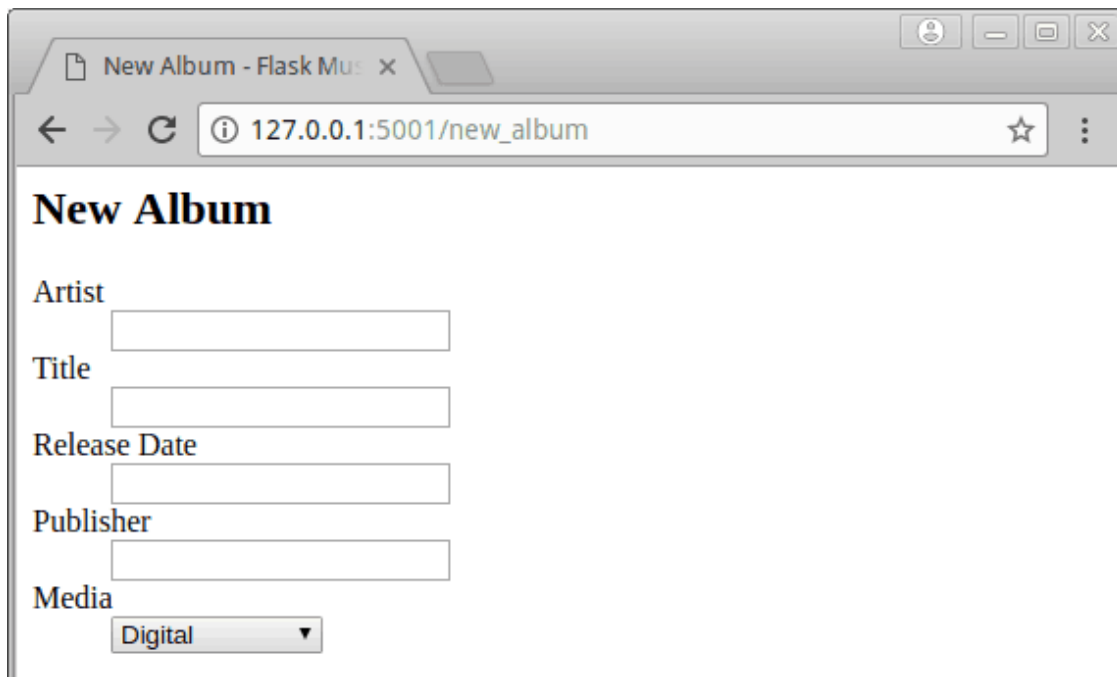
```
1<doctype html>
2<head>
3    <title>Flask Music Database</title>
4</head>
5
6<h2>Flask Music Database</h2>
7
8<p><p>
9<a href="{{ url_for('.new_album') }}"> New Album </a>
10
11{% with messages = get_flashed_messages() %}
12    {% if messages %}
13        <ul class=flashes>
14            {% for message in messages %}
15                <li>{{ message }}</li>
16            {% endfor %}
17        </ul>
18    ...
```

```
8 {% endif %}
9 {% endwith %}
0
1 {% from "_formhelpers.html" import render_field %}
2 <form method=post>
3   <dl>
4     {{ render_field(form.select) }}
5     <p>
6       {{ render_field(form.search) }}
7     </p>
8   <p><input type=submit value=Search>
9 </form>
```

Now if you load the main page of your web application, it should look like this:



If you click on the **New Album** link, then you should see something like this in your browser:





Now we have an ugly but functional new album form, but we didn't actually make the **Submit** button work. That is our next chore.

Saving Data

We need our new album form to save the data when the submit button is pushed. What happens when you press the **Submit** button, though? If you go back to the `new_album.html` file, you will note that we set the **form method** to `POST`. So we need to update the code in `main.py` so it does something on `POST`.

To do the save, we need to update the `new_album()` function in `main.py` so it ends up like this:

```
1@app.route('/new_album', methods=['GET', 'POST'])
2def new_album():
3    """
4    Add a new album
5    """
6    form = AlbumForm(request.form)
7
8    if request.method == 'POST' and form.validate():
9        # save the album
10       album = Album()
11       save_changes(album, form, new=True)
12       flash('Album created successfully!')
13       return redirect('/')
14
15    return render_template('new_album.html', form=form)
```

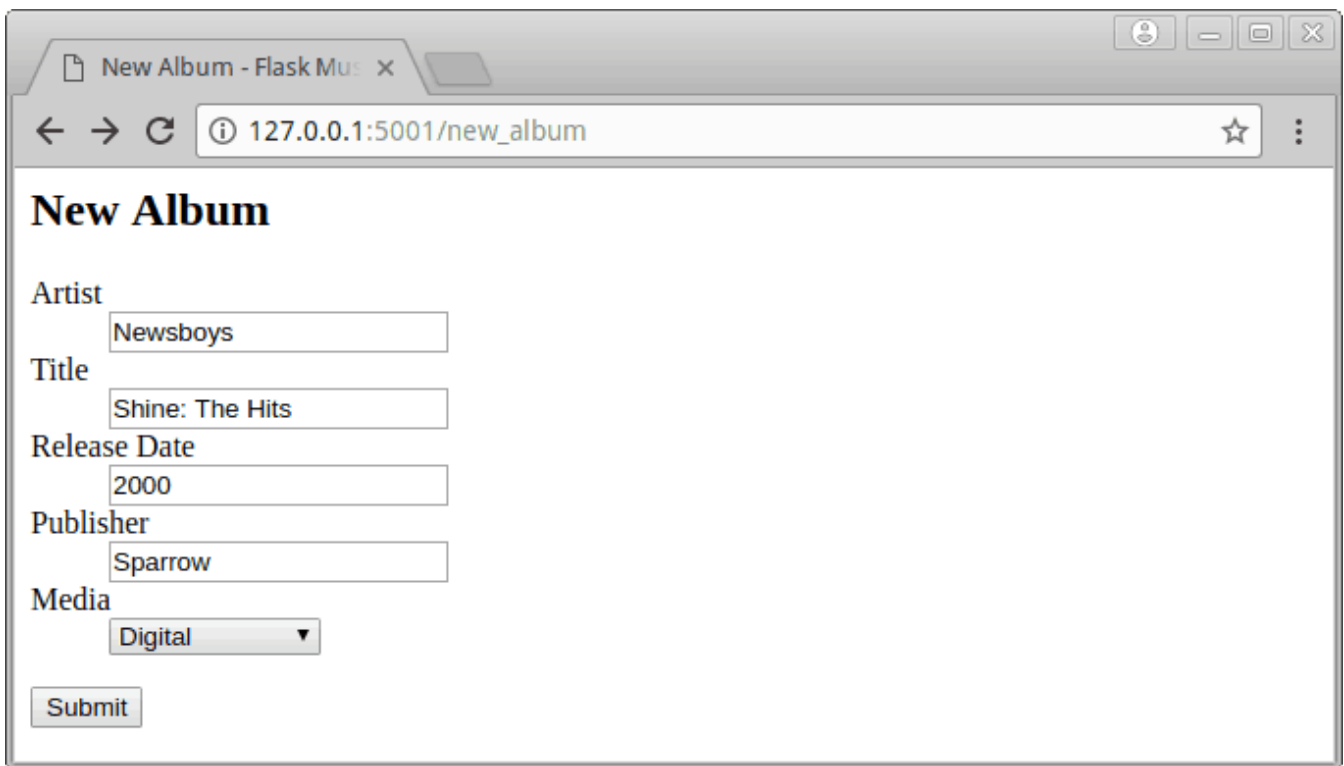
Now when we post, we create an `Album` instance and pass it to a `save_changes()` function along with the form object. We also pass along a flag that indicates if the item is new or not. We will go over why I added that last one later on in the article. For now though, we need to create the `save_changes()` function. Save the following code in the `main.py` script.

```
1def save_changes(album, form, new=False):
2    """
3    Save the changes to the database
4    """
5    # Get data from form and assign it to the correct attributes
6    # of the SQLAlchemy table object
7    artist = Artist()
8    artist.name = form.artist.data
9
10   album.artist = artist
```

```
1 album.title = form.title.data
2 album.release_date = form.release_date.data
3 album.publisher = form.publisher.data
4 album.media_type = form.media_type.data
5
6 if new:
7     # Add the new album to the database
8     db_session.add(album)
9
10 # commit the data to the database
11 db_session.commit()
```

Here, we extract the data from the form and assign it to the album object's attributes accordingly. You will also notice that we need to create an `Artist` instance to actually add the artist to the album correctly. If you don't do this, you will get a SQLAlchemy-related error. The `new` parameter is used here to add a new record to the database.

Here is a session I did to test it out:



The screenshot shows a web browser window with the title 'New Album - Flask Mus'. The address bar shows '127.0.0.1:5001/new_album'. The page content is titled 'New Album' and contains a form with the following fields:

- Artist:
- Title:
- Release Date:
- Publisher:
- Media:

A 'Submit' button is located at the bottom left of the form.

Once the item saves, it should take you back to the homepage of the website. One thing to note is that I don't do any checking in the database to prevent the user from saving an entry multiple times. This is something you can add yourself if you feel like taking on the challenge. Anyway, while I was testing this out, I submitted the same entry a few times, so when I do a search I should end up with multiple entries for the same item. If you try doing a search now, though, you will end up with an error because we haven't created the results page yet.

Let's do that next!

Displaying Search Results

I prefer having tabulated results, which requires using a table. Rather than messing around with HTML table elements, you can download yet another Flask extension called Flask Table. To install it, just use `pip` like this:

```
1 pip install flask_table
```

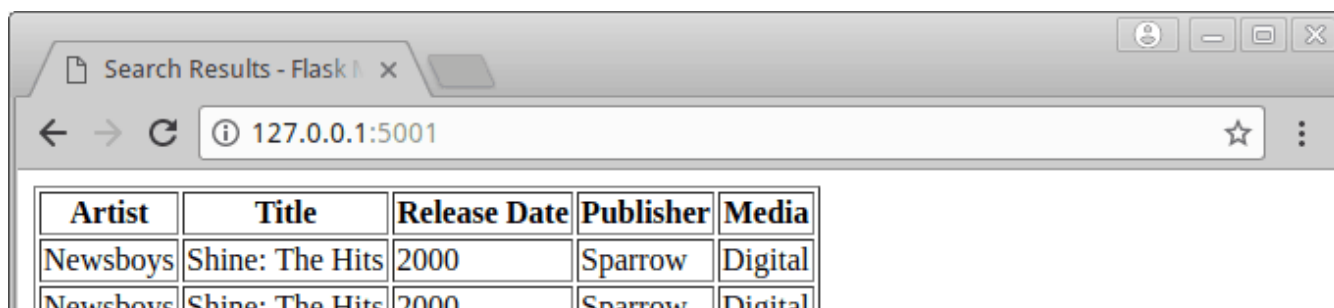
Now that we have Flask Table installed, we need to create a table definition. Let's create a file that we will call `tables.py` and save it in our `musicdb` folder. Open that up in your editor and add the following code:

```
1 from flask_table import Table, Col
2
3 class Results(Table):
4     id = Col('Id', show=False)
5     artist = Col('Artist')
6     title = Col('Title')
7     release_date = Col('Release Date')
8     publisher = Col('Publisher')
9     media_type = Col('Media')
```

When you define the table class, you will want to make the class attributes the same name as those in the object that will be passed to it. In this case, I used the attributes from the `Album` class here. Now, we just need to create a `results.html` file and save it to the `templates` folder. Here is what should go in that file:

```
1 <doctype html>
2 <title>Search Results - Flask Music Database</title>
3 {{ table }}
```

As you can see, all we needed to do was add a `title` element, which is actually optional, and add a table object in Jinja. Now when you run the search with an empty string, you should see something like this:



The screenshot shows a web browser window with the title "Search Results - Flask Music Database". The address bar shows the URL "127.0.0.1:5001". The main content area displays a table with the following data:

Artist	Title	Release Date	Publisher	Media
Newsboys	Shine: The Hits	2000	Sparrow	Digital
Newsboys	Shine: The Hits	2000	Sparrow	Digital

Newsboys	Shine: The Hits	2000	Sparrow	Digital
Newsboys	Shine: The Hits	2000	Sparrow	Digital

Yes, it's pretty plain, but it works and you can now see everything in your database.

Editing Data in the Database

The last item that we need to cover is how to edit the data in the database. One of the easiest ways to do this would be to search for an item and add a way for the user to edit the items that were found. Open up the `tables.py` file and add on a `LinkCol`:

```
1 from flask_table import Table, Col, LinkCol
2
3 class Results(Table):
4     id = Col('Id', show=False)
5     artist = Col('Artist')
6     title = Col('Title')
7     release_date = Col('Release Date')
8     publisher = Col('Publisher')
9     media_type = Col('Media')
10    edit = LinkCol('Edit', 'edit', url_kwargs=dict(id='id'))
```

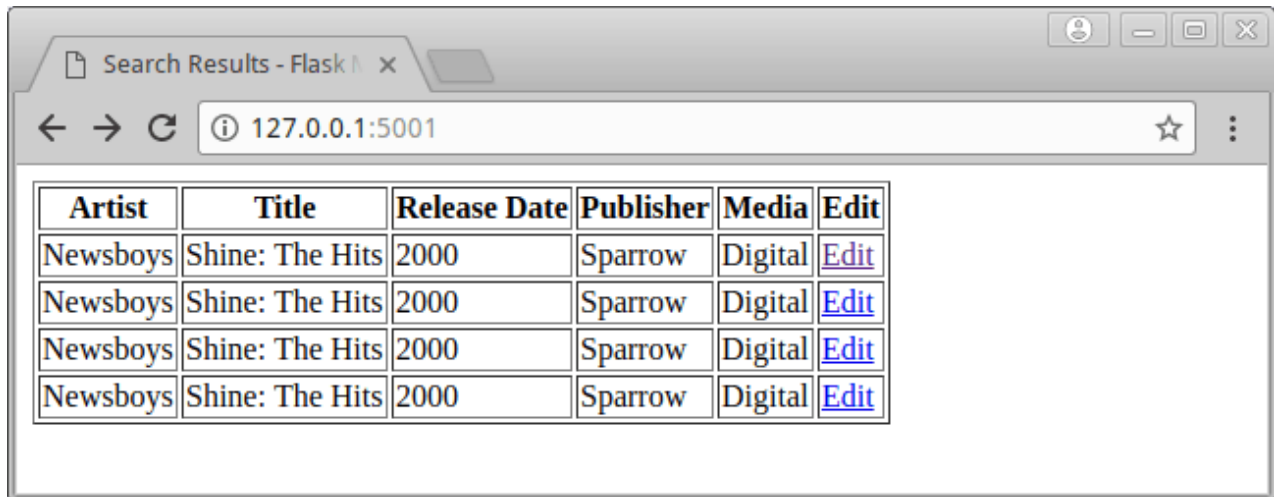
The `LinkCol` takes the column name as a string along with what the endpoint should be. The endpoint is the function that will be called when the link is clicked. We also pass along the entry's id so we can look it up in the database (i.e. the `url_kwargs` argument). Now, we need to update our `main.py` file with a function called `edit()`:

```
1 @app.route('/item/<int:id>', methods=['GET', 'POST'])
2 def edit(id):
3     qry = db_session.query(Album).filter(
4         Album.id==id)
5     album = qry.first()
6
7     if album:
8         form = AlbumForm(formdata=request.form, obj=album)
9         if request.method == 'POST' and form.validate():
10             # save edits
11             save_changes(album, form)
12             flash('Album updated successfully!')
13             return redirect('/')
14         return render_template('edit_album.html', form=form)
15     else:
16         return 'Error loading #{id}'.format(id=id)
```

The first item to take note of here is that we have a custom route set up for the URL that uses

the ID we pass to it to create a unique URL. Next, we do a database search for the ID in question. If we find the ID, then we can create our form using the same form we created earlier. However this time we pass it the album object so the form gets pre-filled with data so we have something to edit. If the user presses the **Submit** button on this page, then it will save the entry to the database and flash a message to the user to that effect. If we pass in a bad ID, then a message will be shown to the user.

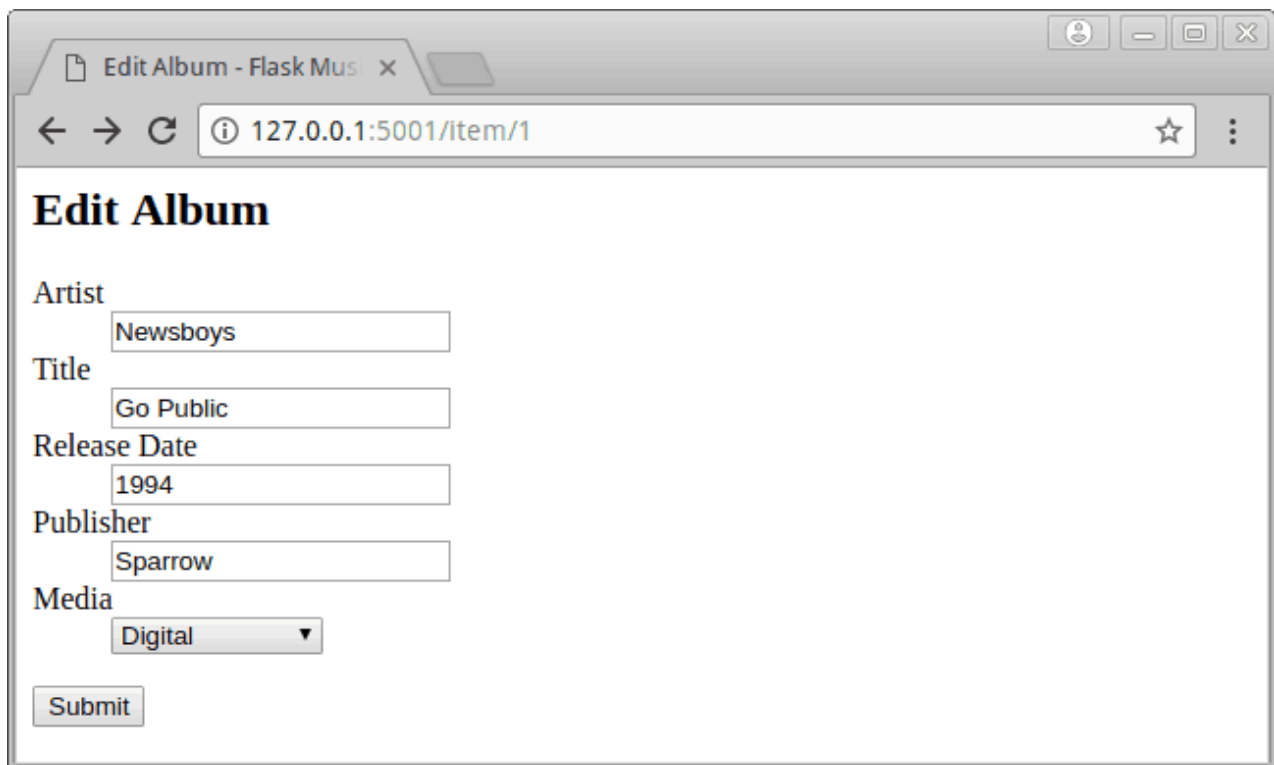
Now when we run the empty search from earlier you should see this:



The screenshot shows a web browser window titled 'Search Results - Flask'. The address bar displays '127.0.0.1:5001'. The main content area contains a table with the following data:

Artist	Title	Release Date	Publisher	Media	Edit
Newsboys	Shine: The Hits	2000	Sparrow	Digital	Edit
Newsboys	Shine: The Hits	2000	Sparrow	Digital	Edit
Newsboys	Shine: The Hits	2000	Sparrow	Digital	Edit
Newsboys	Shine: The Hits	2000	Sparrow	Digital	Edit

Let's click on the first row's edit link:



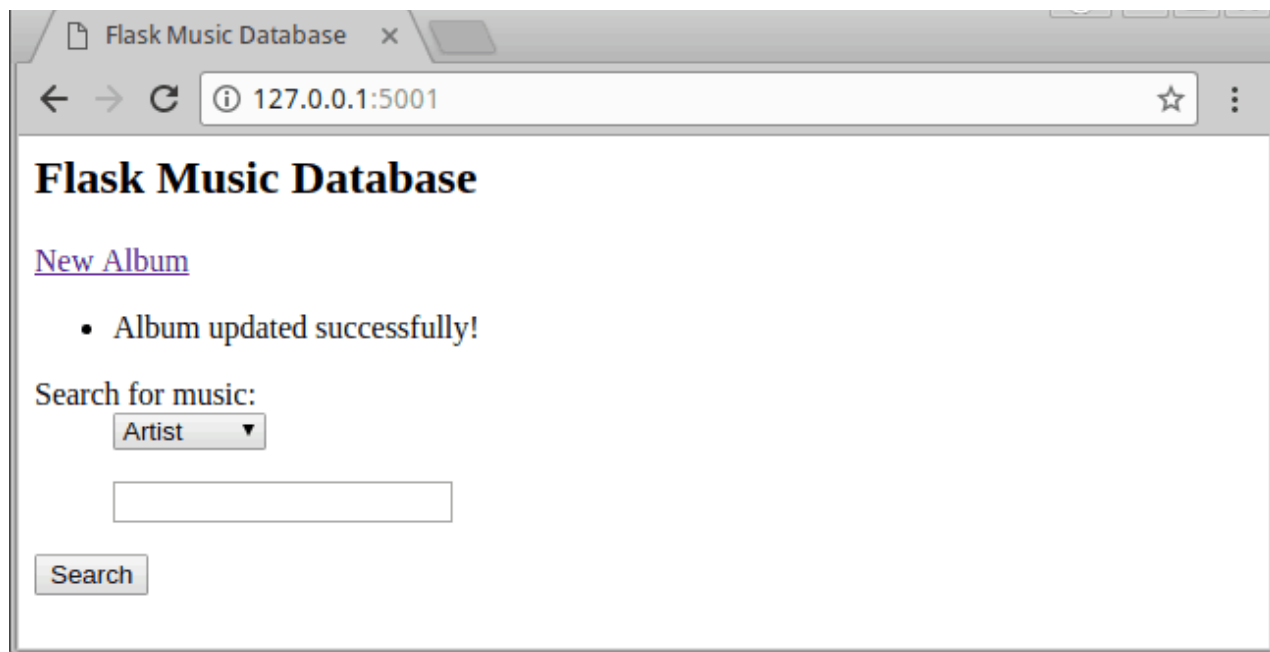
The screenshot shows a web browser window titled 'Edit Album - Flask Mus'. The address bar displays '127.0.0.1:5001/item/1'. The main content area contains a form titled 'Edit Album' with the following fields:

- Artist: Newsboys
- Title: Go Public
- Release Date: 1994
- Publisher: Sparrow
- Media: Digital (dropdown menu)

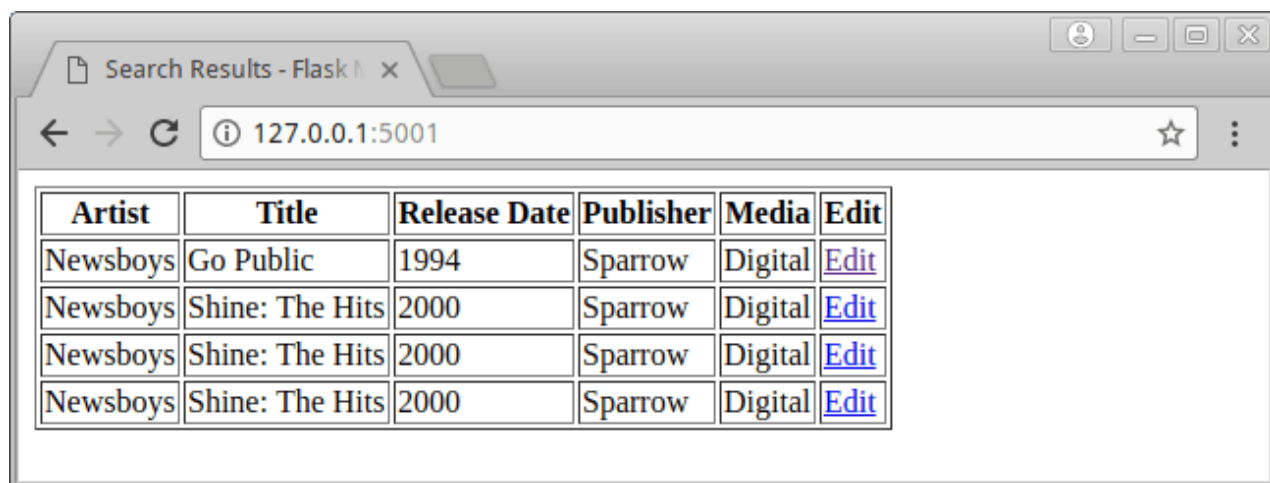
At the bottom of the form is a 'Submit' button.

Here, I edit most of the fields on the page. Then I click **Submit** and get this:





Supposedly, the entry was updated with my changes. To verify, try running another empty search:



That looks right, so now we have the editing functionality complete!

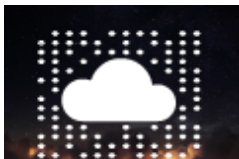
Wrapping Up

At this point, you should be able to add entries to the database, display all the entries, and edit said entries. The main item missing is how to filter the search results so that it actually looks for the search term you want instead of always returning everything in the database. We should probably also add the ability to delete an item from the database. Those are the topics we will look at in the next article. For now, have fun and happy coding!

Download Code

Download a tarball of the code from this article.

Immediate Cloud Gains Pack a Long-Term Punch



Read the Business Impact Brief by 45° learn more about the benefits of moving and how short-term cloud initiatives can help organizations develop a solid foundation for

Like This Article? Read More From DZone



related
article
thumbnail

DZone Article

Visualizing Atmospheric Carbon Dioxide



related
article
thumbnail

DZone Article

Squirrel Perimeter and the Isoparametric Problem With Python



related
article
thumbnail

DZone Article

Plotly Dash and OmniSciDB for Real-Time Data Visualization



related
refcard
thumbnail

Free DZone Refcard

Hybrid Relational/JSON Data Modeling and Querying

Topics: DATA VISUALIZATION , DATABASE , FLASK , PYTHON , TUTORIAL

Published at DZone with permission of Mike Driscoll , DZone MVB. [See the original article here.](#)

Opinions expressed by DZone contributors are their own.

ABOUT US

[About DZone](#)
[Send feedback](#)
[Careers](#)

ADVERTISE

[Developer Marketing Blog](#)
[Advertise with DZone](#)
[+1 \(919\) 238-7100](#)

CONTRIBUTE ON DZONE

[MVB Program](#)
[Zone Leader Program](#)
[Become a Contributor](#)
[Visit the Writers' Zone](#)

LEGAL

[Terms of Service](#)
[Privacy Policy](#)

CONTACT US

600 Park Offices Drive

Suite 150

Research Triangle Park, NC 27709

support@dzzone.com

+1 (919) 678-0300

Let's be friends:    

DZone.com is powered by  AnswerHub logo