BUILD A LIVE TABLE USING FLASK

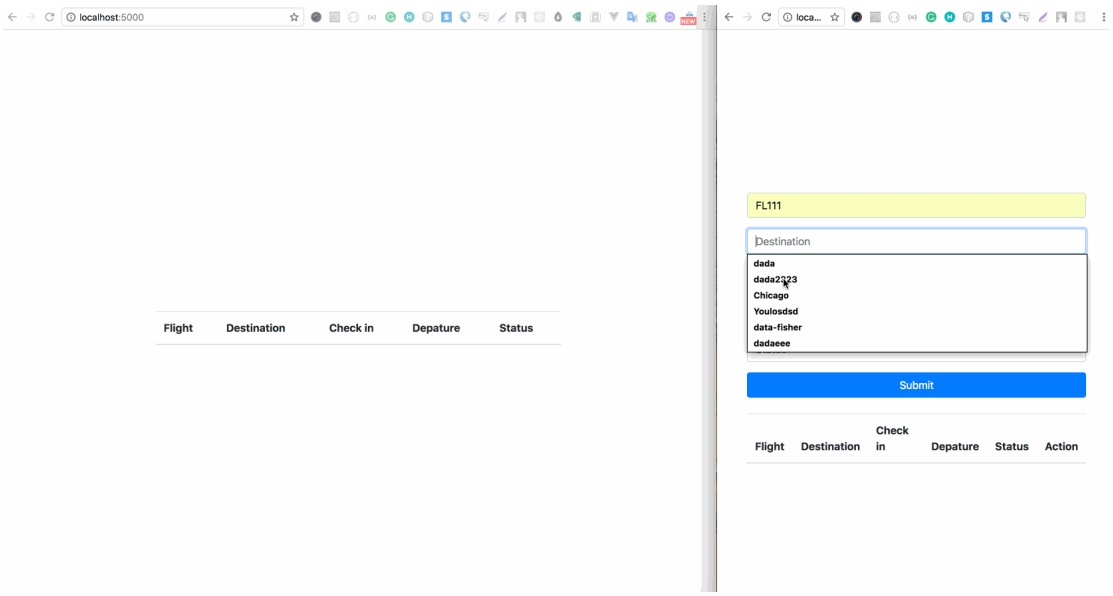
Gideon Onwuka August 27th, 2018

> You will need Python 3+ installed on your machine.

In this tutorial, you will learn how to build a realtime table using Python(Flask) and Pusher Channels. In the process, we are going to build a realtime flights status app.

If you have ever used a web application where you have to refresh the page constantly to check for updates, then you have faced this problem before. At times you miss out on some important update either because you were tired of refreshing the page or you did not refresh the page on time. Let's see how we can address this problem using Pusher Channels.

Here is a preview of what we'll be building:



PREREQUISITES

This tutorial uses the following:

- Python 3+ (If you are a Windows user, you can follow this guide to install it. For Mac/Linux users, you can install it by following the guide here or here respectively if you don't have it installed already)
- JavaScript(jQuery)
- Channels. Channels will be responsible for the real-time feature.

Make sure you have the right Python version installed by typing the below in your command line:

```
python --version
```

Or:

```
python3 --version
```

If the command prints something similar to `Python 3.*.*` or higher, then we are good to go.

A basic understanding of Python 3 and JavaScript will be helpful to follow along with this tutorial, although it's not entirely required. Let's get started.

GETTING OUR PUSHER API KEY

Channels does the job of adding realtime functionality to applications. We'll use it to add realtime functionality to the app. But before we can start using the platform, we need an API key.

Head over to Pusher and log inhere or sign up (if you don't have an account already).

Once you are logged in, create a new app then note down your Pusher `app_id`, `key`, `secret` and `cluster`. We'll need them later.

SETTING UP THE APPLICATION

Now, let's create a simple structure for the application.

Create the below files and folders in any convenient location on your system:

```
realtime-table
├── .env
├── .flaskenv
├── app.py
├── database.py
├── models.py
├── requirements.txt
├── static
│   ├── custom.js
│   └── style.css
└── templates
    └── index.html
    └── base.html
    └── backend.html
    └── update_flight.html
```

If you prefer the command prompt, you cab use the below commands on Mac or Linux to create the files and folders:

```
# Create folders
$ mkdir realtime-table && cd realtime-table && mkdir templates static

# Create files
$ touch app.py database.py models.py static/{custom.js,style.css} templates/{index.html,base.html,update_flight.html,backend.html} requirements.txt .flaskenv .env
```

CREATING A VIRTUAL ENVIRONMENT

It's a good idea to have an isolated environment when working with Python. virtualenv is a tool to create an isolated Python environment. It creates a folder which contains all the necessary executables to use the packages that a Python project would need.

From your command line, change your directory to the project root folder - `realtime-table` - then execute the below command:

```
$ python3 -m venv env
```

Or:

```
$ python -m venv env
```

The command to use depends on which is associated with your python3 installation.

Finally, activate the virtual environment:

```
$ source env/bin/activate
```

If you are using Windows, activate the virtualenv with the below command:

```
> \path\to\env\Scripts\activate
```

This is meant to be a full path to the activate script. Replace `\path\to` with your correct path name.

Next add the Python libraries that we'll be using. Update `requirements.txt` with the code below:

```
Flask==1.0.2
python-dotenv==0.8.2
pusher==2.0.1
SQLAlchemy==1.2.0
Flask-SQLAlchemy==2.1
```

- python-dotenv: this library will be used by Flask to load environment configurations files.
- pusher: this is the Pusher Python library that makes it easy to interact with its API.
- Flask: the Python framework we are using to build the app.
- SQLALchemy: a Python SQL toolkit and Object Relational Mapper that gives application developers the full power and flexibility of SQL.

ADDING FLASK ENVIRONMENT CONFIGURATION

Next, add the following to the `.flaskenv` file:

```
FLASK_APP=app.py
FLASK_ENV=development
```

This will instruct Flask to use `app.py` as the main entry file and start up the project in development mode.

ADDING OUR PUSHER KEYS

Add the following to the `.env` file:

```
PUSHER_APP_ID=app_id
PUSHER_KEY=key
PUSHER_SECRET=secret
PUSHER_CLUSTER=cluster
```

Replace `app_id`, `key`, `secret` and `cluster` with your own Pusher keys which you have noted down earlier.

Next, create a Flask instance:

```
# app.py

from flask import Flask, request, jsonify, render_template, redirect
import os
import json
import pusher
from datetime import datetime

app = Flask(__name__)

@app.route('/')
def index():
    return render_template('index.html')

# run Flask app
if __name__ == "__main__":
    app.run()
```

Here, we created a route - `/` -, which will render the `index.html` file in the `templates` folder.

Now, install the libraries in `requirements.txt`:

```
pip install -r requirements.txt
```

Once the installation is complete, run the app:

```
flask run
```

If it is successful, your command output should be similar to the below:

```
 * Serving Flask app "app.py" (lazy loading)
 * Environment: development
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 ...
```

This shows that the app is now running at http://127.0.0.1:5000/. If you visit the URL you will get a blank page because there is no content in the `templates/index.html` file yet.

SETTING UP THE DATABASE

We'll need a database to persist our flight's data. We'll be using SQLite for this.

Add the following code to `database.py` file:

```
from sqlalchemy import create_engine
from sqlalchemy.orm import scoped_session, sessionmaker
from sqlalchemy.ext.declarative import declarative_base

engine = create_engine('sqlite:///database.db', convert_unicode=True)
db_session = scoped_session(sessionmaker(autocommit=False,
                                         autoflush=False,
                                         bind=engine))
Base = declarative_base()
Base.query = db_session.query_property()

def init_db():
    import models
    Base.metadata.create_all(bind=engine)
```

We are using SQLAlchemy to initialize our database connection.

In the `init_db()` function, we imported our models and finally call `Base.metadata.create_all` to create all the tables specified in the model's file.

Now add the code for our models to the `models.py` file:

```
from sqlalchemy import Column, Integer, String, DateTime
from database import Base
from datetime import datetime

class Flight(Base):
    __tablename__ = 'flights'
    id = Column(Integer, primary_key=True)
    flight = Column(String(50))
    destination = Column(String(120))
    check_in = Column(DateTime, default=datetime.utcnow)
    departure = Column(DateTime, default=datetime.utcnow)
    status = Column(String(120))

    def __init__(self, flight=None, destination=None, check_in=None, depature=None, status=None):
        self.flight = flight
        self.destination = destination
        self.check_in = check_in
        self.depature = depature
        self.status = status
```

```
    def __repr__(self):
        return '<Flight %r>' % (self.flight)
```

Here, we created a class named Flight which holds the schema for our flight's table. You must have noticed, in the class, we named our table 'flights' which has columns - id, flight, destination, check_in, departure, status.

Now that we have the schema for our flight's table, we can now create the table directly from the file.

Next, import the database helper package along with our Flight model to `app.py`:

```
# app.py
# [...]
from database import db_session
from models import Flight
# [...]
```

Finally, let's tell Flask to close the connection to the database as soon as an operation is complete. Add the following code to `app.py` right before the `if __name__ == "__main__"` line :

```
@app.teardown_appcontext
def shutdown_session(exception=None):
    db_session.remove()
```

CREATE THE DATABASE AND TABLES

Next, let's create the database and tables. Open up a new command window and change your directory to the project's root folder, then run the below commands:

⚠ Make sure to activate your virtual environment - `source env/bin/activate`. If your virtual environment is not active, you will get an import error while running the below code.

```
$ python
>>> from database import init_db
>>> init_db()
```

In the preceding command:

- The first command will open up the python interpreter.
- The second command will import the init_db function from the `databse.py` file.
- The last command will create our table as we have defined in the `models.py` file.

If the command is successful, you will see a new file named `database.db`.

BUILDING THE APP INTERFACES

We need a page to list the flights on the front-end and also other pages to add and update flights.

ADDING THE BASE LAYOUT

We'll be using a template inheritance approach to build our views which makes it possible to reuse the layouts instead of repeating some markup across pages.

Add the following markup to the `templates/base.html` file:

```
<!-- /templates/base.html  -->

<!doctype html>
<html lang="en">
  <head>
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    <!-- Bootstrap CSS -->
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.0/css/bootstrap.min.css">
    <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
    <title>Realtime Table</title>
  </head>
  <body>
    <div class="container h-100">
      <div class="row align-items-center h-100">
        <div class="col-md-8 col-sm-12 mx-auto">
          <div class="h-100 justify-content-center">
            {% block content %}  {% endblock %}
          </div>
        </div>
      </div>
    </div>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
    <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.0/js/bootstrap.min.js"></script>
    <script src="https://js.pusher.com/4.1/pusher.min.js"></script>
    <script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/moment.js/2.22.2/moment.min.js"></script>
    <script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/jquery-datetimepicker/2.5.20/jquery.datetimepicker.min.js"></script>
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/jquery-datetimepicker/2.5.20/jquery.datetimepicker.min.css" />
    <script src="{{ url_for('static', filename='custom.js')}}"></script>
  </body>
</html>
```

This will serve as the base layout for all our views. All other views will inherit from the base file.

In this file, we are importing the following libraries:

In this file, we are importing the following libraries:

- Bootstrap
- jQuery plugin for date and time picker
- Pusher JavaScript library
- jQuery
- Moment.js A JavaScript library that helps to Parse, validate, manipulate, and display dates and times.

FLIGHT LISTING PAGE

This will serve as the front page of the application. We'll list all flights that we'll add to the database on this page.

Add the following to the `templates/index.html` file:

```
<!-- /templates/index.html  -->

{% extends 'base.html' %}

{% block content %}
  <div class="flight-container" style="overflow: auto; max-height: 80vh">
    <div class="table-responsive">
      <table class="table" id="flights">
        <thead>
          <tr>
            <th scope="col">Flight</th>
            <th scope="col">Destination</th>
            <th scope="col">Check in</th>
            <th scope="col">Depature</th>
            <th scope="col">Status</th>
          </tr>
        </thead>
        <tbody>
        {% for flight in flights %}
          <tr id={{ flight.id }}>
            <th scope="row">{{ flight.flight }}</th>
            <td> {{flight.destination }} </td>
            <td> {{ flight.check_in }} </td>
            <td> {{ flight.depature }} </td>
            <td> {{ flight.status }} </td>
          </tr>
        {% endfor %}
        </tbody>
      </table>
    </div>
  </div>
{% endblock %}
```

The `{% for flight in flights %}` will loop through the flight's database that we'll send to this page and list them. An important thing to note here is, we are assigning each table row a unique ID - `<tr id={{ flight.id }}>` - so that we can easily track and update any each rows using the ID.

PAGE FOR ADDING NEW FLIGHT

Next, add the markup for adding a new flight. Add the following code to the `template/backend.html` file:

```
<!-- /templates/backend.html  -->

{% extends 'base.html' %}

{% block content %}
  <form method="POST" id="target">
    <div class="form-group">
      <input type="text" class="form-control" name="flight" placeholder="Flight">
    </div>
    <div class="form-group">
      <input type="text" class="form-control" name="destination" placeholder="Destination">
    </div>
    <div class="form-group">
      <input
        type="text"
        class="form-control datetime"
        name="check_in"
        placeholder="Check in"
      >
    </div>
    <div class="form-group">
      <input
        type="text"
        class="form-control datetime"
        name="depature"
        placeholder="Depature"
      >
    </div>
    <div class="form-group">
      <input type="text" class="form-control" name="status" placeholder="Status">
    </div>
    <button type="submit" class="btn btn-block btn-primary">Submit</button>
  </form>

  <br />

  <div class="table-responsive">
    <table class="table" id="flights">
      <thead>
        <tr>
          <th scope="col">Flight</th>
          <th scope="col">Destination</th>
          <th scope="col">Check in</th>
          <th scope="col">Depature</th>
          <th scope="col">Status</th>
```

```
                <th scope="col">Action</th>
              </tr>
            </thead>
            <tbody>
            {% for flight in flights %}
              <tr id={{ flight.id }}>
                <th scope="row">{{ flight.flight }}</th>
                <td> {{flight.destination }} </td>
                <td> {{ flight.check_in }} </td>
                <td> {{ flight.depature }} </td>
                <td> {{ flight.status }} </td>
                <td> <a href="/edit/{{ flight.id }}">Edit</a> </td>
              </tr>
            {% endfor %}
            </tbody>
        </table>
    </div>
{% endblock %}
```

In this page, we have added a form for adding new flight. Also at the bottom of the page, we'll list all the flight that is in the database so edit the flights.

UPDATING FLIGHTS

Add the markup for updating flight to `templates/update_flight.html`:

```
<!-- /templates/update_flight.html  -->

{% extends 'base.html' %}

{% block content %}
    <form method="POST" id="target">
        <div class="form-group">
            <input type="text" class="form-control" name="flight" value="{{ data.flight }}">
        </div>
        <div class="form-group">
            <input type="text" class="form-control" name="destination" value="{{ data.destination }}">
        </div>
        <div class="form-group">
            <input
              type="text"
              class="form-control datetime"
              name="check_in"
              value="{{ data.check_in }}"
            >
        </div>
        <div class="form-group">
            <input
              type="text"
              class="form-control datetime"
              name="depature"
              value="{{ data.depature }}"
            >
        </div>
        <div class="form-group">
            <input type="text" class="form-control" name="status" value="{{ data.status }}">
        </div>
        <button type="submit" class="btn btn-block btn-primary">Update</button>
    </form>
{% endblock %}
```

INITIALIZE THE JQUERY DATE AND TIME PICKER PLUGIN

Since some of our input forms will require a date and time input. Initialize the jQuery date and time picker plugin by adding the following code to the `static/custom.js` file:

```
// /static/custom.js

$.datetimepicker.setDateFormatter({
    parseDate: function (date, format) {
        var d = moment(date, format);
        return d.isValid() ? d.toDate() : false;
    },
    formatDate: function (date, format) {
        return moment(date).format(format);
    },
});

$('.datetime').datetimepicker({
    format:'DD-MM-YYYY hh:mm A',
    formatTime:'hh:mm A',
    formatDate:'DD-MM-YYYY',
    useCurrent: false,
});
```

Finally add our styling to `static/style.css`:

```
/* /static/style.css  */

body,html {
    height: 100%;
}

.flight-container {
    margin: 0px;
    padding: px;
}
```

And now we have all our user interfaces ready. Next, we'll query the database for flights and list them to the page.

LISTING, ADDING AND EDITING A FLIGHT

LIST FLIGHTS TO THE FRONT PAGE

Update the `/` route to query the database and return flights to the page:

```
# app.py

@app.route('/')
def index():
    flights = Flight.query.all()
    return render_template('index.html', flights=flights)
```

Here we fetch all the flight in the database and pass it down to the `template/index.html` view.

If you visit the index page again - http://localhost:5000/- you will see an empty table:

Flight	Destination	Check in	Depature	Status

ADDING FLIGHTS

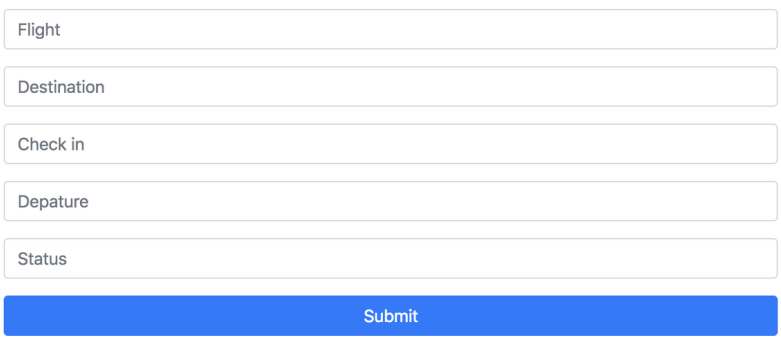
Next, add the following code to `app.py` file so we can add a new flight to the database:

```
# app.py

# [...]

@app.route('/backend', methods=["POST", "GET"])
def backend():
    if request.method == "POST":
        flight = request.form["flight"]
        destination = request.form["destination"]
        check_in = datetime.strptime(request.form['check_in'], '%d-%m-%Y %H:%M %p')
        depature = datetime.strptime(request.form['depature'], '%d-%m-%Y %H:%M %p')
        status = request.form["status"]
        new_flight = Flight(flight, destination, depature, check_in, status)
        db_session.add(new_flight)
        db_session.commit()

        return redirect("/backend", code=302)
    else:
        flights = Flight.query.all()
        return render_template('backend.html', flights=flights)

# [...]
```

In the preceding code:

- First we created a new route named `/backend` using `@app.route('/backend', methods=["POST", "GET"])` which accepts both GET and POST requests.
- Next, we check if the request method is a POST or a GET method so we know which block of code to execute.
- If it's a POST request, we'll grab all the data from the request. Then with the data, we'll add a new flight to the database and finally redirect to the `/backend` route.
- Else, if the request is a GET method, we'll fetch all the flight details from the database then send the data to the `database.html` view.

Now, we have a route - http://127.0.0.1:5000/backend for adding new flight:



Flight	Destination	Check in	Depature	Status	Action

UPDATING FLIGHTS

Next, add the following code to the `app.py` file for updating a flight record:

```
# app.py

# [...]

@app.route('/edit/<int:id>', methods=["POST", "GET"])
def update_record(id):
    if request.method == "POST":
        flight = request.form["flight"]
        destination = request.form["destination"]
        check_in = datetime.strptime(request.form['check_in'], '%d-%m-%Y %H:%M %p')
        depature = datetime.strptime(request.form['depature'], '%d-%m-%Y %H:%M %p')
        status = request.form["status"]

        update_flight = Flight.query.get(id)
        update_flight.flight = flight
        update_flight.destination = destination
        update_flight.check_in = check_in
        update_flight.depature = depature
        update_flight.status = status
        db_session.commit()

        return redirect("/backend", code=302)
    else:
        new_flight = Flight.query.get(id)
        new_flight.check_in = new_flight.check_in.strftime("%d-%m-%Y %H:%M %p")
        new_flight.depature = new_flight.depature.strftime("%d-%m-%Y %H:%M %p")

        return render_template('update_flight.html', data=new_flight)

# [...]
```

In the preceding code:

- First we created a new route named `/edit/<int:id>` using `@app.route('/edit/<int:id>', methods=["POST", "GET"])` which accepts both GET and POST requests. `<int:id>` is the flight ID that we want to update.
- Next, we checked if the request method is a POST or a GET method so we know which block of code to execute.
- If it's a POST request, we'll then grab all the input from the request. Then using the ID of the flight passed from the route, we'll query the database for the flight record then update the particular record and finally render the `backend.html` view.
- Else, if the request is a GET method, we'll fetch the flight of the ID passed from the route from the database then send the data to the `update_record.html`.

Now, we have a route - http://127.0.0.1:5000/edit/ for adding new flight.

ADDING REALTIME FUNCTIONALITY

Now we can add and update a flight but all these can not be seen in realtime by other users. Let's make it realtime.

Channels is a pub/sub messaging service. We'll subscribe to a channel, then on the channel, we'll trigger and listen to events.

To start talking to Channels, we need to download the Python library and Javascript library. The python library will help us trigger events from our python code when we add or update a flight. With the Javascript code, we'll listen for these events. When an event is fired, we'll take action and update the page accordingly.

What we'll be doing is, when we add a new flight, we'll trigger an event named `new-record`. When we update a record, we will trigger another event named `update-record`. Then from our view file, we'll listen to these events. When there is a `new-record` event, we'll add the flight to the page in realtime, and when there is an `update-record` event, we'll update the fight detail on the page in realtime.

INITIALIZE CHANNELS PYTHON LIBRARY

Initialize Pusher's Python library by adding the following code to the `app.py` file right after the `app = Flask(__name__)` line of code:

```
# app.py

# [...]

pusher_client = pusher.Pusher(
    app_id=os.getenv('PUSHER_APP_ID'),
    key=os.getenv('PUSHER_KEY'),
    secret=os.getenv('PUSHER_SECRET'),
    cluster=os.getenv('PUSHER_CLUSTER'),
    ssl=True)

# [...]
```

TRIGGERING THE NEW-RECORD EVENT

Now, let's trigger an event when we add new flight record. Add the following code to the `backend()` function block in `app.py` after the `db_session.commit()` line:

```
# app.py

# [...]

        data = {
            "id": new_flight.id,
            "flight": flight,
            "destination": destination,
            "check_in": request.form['check_in'],
            "depature": request.form['depature'],
            "status": status}

        pusher_client.trigger('table', 'new-record', {'data': data })

# [...]
```

Here,

- The `data` variable holds the new record we just added to the database.
- The `pusher_client.trigger` is a method for triggering events. The first parameter holds the channel name, the second parameter holds the event's name while the third parameter holds the data we are passing along.
- Then using pusher_client.trigger, we trigger an event `update-record` on the table's channel.

TRIGGERING THE UPDATE-RECORD EVENT

Next, trigger an event when we update a record.

Add the following code to `app.py` in the `update_record()` function after `db_session.commit()` line:

```
# app.py

# [...]

        data = {
            "id": id,
            "flight": flight,
            "destination": destination,
            "check_in": request.form['check_in'],
            "depature": request.form['depature'],
            "status": status}

        pusher_client.trigger('table', 'update-record', {'data': data })

# [...]
```

SHOW FLIGHT IN REALTIME WHEN THEY'RE ADDED

Now, let's listen to the new flight event and show the data on the homepage.

Add the following code to the `static/custom.js` file:

```
// /static/custom.js

// Initialize Pusher
const pusher = new Pusher('<PUSHER_KEY>', {
    cluster: '<CLUSTER>',
    encrypted: true
});

// Subscribe to table channel
var channel = pusher.subscribe('table');
```

First, we initialized the Pusher JavaScript library then we subscribe to a channel named `table`. It's on this channel that we'll listen to events and act on it as soon as they happen. Make sure to replace `<PUSHER_KEY>` and `<CLUSTER>` placeholders with your actual Pusher keys.

Next, listen to the event and show the data. Add the following code to `static/custom.js`:

```
// /static/custom.js

channel.bind('new-record', (data) => {
    const check_in = moment(`${data.data.check_in}`, 'DD/MM/YYYY hh:mm a').format('YYYY-MM-DD hh:mm:ss a')
    const depature = moment(`${data.data.depature}`, 'DD/MM/YYYY hh:mm a').format('YYYY-MM-DD hh:mm:ss a')

  $('#flights').append(`
      <tr id="${data.data.id} ">
          <th scope="row"> ${data.data.flight} </th>
          <td> ${data.data.destination} </td>
          <td> ${check_in} </td>
          <td> ${depature} </td>
          <td> ${data.data.status} </td>
      </tr>
  `)
});
```

SHOW REALTIME UPDATE OF FLIGHTS

Now, let's listen for the `update-record` event. When there is an update, we'll update the page accordingly.

Add the following code to `static/custom.js`:

```
// /static/custom.js

channel.bind('update-record', (data) => {
    const check_in = moment(`${data.data.check_in}`, 'DD/MM/YYYY hh:mm a').format('YYYY-MM-DD hh:mm:ss a')
```

```
    const depature = moment(`${data.data.depature}`, 'DD/MM/YYYY hh:mm a').format('YYYY-MM-DD hh:mm:ss a')

    $(`#${data.data.id}`).empty()

    $(`#${data.data.id}`).html(`
        <th scope="row"> ${data.data.flight} </th>
        <td> ${data.data.destination} </td>
        <td> ${check_in} </td>
        <td> ${depature} </td>
        <td> ${data.data.status} </td>
    `)
});
```

When there is an `update-record event`, we will grab the ID of the updated record. Then using the ID we'll update that particular row using `$(#${data.data.id}).html(`.

Well done! To test what we have built,

- Restart the server by pressing `CTRL+C` and then type `flask run`.
- Open the index page - localhost:5000/.
- Then, open the backend in another tab - localhost:5000/backend.
- Next, add or update a flight. You would see the changes appear in realtime on the index page.

CONCLUSION

In this tutorial, you have learned how to build a realtime table using a real-world example. You have also learned how to setup Channels for adding realtime functionality to web applications.

Although we are using Flask, this same approach can be applied to any other Python project or Python frameworks. You can get the full source code for this tutorial on GitHub.

FLASK LIVE TABLE PYTHON JAVASCRIPT CHANNELS