1. Build backend/API first: in this case a CRUD backend (create, remove, update, delete) for contact management. What that involves is setting up different routes or endpoints that can be ‘called’ from any type of frontend or back end. In this way we create this protocol or app programming interface that now defines a set of operations that our application and incorporate it into any frontend or built an app with it or anything we want to call up these different methods.
2. Frontend

**BACKEND**

1. **Setup work environment**

* Instal Node.js if it isn’t installed yet on your machine
* Create folders if needed inside project folder (ie. Backend)
* Create a new React App (and work folder ‘frontend’) with the following command:   
  **npm create vite@latest frontend -- --template react**
* Inside crated ‘frontend’ folder, install npm:  
  **npm install**
* Inside ‘backend’ folder, create virtual environment and install flask, flask-SQLAlchemy (ORM helping us to connect and map entries in SQL into a python object), and flask-cors (to allow cross-origin requests):

**python -m venv .env  
.env/Scripts/activate** or **source .env/bin/activate**python -m pip install  **pip install Flask   
pip install Flask-SQLAlchemy**

**pip install flask-cors**

* Create the following files:  
  **config.py** Main configuration of the app **main.py** Main routes/endpoints will be contained here (or into separate files if the app is larger) **models.py** Database models / database interaction

1. **Config application**

Build API first (flask), *config.py:*

from flask import Flask

from flask\_sqlalchemy import SQLAlchemy

from flask\_cors import *CORS*

**CORS (Cross-Origin-Requests):** allowing us to send a request to this backend from a different URL. By default, the request is protected so the server can’t be hit from a different URL. In this case we have a frontend (different server) communicating with the backend, and in order for that to work the CORS error must be removed which can pop up.

app = *Flask*(\_\_name\_\_)

*CORS*(app)

In this way CORS ‘wraps’ our app, disabling the cors error for us, allowing us to actually send cross origin requests to our app.

app.*config*["*SQLALCHEMY\_DATABASE\_URI*"] = 'sqlite:///mydatabase.db'

Initialize Database config, storing a file (sqlite) specifying location where to save it (sqlite location)

app.*config*["*SQLALCHEMY\_TRACK\_MODIFICATIONS*"] = False

in order to track all modifications, set to DB. For now it is set to false for development purposes.

db = *SQLAlchemy*(app)

Instance of database passing our app: creating a database instance giving access to the database specified in line 8 (location), so that CRUD operations can be performed. Simply by using the database object (‘db’) from flask-sqlalchemy. An ORM that allows a normal python class (code) to modify sql db.

1. **Define database models *(models.py)*:**

Define what kind of fields are needed, how are these added-deleted, different keys that we might want to check. Once creating different data, we can start creating views allowing us to create or modify the data (which is what the API is responsible for doing).

- Import instance giving access to sqlalchemy (db)

**-** Create class Contact, inheriting from db.Model . A database model represented as a python class, and now in python it defines the different fields that this object will have.

🡪 Always need to have an id for all database instances.

db.Column specifying a type of field (integer), primary\_key=True specifies that every entry single entry used to index this object must be unique. This is autogenerated for us.

db.String(80) specifies an string field up to 80 characters, doesn’t need to be unique (unqui=False), billable=False means that you a ‘no value’ cannot be passed

from config import db

class *Contact*(*db*.*Model*):

*id* = db.Column(db.Integer, *primary\_key*=True)

    first\_name = db.Column(db.String(80), *unique*=False, *nullable*=False)

    last\_name = db.Column(db.String(80), *unique*=False, *nullable*=False)

    first\_name = db.Column(db.String(120), *unique*=True, *nullable*=False)

function to convert data from objects into a python dictionary to then convert to JSON in order to pass from the API. JSON (JavaScript Object Notation) is used back and forth so the API will receive json to create different objects and return json as a response.

    def *to\_json*(*self*):

        return {

            "id": *self*.id,

            "firstName": *self*.first\_name,

            "lastName": *self*.last\_name,

            "email": *self*.email,

        }

So ‘def to\_json(self):’ returns a python dictionary with keys returning values frm the db instance (ORM).

The keys written in ‘camelCase’, associated with their value (from Contact instance), are due to the JSON convention for camelCase fields (returning values from API). Whereas snake\_case is the convention for python.

1. **Routes/Endpoints for API**

Figure out what are the different routes or endpoints, that we want the API to access different data, resources, create resources, etc. In this case it is a CRUD application (crate [post], read [get], update, delete).

**# to CREATE what do we need?** first\_name, last\_name

These are values needed in order to submit a request to the CREATE endpoint.

**Note:** When we create an API, we typically have a server that’s running the API. The API is some kind of address (i.e. localhost:5000, since its running in local machine, or google.com, etc). An endpoint is simply something that comes out of this domain such as **‘localhost:5000/create\_contact’**. When this endpoint receives a **request**, some data is needed to be submitted alongside just the endpoints (ie. First\_name, last\_name, or Username & password, etc).  **Request:** is anything that is sent to some kind of server or API. Requesting something to happen. The request has a ‘*type’*, and it can be one of many different things. These are the most typical: ‘GET’ (access some type of resource), ‘POST’ (create something new), ‘PUT’ & ‘PATCH’ (updates a resource), and ’DELETE’.

Another type of data that can be exchanged, is **json data**. It is information that comes alongside our request and is used while handling the request to do something. So if there is a ‘DELETE’ request, we need to pass the type of contact we want to delete

**localhost:5000/delete\_contact’** \*endpoints written in python code

Type: DELETE \* request from an external website / or frontend

Json: { }

* In this case, the ‘frontend’ sends a request to the ‘backend’, and the ‘backend’ is going to return a response.

**Response:** the response will contain a few things. ‘*Status’* will specify If the request was successful (200: success, 201: created, 400: ‘bad request, 403: forbidden/unauthorized, 404: not found [ie. The html tried to be rendered wasn’t found]) to indicate what happened. ‘*Json data*’ would be returned (ie. ‘GET’ request)

**localhost:5000/get \_contact’** *(request)*

Status: 200 \* API response: how it was handled

Json: {firstName: \_\_\_\_\_, last….,etc} \* result looked for (in json data)

from flask import request, *jsonify*

from config import app, db

from models import *Contact*

Import flask ***request***object & ***jsonify*** allowing us to return JSON data. Also import ***app*** and ***dabase*** from config, and ***Contact*** model from models.

if \_\_name\_\_ == "\_\_main\_\_":

    with app.*app\_context*():

        db.*create\_all*()

    app.*run*(*debug*=True)

executes if this script is run directly

Run Flask application (main.py) file directly (\_\_name\_\_ special method), instantiating database. In other words: once application starts, get Contact of application and create [ ‘*db.create\_all()‘* ] all the different models defined in database. It would only do this if they’re not already created, but the idea is to ‘spin up the database’ if it doesn’t already exist. And this will run all our different endpoints and the API.

*( ‘with’ simplifies resource management particularly for objects that require cleanup or initialization actions. It is commonly used with file handling, bit it can also be used with other objects that support the context management protocol. )*

***‘GET’ METHOD’***

*@app*.*route*("/contacts", *methods*=['GET'])

def *get\_contacts*():

    contacts = *Contact*.query.*all*()

    json\_contacts = *list*(*map*(lambda *x*: *x*.to\_json(), contacts))

    return *jsonify*({"contacts": json\_contacts})

if \_\_name\_\_ == "\_\_main\_\_":

    with app.*app\_context*():

        db.*create\_all*()

    app.*run*(*debug*=True)

*(decorator, written just before a function)*

At app’s route, specify the route/endpoint to go to (URL), method type (ie. ‘GET’) for the given URL.

*(function)*

Name the function, specifying how to handle the given request (GET) that is sent to the route (“/contacts”).

**contacts = Contact.query.all()**

Uses flask-sqlalchemy’ (ORM) to get all the different contacts inside db. Since they are python objects and they can’t be returned from our code, they have to be converted to JSON data (serialized).

**json\_contacts =list(map(lambda x: x.to\_json(), contacts))**

Our ‘**contacts**’ is a list is a list of contact objects, and we know that all thse contact objects have the ‘**to\_json**’ method, all we need to do is to call that method for all the different contacts and then create a new list that just contains the JSON for the contact. So what **map** does, is take all the elements from this list, and apply a function to them and gives us the result in a new list. The **lambda function** (a shortcut for writing a function in one line), 1. Say ‘lambda’, 2. Specify a variable in this case it is the parameter, and 3. Use that parameter and call anything you want. In this case X is going to be one of the contacts from our list, we’re going to call on ‘**to\_json**’ on that contact and put it inside a new list.

The only thing, this map actually returns a new map object, so what we want is a ‘*list*’ so we just convert it to a list with the list function.

**return jsonify({“contacts”: json\_contacts})**

Return a json object that says “contacts” and it’s equal to **json\_contacts.** In other words, we have this “contacts” key in our python dictionary and it’s going to be associated with the json\_contacts (list created with all the contacts previously from the same function) and converted to JSON using jsonify function.

1. **Run program**

python main.py

1. **Creating the Contacts**

*@app*.*route*("/create\_contact", *methods*=['POST'])

def *create\_contacts*():

    first\_name = request.json.get('firstName')

    last\_name = request.json.get('lastName')

    email = request.json.get('email')

    if not first\_name or not last\_name or not email:

        return (

*jsonify*({"message": "You must include a first name, last name and email"})

        )

    new\_contact = *Contact*(*first\_name*=first\_name, *last\_name*=last\_name, *email*=email)

    try:

        db.session.*add*(new\_contact)

        db.session.*commit*()

    except *Exception* as e:

        return *jsonify*({"message": *str*(e)}), 400

    return *jsonify*({"message": "User created!"}), 201

* Add route for creating contact, specifying route/endpoint (URL) and method (POST).
* Create function create\_contact() where it gets the data associated with contact we want to create. So what’s going to happen here is, look at the json data submitted here, and make sure it’s valid for creating our contact object (it would follow the object model requirements).
* **If not** first\_name **or not**  last\_name **or not** email: returns error if required data is not provided. Specifying the appropriate error code.
* In other words, look in the **json** data, use the **.get** ‘cause we don’t know if this key is going to exist or not. If it doesn’t exists it’s going to return ‘none’ , otherwise is going to give us the value for every key (first n-last n-email).
* If information is provided completely (valid inputs), we make a new contact, add it to the database, and reply that contact creation was done successfully.
* Then add it the database (**try** walk…)

1. Python class is created corresponding to the entry (newcontact variable)
2. **(try)** We then add it to the data base session **(‘staging area’** not yet written in db**):** *db.session.add(new\_contact)*
3. Commit **(write** in **db** permanently**):** *db.session.commit()*
4. **(except)** catch any Exception to return it as a JSON message to the user.
5. If passed the try-except block, we return (a ‘jsonified’) message to user about success of ‘POST’ request, with status code 201 (created) or 200 if it’s a different kind of request.

USUALLY TESTING THE API WOULD BE DONE BEFORE WORKING ON THE FRONTEND: for this purpose for such simple routes it’ll be skipped.

POSTMAN is one of those used to test it out.

1. **Updating a Contact**

*@app*.*route*("/update\_contact/<int:user\_id>", *methods*=['PATCH'])

“we need to pass through this route (/update\_contact), a number indicating the id of the user to update”. It’s going to look like “/update\_contact/9”. Also, specify the valid methods (PATCH or POST)

def *update\_contact*(*user\_id*):

    contact = *Contact*.query.*get*(*user\_id*)

Grab new information and update field(s).   
- We take user id as a parameter (matching the path used previously in the decorator).   
Look into database find user by id, and store it ‘contact’ (variable). First thing to update is to verify that contact exists.

    if not contact:

        return *jsonify*({"message": "User not found"}), 404

- if not found generate error message

    data = request.json

    contact.first\_name = data.get("firstName", contact.first\_name)

    contact.last\_name = data.get("lastName", contact.last\_name)

    contact.email = data.get("emal", contact.email)

    db.session.*commit*()

    return *jsonify*({"message": "User updated."}), 200

- if found, we’ll start pursuing through the json data, and get the object fields (firstN, lastN, email).

What this is saying (contact.first\_name= ), is modify the this contact’s first\_name (attribute) to be equal to whatever the json data first name is that was provided (**request.json)**. If there’s no new name to update, leave it as it was ( **data.get(“firstName”, contact.first\_name** ). The **.get** method, looks for the key inside of a dictionary (which is what **request.json** would be). If the key exists, it returns **“firstName”**, if instead, it returns whatever the second argument is inside the .get function ( **contact.first\_name** ).

It handles any possible combinations of the different data that could be passing us (one or different fields)

- In the same way we would commit the change into the database. And since the contact already existed, already added in the session, when it commits it does it permanently.

- return ‘jsonified ‘message

1. **Deleting Contact**

*@app*.*route*("/delete\_contact/<int:user\_id>", *methods*=['DELETE'])

def *delete\_contact*(*user\_id*):

    contact = *Contact*.query.*get*(*user\_id*)

    if not contact:

        return *jsonify*({"message": "User not found"}), 404

    db.session.*delete*(contact)

    db.session.*commit*()

    return *jsonify*({"message": "User deleted!"}), 200

1. GET: query all the contacts, convert them into json and return them
2. POST: get all the json data (from fronten / user), make sure that we actually have all fields, create the new contact and add it into the session (db) and return message.
3. UPDATE: get all the contacts by id, update data field by **request.json**
4. DELETE: get contact by id and delete. Return message

(TEST, ie. POSTMAN, etc)

**FRONTEND**

1. **Clean unnecessary files from frontend-react boilerplate (since it’s not a big project)**

* Assets folder (inside ‘src’)
* In ‘app.jsx’, delete reactLogo and vitLogo import (from assets), and content return leaving only the ( <> </> ). So it looks like this:

import { useState } from 'react'

import './App.css'

function *App*() {

  return (

    <>

    </>

  )

}

export default *App*

* Delete ‘vite.svg’ from public folder
* In ‘app.jsx’, delete reactLogo and vitLogo import (from assets), and content return leaving only the ( <> </> ). So it looks like this:
* Remove reference to logo in ‘index.html’ and change doc’s title to ‘Contact List’ or something.

1. **Fetch the Contacts**

By default there are none, but understand how the request works by sending it.

Then, display them, and then add the form for creating contacts, updating, deleting, etc..

(Test)

1. From backend (folder), run main.py:  
   **python main.py**
2. From ‘app.jsx’, setup a state that’s going to store our contacts on an empty list to store the contacts.

function *App*() {

  const [contacts, s*etContacts*] = *useState*([])

1. Make an anonymous (=>) & asynchronous function to fetch the contacts. It’s going to be an asynchronous function because it has to wait a second to fetch the contacts.  
   Then in its implementation, an asynchronous operation would be performed to fetch data from backend (API, or reading from a file, or querying a database).

  const *fetchContacts* = async () => {

    const response = await *fetch*('http://127.0.0.1:5000')

**fetch** is used to make an HTTP request (by default GET), **from** flask’s server’s url (‘*http://127.0.0.1:5000/contacts’).* All we’re doing is sending a GET request to the url endpoint and waiting for that to give a response.

Once it gives a response, we want to get the JSON data associated with the response.

    const data = await response.*json*()

*setContacts*(data.contacts)

    console.*log*(data.contacts)

  }

‘data’ is set to be equal to the (awaited) response jsonified, and we setContacts to . This is 'going to return the JSON (the key value pair with contacts and their value as a list {‘ “*contacts”: [] ‘*}, ‘data’ grabs this contacts property [list] and sets that in the state [ *useState([])* ].   
  
And in the meantime, we can take a look at how that looks like (console.log(data.contacts)) calling it when the component renders. **We just want to call it once as soon as the website loads.**

To do that, we ‘useEffect’ (by importing it), and putting a useEffect inside the App() function, using an arrow function ( => ).

*useEffect*(() => {

*fetchContacts*()

  }, [])

It basically sasys ‘as soon as this component loads, call this function’.

function *App*() {

  const [contacts, *setContacts*] = *useState*([]) //(3)

*useEffect*(() => { //(1)

*fetchContacts*()

  }, [])

  const *fetchContacts* = async () => { //(2)

    const response = await *fetch*('http://127.0.0.1:5000')

    const data = await response.*json*()

*setContacts*(data.contacts)

    console.*log*(data.contacts)

  }

1. As soon as this component renders, call this function (‘*fetchContacts()’*)
2. Calling it (*‘fetchContacts()’*), is going to give us the contacts from the API endpoint
3. And set it in the state, and we can use that to list all the contacts set again   
     
   **(no contacts would be seen as there aren’t any yet, but this at least is going to allow us to test by sending a request to our back end)**

Split terminal window, and see into the frontend directory, and run the code: **npm run dev**  
It will spin up the frontend server (running on port:5173) and run the the localhosts’ url.  
  
- A ‘GET’ request would be seen on the backend terminal every time the page loads.

- Inspecting webpage, there would be an empty array

1. **Display all the Context**

Make a new component in ‘src’ folder called ‘ContactList.jsx’. This is going to be the component for rendering our contacts.

A new component ‘ContactList’ is equal to a function. Inside it, there will be a prop named ‘contacts’ so we can have them to render. Then we return some basic HTML with a div with a header and a table.

import React from 'react'

const *ContactList* = ({*contacts*}) => {

    return <div>

        <h2>Contacts</h2>

        <table>

            <thead>

                <tr>

                    <th>First Name</th>

                    <th>Last Name</th>

                    <th>Email</th>

                    <th>Actions</th>

                </tr>

            </thead>

            <tbody>

                {*contacts*.*map*((*contact*) => (

                    <tr key={*contact*.id}>

                        <td>{*contact*.firstName}</td>

                        <td>{*contact*.lastName}</td>

                        <td>{*contact*.email}</td>

                        <td>

                            <button>Update</button>

                            <button>Delete</button>

                        </td>

                    </tr>

                ))}

            </tbody>

        </table>

    </div>

}

export default *ContactList*

‘thead ‘ would have some static information, and the ‘tbody’ would be dynamically rendered.   
The ‘tbody’ would have the prop (contacts) mapping the function (contacts) with all the contacts available, returning a new row for them (<tr></tr> inside function).

We need ‘ **key**’ inside the table element (<tr>) in order to display the dynamic data, so we pass the contact object id, and the models passed through JSON syntax (<td>)

We then need the Update and Delete buttons (inside <td> as well). Functions will be added to them later on. For now let’s render this component form app.jsx.

In order to import it to app.jsx, we need to export it.

1. **Render ContactList component (‘app.jsx) / View**

import ContactList from './ContactList'

Import it first

Render ContactList

return <ContactList contacts={contacts} />

Just to test it out, we can comment out the fetching of contacts (useEffect) and enter some values on the useState just to see (mock contact).

function *App*() {

  const [contacts, *setContacts*] = *useState*([{firstName: "Oscar", lastName: "Arguelles", email: "daso@das.com", id: 1}])

*useEffect*(() => {

*//fetchContacts()*

  }, [])

(remove mock contact from from ‘app.jsx’)

1. **Create Contacts Component**

Create ‘ContactForm.jsx’ inside src folder, and import useState from react.

Craete an arrow function and inside we need a state for our forms where we need to store our firstName/lastName/email, etc (model objects). We’re going to used use that and submit that to our API to create new content.

import {useState} from 'react'

const ContactForm = (*//props) => {*

*//state for form*

}

Set content we want to store, ‘useState’ set to empty string.

const *ContactForm* = () => {

    const [firstName, *setFirstName*] = *useState*("")

    const [lastName, *setLastName*] = *useState*("")

    const [email, *setEmail*] = *useState*("")

Return form

return (

        <form onSubmit={*onSubmit*}>

            <div>

                <label htmlFor="firstName">First Name:</label>

                <input

                    type="text"

                    id="firstName"

                    value={firstName}

                    onChange={(*e*) => *setFirstName*(*e*.target.value)}

                />

            </div>

            <div>

                <label htmlFor="lastName">Last Name:</label>

                <input

                    type="text"

                    id="LastName"

                    value={lastName}

                    onChange={(*e*) => *setLastName*(*e*.target.value)}

                />

            </div>

            <div>

                <label htmlFor="email">Email:</label>

                <input

                    type="text"

                    id="email"

                    value={email}

                    onChange={(*e*) => *setEmail*(*e*.target.value)}

                />

            </div>

            <button type="submit">Create Contact</button>

        </form>)

}

export default *ContactForm*

Inside <div> element (of form element), we’ll have the first label or input element for ‘firstName’.

onChange is set to ‘get a function ‘e’, setFirstName to e. target value. Replicate the same format for lastName, and email.

Then have a button to create these elements. All we want to do is create a function so that when it’s pressed, id does what it is intended to (1)(2).

(Export this as well)

1. Function will be triggered when button is pressed (onSubmit). It will take an ‘e’ and inside it, a function to prevent the page to refresh automatically, and set up a post request.

(onSubmit)

1. Define data that we want to pass with the **json** as the **request**. We pass in the information that corresponds with what we’re looking for in the API when we create new content.
2. Specify the url (‘POST’ request in this case)
3. Set some options for the request. 1st specify the method (POST) since it is not a GET request (which is automatically done by default). 2nd a header (an object) specifying that it is json data so the API knows that we have json data. 3rd body, ‘stringify’ the json string to then include in the body of our request.
4. Send the request (to API). Fetch the URL (Await) , with the options specified. Since await (async) is used for our fetch, ***async*** has to be used on the ***onSubmit.***
5. Check if response was successful with status codes. Alert user if there was an error (not 201 && not 200.

    const [email, *setEmail*] = *useState*("")

    const *onSubmit* = async (*e*) => { *//(1)*

*e*.*preventDefault*()

        const data = { *//(2)*

            firstName,

            lastName,

            email,

        }

        const url = 'http://127.0.0.1:5000/create\_contact' *//(3)*

        const options = { *//(4)*

            method: "POST",

            headers: {

                "Content-Type": "application/json"

            },

            body: JSON.*stringify*(data)

        }

        const response = await *fetch*(url, options) *//(5)*

        if (response.status !== 201 && response.status !== 201){ *//(6)*

            const data = await response.*json*()

*alert*(data.message)

        } else {

*//successful response*

        }

    }

1. **Render contact form (app.jsx):**

Modify return to incorporate ContactForm

  return (

    <>

      <ContactList contacts={contacts}/>

      <ContactForm />

    </>

    );

}

We can try now the form and see that we haven’t force-refreshed the screen, but once it is refreshed, the data entered would show up along with the Action buttons (Update, Delete), generating the ‘POST’ request along with the status code (on the backend terminal).