**1) Identifying the Problem**

* **Goal**: Display a map in Streamlit where clicking on a marker returns information to Python (e.g., marker coordinates or metadata).
* **Limitation**: Neither st.map() nor st.pydeck\_chart() natively provides a Python callback for marker clicks. While PyDeck can show tooltips or run JavaScript, it doesn’t pass click events **back** to Python by default.

**2) Evaluating Possible Approaches**

1. **st.map()**
   * Very convenient but no custom interactivity; no way to handle marker clicks in Python.
2. **st.pydeck\_chart()**
   * Allows hover tooltips and highlighting but still doesn’t send click data back to Python.
3. **Direct JavaScript Injection**
   * You can embed a map in HTML/JS and handle clicks in the browser, but there’s no built-in path to feed those clicks into Python.
4. **Build a Custom Streamlit Component** (the ultimate solution)
   * Leverages Streamlit’s Components API to create a **bidirectional** bridge between a front-end (JavaScript/React) and Python.

**3) Building a Custom Streamlit Component**

**3.1 Project Structure**

You set up a folder named my\_map\_component containing:

* frontend/: A React/Leaflet app compiled with Parcel (or another bundler).
* \_\_init\_\_.py: Declares the component to Streamlit.
* demo\_app.py: A simple Streamlit script demonstrating how to use the component.

The final structure resembled:

my\_map\_component/

├─ demo\_app.py

├─ \_\_init\_\_.py

└─ frontend/

├─ package.json

├─ src/

│ ├─ index.html

│ ├─ index.jsx

│ └─ MyMapComponent.jsx

└─ build/

**3.2 Front-End (React + Leaflet)**

1. **Leaflet** was chosen for easy marker handling.
2. **React** to render the map and handle state.
3. **streamlit-component-lib** for communication between JavaScript and Python.

Key points:

* **Markers** are created from data passed in from Python (props.args.data).
* A **click event** on each marker calls Streamlit.setComponentValue(...), sending a JSON-serializable dict back to Python (e.g., {clickedName: ..., lat: ..., lon: ...}).
* **Leaflet icons** require special imports so they don’t 404 when bundled.

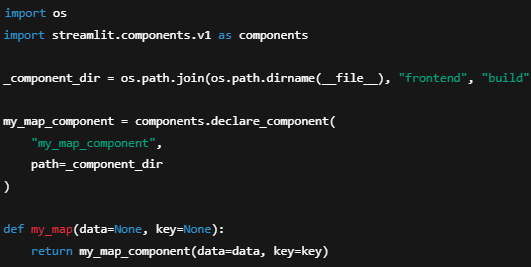
**3.3 Bundling with Parcel**

* You created an **HTML entry point** (index.html) and a **React entry file** (index.jsx) that renders <MyMapComponent />.
* A **package.json** script: "build": "parcel build src/index.html --out-dir build"
* Running npm run build outputs a build/ folder with index.html and the compiled JavaScript.

**4) Integrating the Component in Python**

**4.1 Declaring the Component**

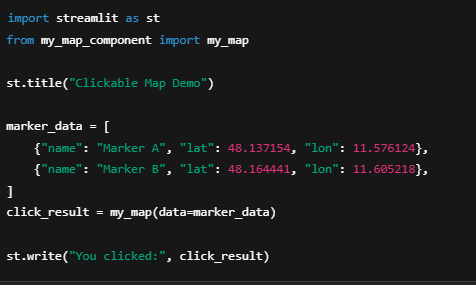
In my\_map\_component/\_\_init\_\_.py:



* **declare\_component** points to the build/ folder.
* **my\_map()** is a helper that passes data (marker info) to the front-end.

**4.2 Using It in a Streamlit App**

In demo\_app.py:



* The front-end **Leaflet** map displays these markers.
* Clicking a marker triggers Streamlit.setComponentValue(...) in JS, returning the marker info to click\_result in Python.

**5) Final Enhancements**

* **Full-Screen Map**: You used custom CSS in Streamlit to remove margins/padding and set overflow: hidden;.
* **Center on Munich**: Default coordinates set in MyMapComponent.jsx.
* **Marker Icon Fix**: Overrode Leaflet’s default icon paths to avoid 404 errors when bundling.

**6) Conclusion**

Because Streamlit’s built-in map functions don’t provide Python callbacks on marker clicks, you **created a custom component**. This approach:

1. **Bridges** front-end interactivity (Leaflet’s click events) with **Python** logic.
2. **Bundled** a React/Leaflet app using Parcel, then served it through Streamlit.
3. **Exchanged data** between JavaScript and Python via Streamlit.setComponentValue(...) and declare\_component.

This solves the limitation of **no direct click callbacks** in st.map() or st.pydeck\_chart() by letting you handle events in JavaScript and pass them to Python for further processing.