HOMEWORK ASSIGNMENT 3

CSCI 571 – Fall 2025

Abstract

Ajax, JSON, Frontend Frameworks, CSS Frameworks, Node.js, and Ticketmaster API

Prof. Marco Papa

papa@usc.edu

Assignment 3: Ajax, JSON, Frontend Frameworks, CSS Frameworks, Node.js, and Ticketmaster API

1. Objectives

1. Get experience with creating backend applications using JavaScript/Node.js on the server side with Fastify/Express/Hono/Elysia framework.
2. Get experience with using Frontend and CSS Frameworks on the client side and creating responsive front-end.
4. Get experience with Ticketmaster APIs, Spotify APIs, Google Maps APIs, Google Geocoding APIs, IPinfo APIs, Facebook share APIs, and Twitter share APIs.
5. Get experience with Google Cloud Platform.

2. Homework Description Resources

1. Homework Description Document (This document)
2. Rubric (aka Grading Guidelines)
3. Web Reference Video
4. Mobile Reference Video
5. Piazza

3. General Directions

1. The backend of this Assignment must be implemented in JavaScript using Node.js Express framework (or Fastify/hono.js/ElysiaJS – however limited TA/CP support will be available). Refer to Node.js website for installing Node.js and learning how to use it. Have a look at "Getting started" guides in Express website to learn how to create backend applications using Express. Node `fetch()` can be useful to make requests from your Node.js backend to Ticketmaster servers. **Implementing the backend in anything other than Node.js will result in a 4-point deduction.**
2. The frontend of this Assignment must be implemented using the **Angular, React, or Vue.js** frameworks.
    a. Refer to Angular setup docs for installing Angular and creating Angular projects. Angular "Tour of Heroes" app tutorial is a very good tutorial to see different Angular concepts in action. **Implementing the frontend in anything other than Angular, React or Vue.js will result in a 4-point deduction.**
3. You are expected to create a responsive website. For that reason, we recommend you to use **Shadcn** + **Tailwind CSS**.
4. The backend of this Assignment must be deployed on Google Cloud. The backend should serve the frontend as well as other endpoints you may define. Please refer to the instructions on D2L Brightspace for deploying Node.js applications to any of the cloud services.

5. You must refer to the Assignment description document (this document), rubric, the reference videos and instructions in Piazza while developing this Assignment. All discussions and clarifications in Piazza related to this Assignment are part of the Assignment description and grading guidelines. Therefore, please review all Piazza threads before submitting the assignment. If there is a conflict between Piazza and this description and/or the grading guidelines, answers and clarifications in Piazza supersede the other documents.
6. The Assignment will be graded using the latest version of the Google Chrome browser. Developing the Assignment using the latest version of Google Chrome is advised.

4. System Overview

The system contains three components: 1) browser (frontend), 2) Node.js application (backend) and 3) external services/APIs (Ticketmaster, Google Geocoding, Ipify.info). You will have to implement both the frontend and the backend. Your backend will include two major functionalities: serving the frontend static files to the browser and responding to the frontend's AJAX requests by fetching data from Ticketmaster servers. You will not directly call the Ticketmaster APIs from the frontend as it requires disclosing a secret API key to the public. You will call ipinfo.io and Google Geocoding API directly from your frontend: calling ipinfo.io directly allows the service to capture user's IP address, not your backends' one; Google Geocoding API support direct calling and has an option to scope key usage to a specific website. The data flow diagram after an AJAX call is shown below in **Figure 1**.

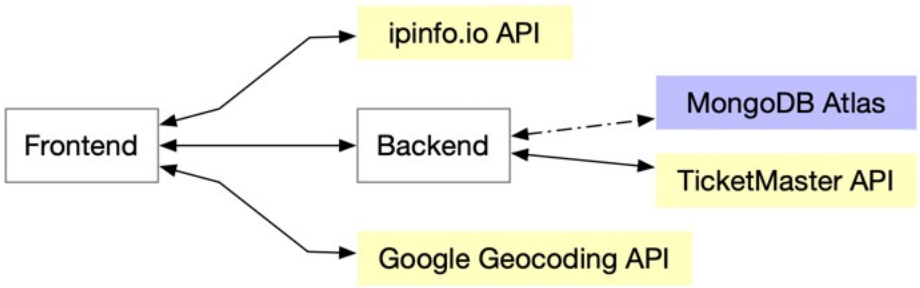


Figure 1: System design overview

NOTE: Setup authorization is required to call Ticketmaster API endpoints and is using request headers with the API key for the app. Refer to the Additional Hints section to see how to add authentication headers. You can re-use the API Key obtained and used in Assignment 2.

Please do not directly call the Ticketmaster API endpoints from your frontend. Calls to other APIs can be made from the frontend.

All read requests from the frontend to the backend must be implemented using the HTTP GET method, as you will not be able to send us sample backend endpoint links as a part of your

submission, if you use HTTP POST. You are encouraged to use other HTTP verbs for working with favorites (adding, removing, etc).

5. Description

In this exercise, you are asked to create a web application that allows you to search for event information using the Ticketmaster API, and the results will be displayed in a card in tabular format. The application will also allow users to mark events as "Favorites" and see the list of all events marked as favorites. Also, users can share a post on Facebook and a tweet on Twitter about the events.

All implementation details and requirements will be explained in the following sections.

There are 3 front-end routes/pages for this application:

a) Search Route ['/search'] – It is the default route of this application which is used to search for events
b) Event detail route ['/event/<id>'] – This route will show information about the corresponding event
c) Favorites Route ['/favorites'] – It displays the list of favorite events

Exact route paths are only given as examples, feel free to change/nest them.

Hint: use a routing library for your framework.

5.1 Navbar component

The Navigation bar must be present on top of all the routes of the application as shown in **Figure 2** below. It consists of the "Events Around" title and the following menu options:

1. Search
2. Favorites

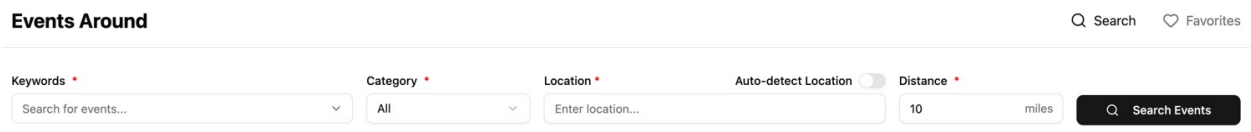


Figure 2: Navbar

5.2 Search Route

The Search route consists of two main sections:

- Search Form
- Results grid view

5.2.1 Search Form

The form has 5 fields: Keywords, Category, Distance (miles), Location, and a checkbox to auto-detect location as you can see in **Figure 3**. You should use the ipinfo.io API (as you did in Assignment 2 to fetch the user's geolocation, if the location checkbox is checked. Otherwise, the user must enter an address location to search.

Location field is a text field. It is disabled and cleared when the "Auto-detect Location" is enabled.
Distance field is a numeric input with "miles" text appended to the right side of the input.
Keywords field should be implemented as a dropdown select with search and suggest functionality.

Please refer to the video for the functionality and visual appearance. When user starts typing, an autosuggest request should be made (see the next section) and corresponding items from the response should be shown. When a request is made, the component should show a spinner on the right and once the value is selected, a cross icon should be visible that clears the input value. Add a reasonable throttling/debouncing to limit the API usage.

Category field should be implemented as a dropdown.

These are the categories to include in the dropdown:

- All
- Music
- Sports
- Arts & Theatre
- Film
- Miscellaneous

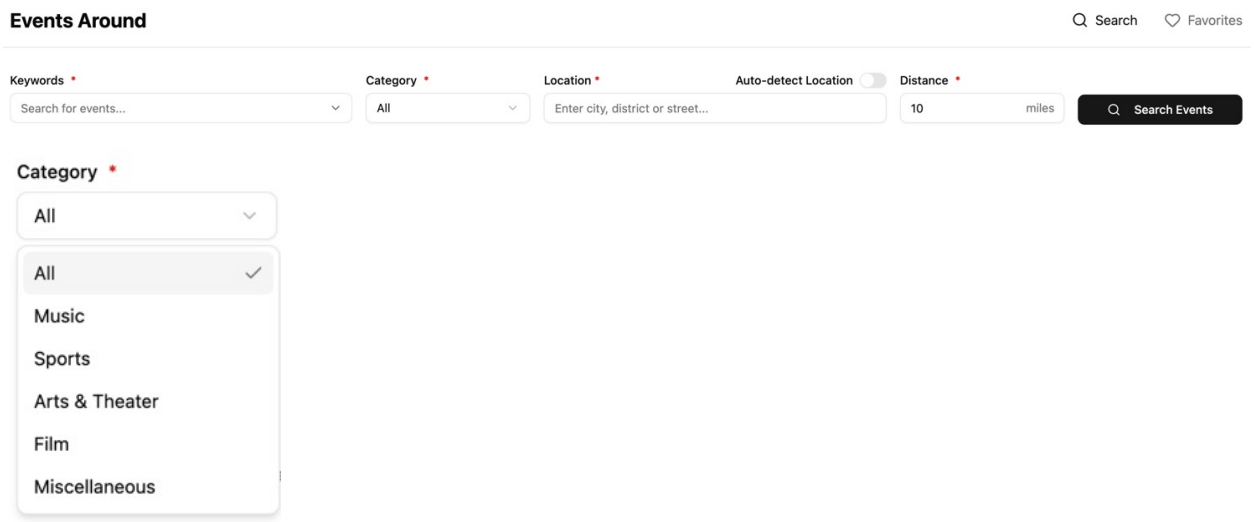


Figure 3: Events search form, search form with location auto-detect, category dropdown

5.2.1.1 Autocomplete data source

The Keyword Input field allows the user to enter a keyword to retrieve results. Based on the user input, the text box should display a list of all the suggestions fetched using a *Ticketmaster suggest API*, as seen in **Figure 4**.

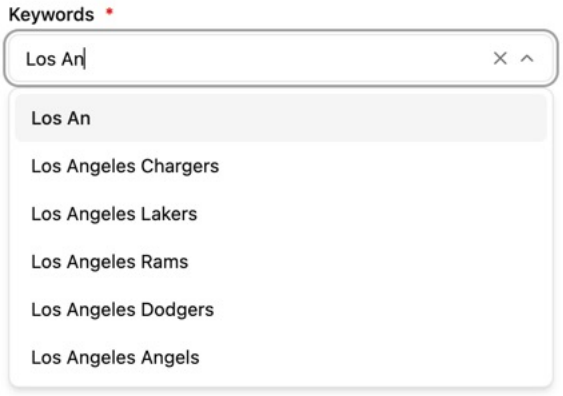


Figure 4: Example of Autocomplete

This is the API endpoint for autocomplete from the *Ticketmaster suggest API*:

GET https://app.ticketmaster.com/discovery/v2/suggest

Please refer to this documentation for the *Ticketmaster Suggest API*:
https://developer.ticketmaster.com/products-and-docs/apis/discovery-api/v2/suggest

This is a sample API call:

https://app.ticketmaster.com/discovery/v2/suggest?apikey=YOUR_API_KEY&keyword=[KEYWORD]

The response is a JSON object, as shown in **Figure 5**.

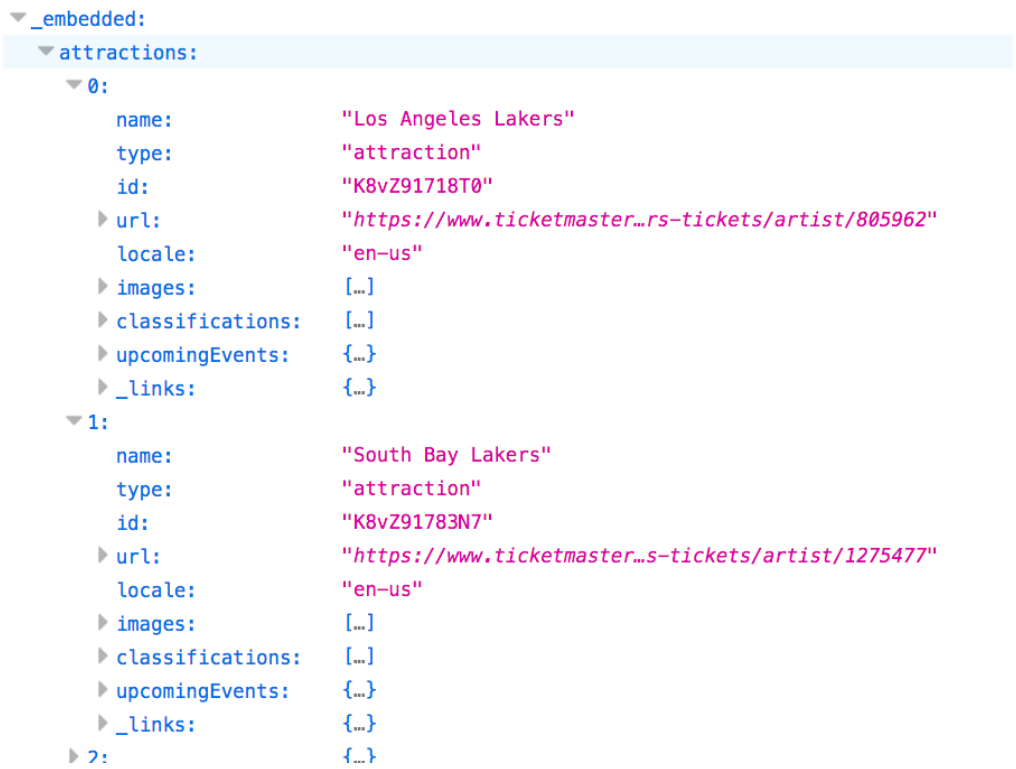


Figure 5: Autocomplete JSON Response

5.2.1.2 Location Field

If the "**Auto-detect your location**" **checkbox** is checked, then the location field should reset the **Location** textbox to blank and disable the field as shown in **Figure 7**. When the **Auto-detect checkbox** is not checked, the user needs to enter the location address. Use the *Google Maps Geocoding API* to get latitude and longitude of the location that the user entered and pass that latitude / longitude values in the Ticketmaster's event search API. This behavior is pretty much the same as in Assignment 2.

The *Google Maps Geocoding API* is documented here:

https://developers.google.com/maps/documentation/geocoding/start

The *Google Maps Geocoding API* expects two parameters:

1. **address**: The location that you want to geocode, in the format used by the national postal service of the country concerned. Additional address elements such as event names and unit, suite or floor numbers should be avoided.
2. **key**: Your Google application's API key. This key identifies your application for purposes of quota management.

An example of an HTTP request to the *Google Maps Geocoding API*, when the location address is "University of Southern California, CA" is shown below:

https://maps.googleapis.com/maps/api/geocode/json?address=University+of+Southern+California+CA&key=YOUR_API_KEY

The response includes the latitude (lat) and longitude (lng) of the address

Figure 6: Example of JSON object from Google Maps Geocoding

Figure 6 shows an example of the JSON object returned by the *Google Maps Geocoding API* web service response.

The latitude (lat) and longitude (lng) of the location are used when constructing a RESTful web service URL to retrieve matching search results.

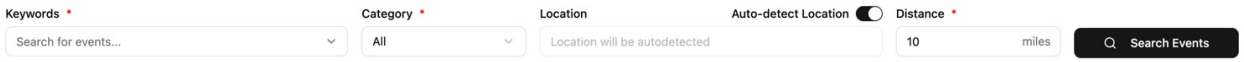


Figure 7: Example of auto-detect location selection

5.2.1.3 Search button

Clicking the **Search Events button** performs a search using the given event *keyword*, the *distance* filter from the given location and the *category* of events. An example of valid input is shown in **Figure 9**. Once the user has provided valid input, your frontend should send a request to your backend NodeJS server with the form inputs. You must use GET to transfer the form data to your web server (**do not use POST**, as you would be unable to provide a sample link to your cloud services). The backend code will act as a proxy - extract the form inputs and make an HTTP call, using the inputs to invoke the *Ticketmaster API* event information service. You need to use the backend to make all the Ticketmaster API calls.

If the user clicks on the SUBMIT button without providing a value in the "Keyword", "Category" or "Location" fields or checking the location checkbox, you should show an error state. An example is shown in **Figure 8**.

<u>Do not call the Ticketmaster API or directly from the front-end, this will lead to a 4-point penalty.</u> You may use fetch to call the *IPinfo API* service and the Google Geocoding API directly from the front-end JavaScript, as in Assignment 2.

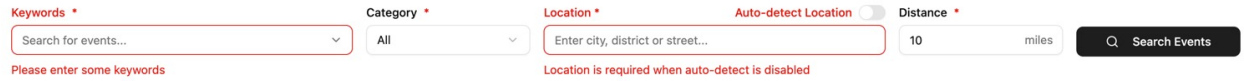


Figure 8: Example of form validation

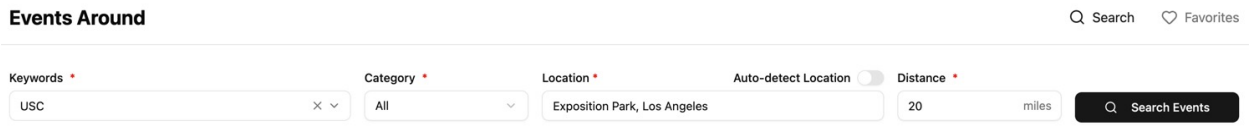


Figure 9: Example of valid input

5.2.2 Results grid view

The response received from the backend after clicking the SUBMIT button will be parsed and displayed in a grid of cards as shown in **Figure 10**. The **event card** consists of 5 columns, and a button:

1) Category
2) Date/Time
3) Cover image
4) Event name
5) Venue name
6) "Like" button

The results table must display a maximum of **20 search results (1st page of Ticketmaster response)**. Each card is such that, when clicking anywhere on it, you navigate user to the Details page, which will be discussed in the next section.

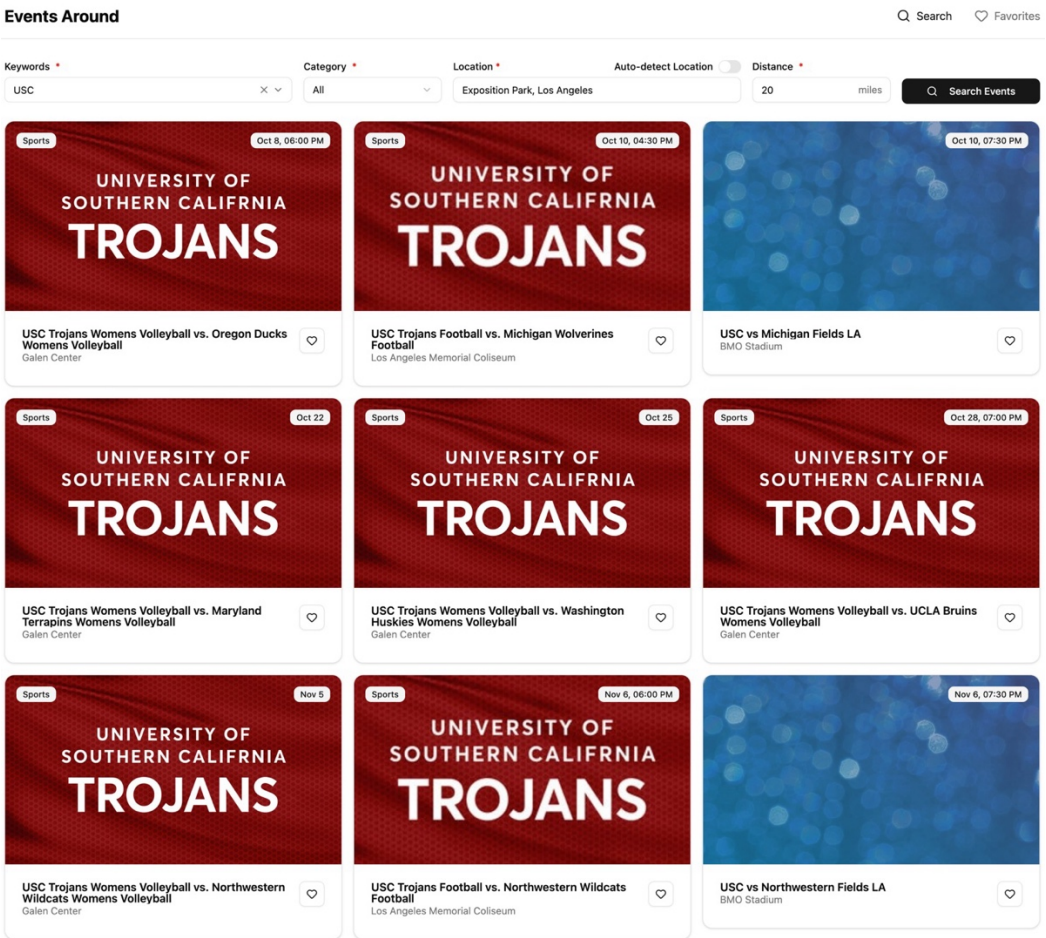


Figure 10: Example of results table

If results are not found, display “No results available”, as shown in **Figure 11**.

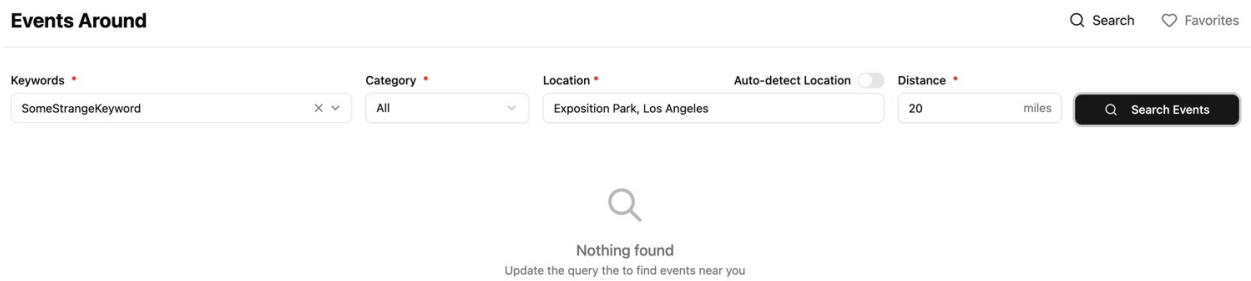


Figure 11: Example of no search result found

The API endpoint for search is:

GET https://app.ticketmaster.com/discovery/v2/events

Refer to the search documentation at:

https://developer.ticketmaster.com/products-and-docs/apis/discovery-api/v2/#search-events-v2

API Sample:

```
https://app.ticketmaster.com/discovery/v2/events.json?apikey=
YOUR_API_KEY&keyword=University+of+Southern+California&segmentId=KZFzniwnSyZf
Z7v7nE &radius=10&unit=miles&geoPoint=9q5cs
```

The Node.js script should pass the JSON object returned by the *Event Search* to the client side or parse the returned JSON and extract useful fields and pass these fields to the client side in **JSON format**. Use your frontend framework of choice to dynamically create cards with information from the response. A sample output is shown in **Figure 10**. Events should be displayed by **ascending order of “Local Date/Time”**.

When the search result contains at least one record, you need to map the data extracted from the API results to the columns to render the HTML result table as described in **Table 1**.

Card information	API service response
Name	The value of “name” attribute directly on “event” item of the result list
Date	The value of the “*localDate*” and “*localTime”* attributes of “*start”* object that is part of “*dates*” attribute object.
Cover image	The value of the first image in “*images*” list of the “*events*” object.

Event	The value of the "*name*" attribute that is part of the "*events*" object.
Genre	The "name" value of the first "segment" attribute under "classifications" list.
Venue	The value of the "*name*" attribute that is part of the first embedded "*venue*.

Table 1: Mapping the result from Event Search API into HTML table

5.3 Details Page

The **Details page** consists of Back button, Event name, Buy Tickets, Favorite button and **three tabs** as follows:

1) Info
2) Artists/Teams
3) Venue

Use shadcn tabs to implement tabs. In the search results, if the user clicks on the event card, user should be navigated to event detail page. The page should request the detailed information using the *Event Details API*, documented at:

https://developer.ticketmaster.com/products-and-docs/apis/discovery-api/v2/#anchor_get

To retrieve event details, the request needs two parameters (output should be JSON):

- ***id:*** ID of the event
- ***apikey:*** Your application's API key. This key identifies your application for purposes of quota management.

Top part, above tabs, should contain these elements:

- "Back to search" link – should return to the search, see the behavior described later.
- Event name
- *Buy Tickets* – Under "Buy Ticket At", there should be a "Ticketmaster" link, a page to buy tickets online which should open in a new page.
- Favorite button – behavior described in future sections.

5.3.1 “Info” tab

The **Events details** tab shows a table containing the detailed info of the event such as Date, Artists/Team, Venue, Genres, Price Ranges, Ticket Status, a link to Buy Tickets, Facebook icon, X (formerly known as Twitter) icon and a Seat map. See **Figure 13**.

-
- *Date* – displays “*localDate*” and “*localTime”*
- *Artists/Team* – displays artists/team *“name”* segmented by a comma
- *Venue* – displays “venue name”
- *Genres* – displays genre in the order of "segment", "genre", "subGenre", "type", "subType"
- *Ticket Status* – in the event details card is color coded. The convention used is follows:
    - On sale: Green
    - Off sale: Red
    - Canceled: Black
    - Postponed: Orange
    - Rescheduled: Orange

Figure 12 displays the desired appearance of the respective status of the tickets.

Figure 12: Color coded card for the “Ticket Status” field

- *Seat Map* – displays an image of the seat map
- *Facebook icon* – On clicking the icon, create a post in a new tab containing the event’s Ticketmaster link as shown in **Figure 17**
- *Twitter icon* – On clicking the icon, create a tweet in a new tab containing *Check <event_name> on Ticketmaster*, followed by event’s Ticketmaster link as shown in **Figure 18**

NOTE: If any of the above fields is not available, please don’t display that attribute in the detail card.

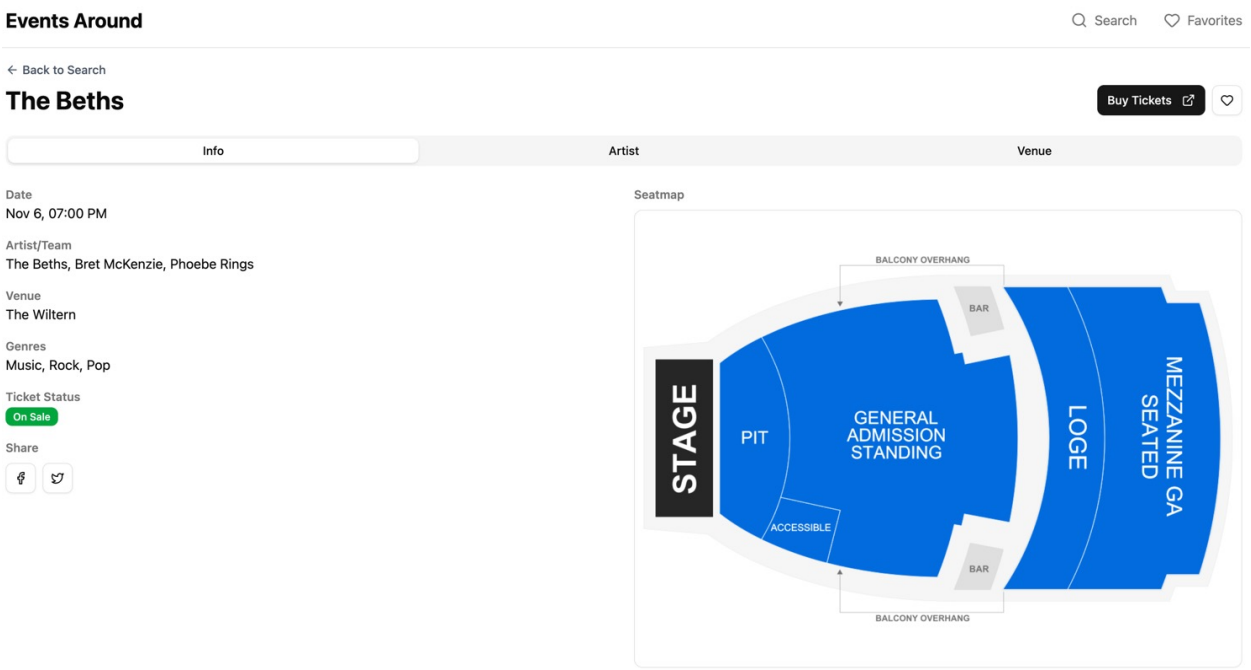


Figure 13: Example of Events Details page

Back button –

Clicking on the **Back to search button** on the top left corner of the Event details card, should navigate back to the search results. Note that all previously filled values as well as the results and the scroll position should be preserved. In other words, user should see exactly the same as before opening detail page. If the detail page was navigated from favorites, open an empty search page.

Favorite icon –

On clicking the **Favorite icon**, display a 'Event Added to Favorites!' alert as shown in **Figure 14, Figure 15 and Figure 16** and change the heart icon to **a red color-filled heart icon** as shown. Once an event is marked as favorite, if we revisit the details of that event, it should display the "red" color-filled heart icon instead of a "white" heart, until the event is removed from the "favorites" page or if the **Favorite** icon is clicked again, which should display a 'Removed from Favorites!' alert and change the icon to display **white heart icon**.

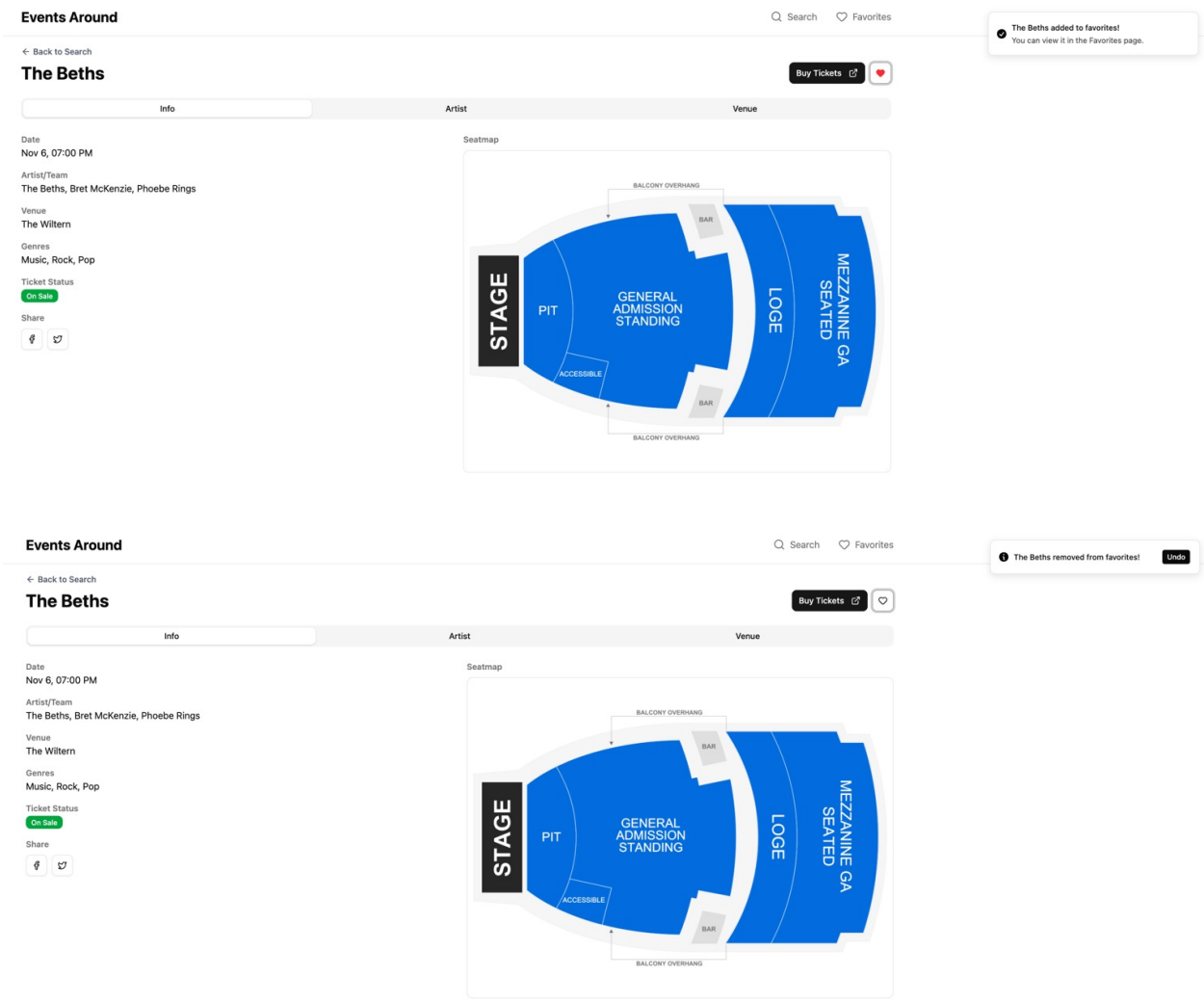


Figure 14: Example of Event added/removed to Favorites.

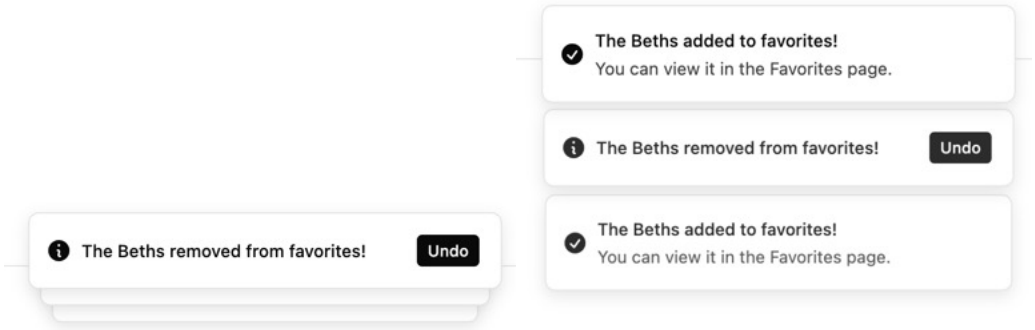


Figure 15: Example of "Sonner" stack of notifications

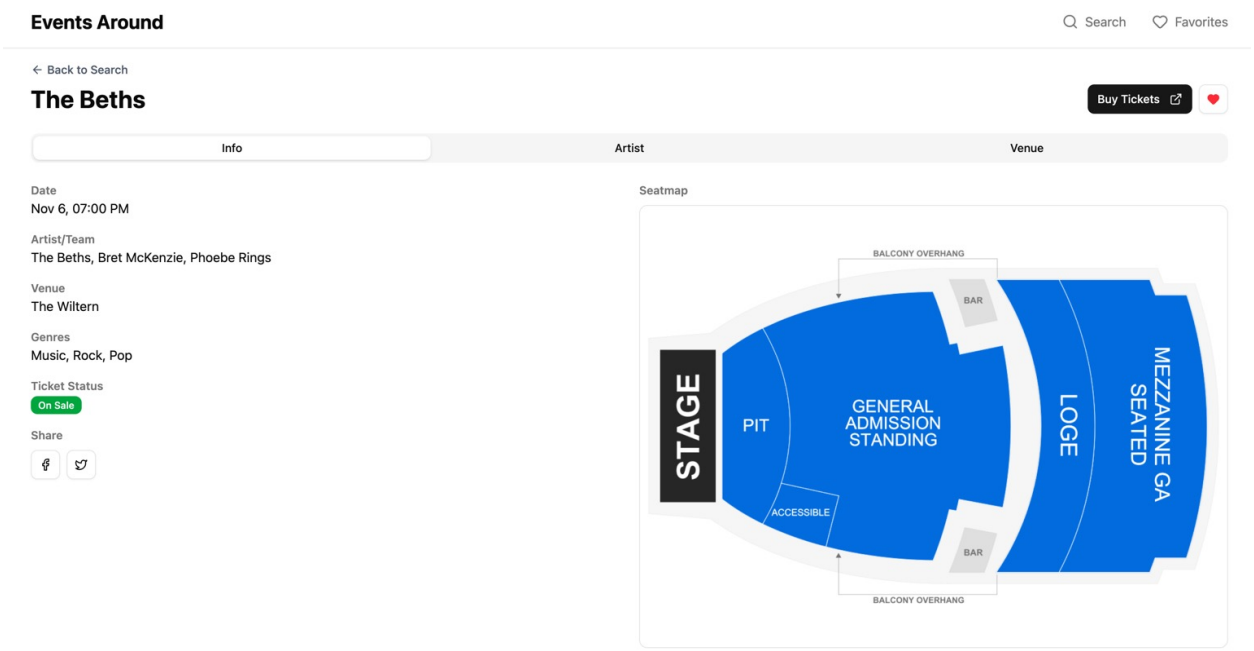


Figure 16: Example of Favorite heart icon turning red after event marked as Favorite.

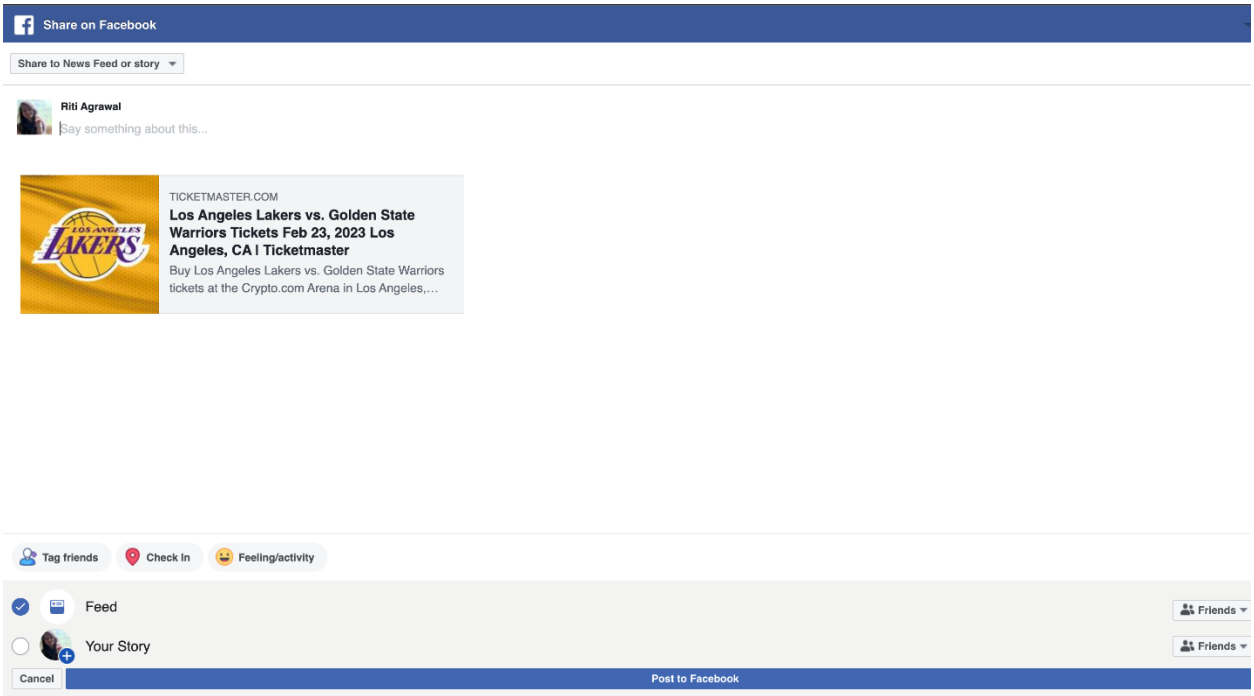


Figure 17: Example of Facebook post

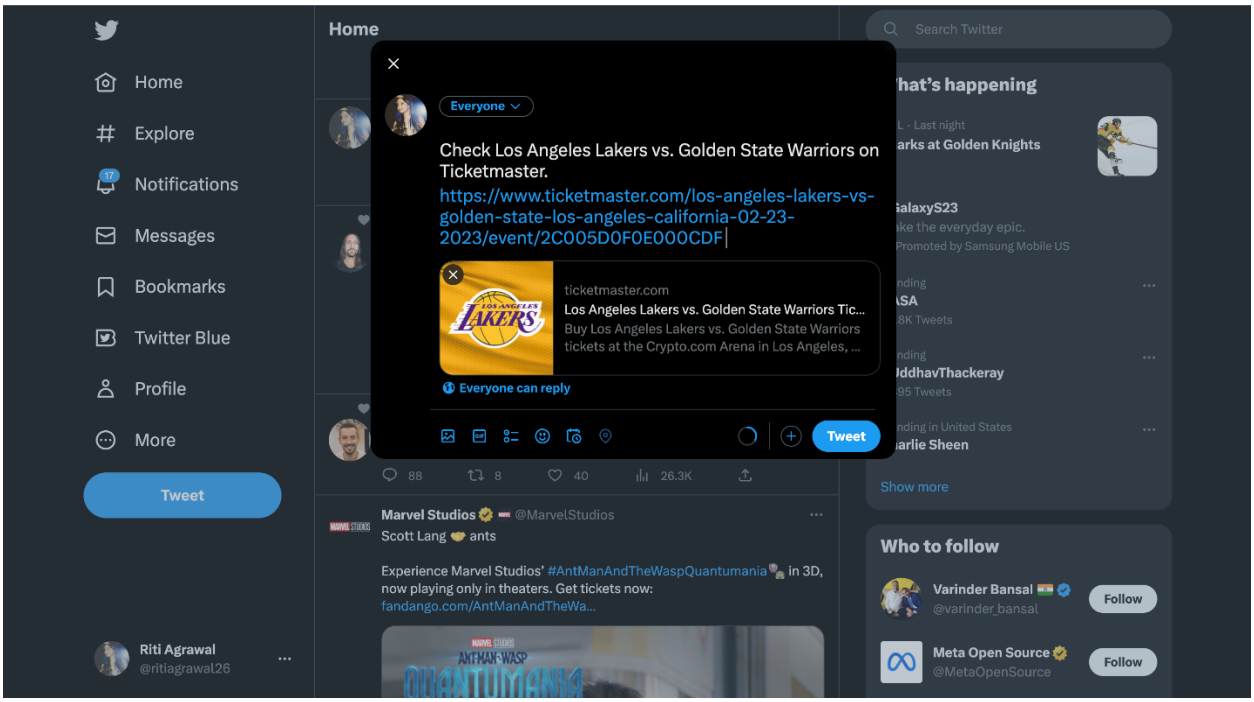


Figure 18: Example of tweet

5.3.2 "Artists/Team" tab

The **Artists/Team** tab will show information about the artist, like popularity, followers, Spotify link, and albums.

Since the artist information will be available on the Spotify API only if it is a music related event, you are supposed to show the artist/teams tab in the info page only if the event is a Music related event. In other events this tab should be disabled, see **Figure 21**.

The *Spotify API* is documented at:
https://developer.spotify.com/documentation/web-api/

After you register for the *Spotify API*, you need to create a project under the "dashboard" on the developer portal of Spotify. You will then be able to get your client id and client secret.

We recommend you use the official client for the *Spotify Web API* available here:
https://github.com/spotify/spotify-web-api-ts-sdk
The "Creating a client instance" section of this documentation explains how to use client ID and client secret to obtain the 'Access Token'."

For the API calls, you need to use the 'search' function with the "type" argument being a list of a single value "artist".

You will get the following result:

Figure 19: An Example Spotify API Call result

Fields in the artist table	**Corresponding response object fields**
Name	*The value of the "name" of the item*
Followers	*The value of the "follwers.total"*
Popularity	*The value of the "popularity"*
Spotify Link	*The value of the "external_urls.spotify"*

Table 2: Mapping the result from Spotify API into HTML table

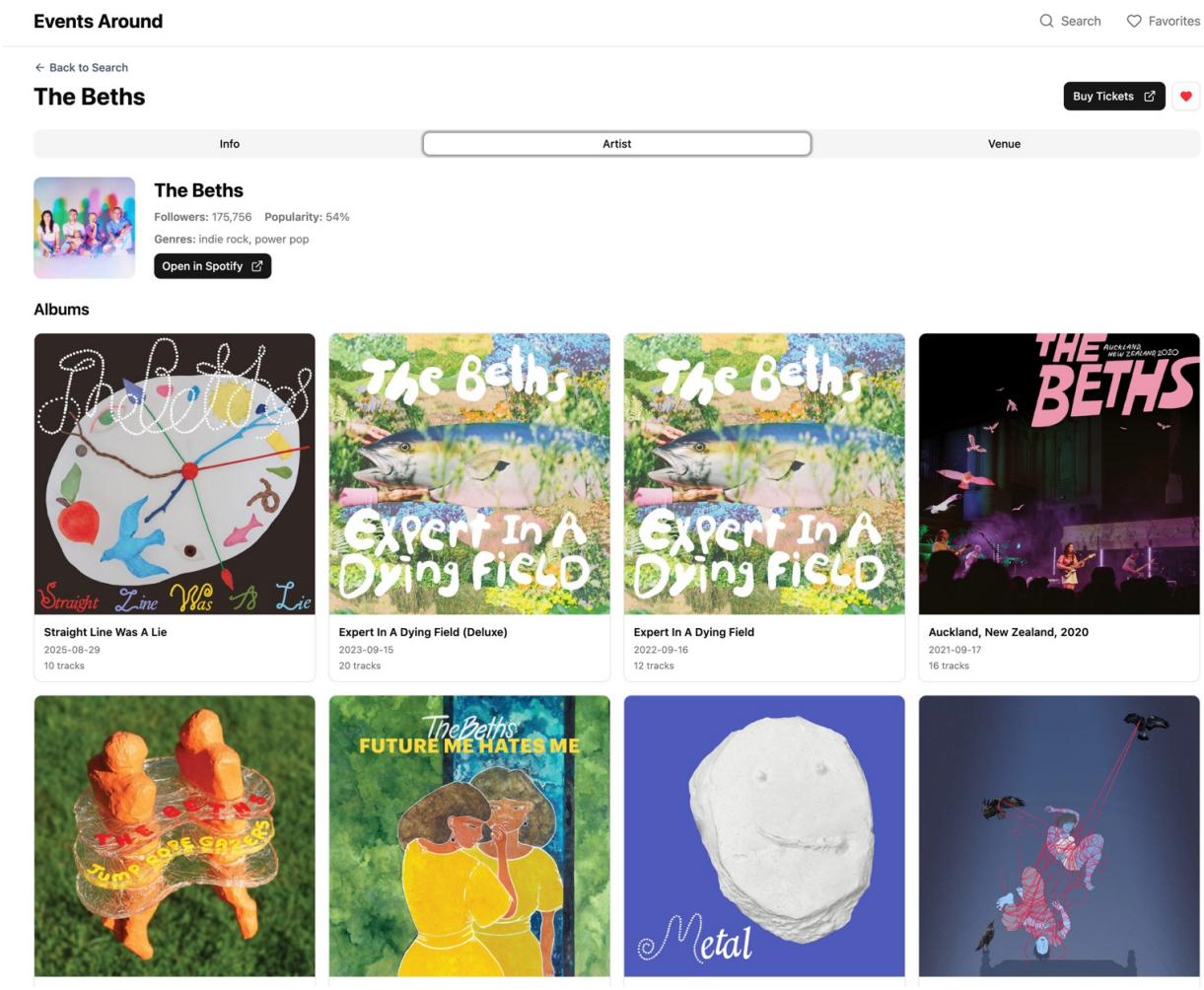


Figure 20: An Example Artists/Teams tab details

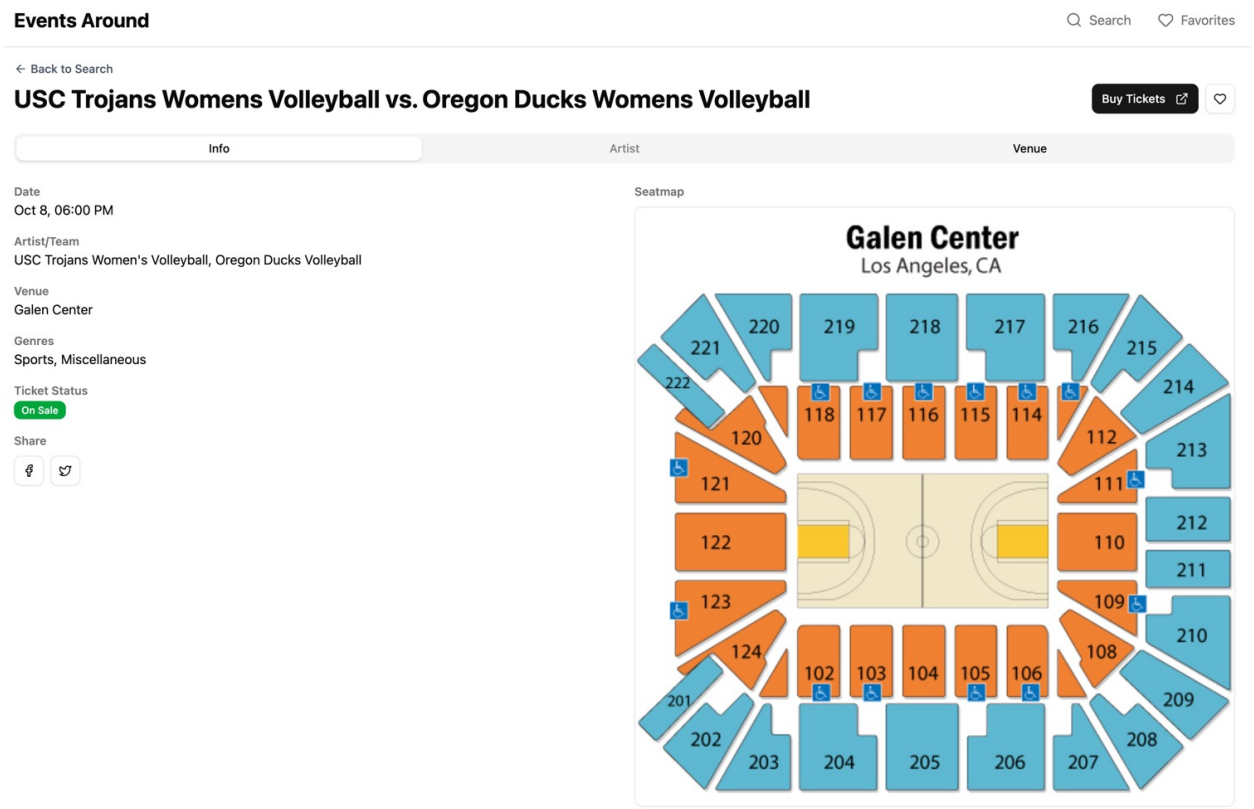


Figure 21: An Example of an event outside "Music" genre

Please note that the *Spotify API* may return more than one artist for each search keyword. Please choose the item or pick the first one and return it.

Format the **Followers** field using xxx,xxx,xxx, as shown in **Figure 20**. For the **Popularity** field, since the value will be 0-100%.

5.3.3 Venue tab

The **Venue tab** should display: Name, Address, "See Events" button, Cover image, Parking info, General Rule, and Child rule. See **Figure 22.**

To get the venue info, use the embedded venue information from the Event Details response.

A table containing the detailed info of the event venue is displayed in this tab (see **Table 3**). The table has the following **six** (6) fields if they are available in the detail search results:

Fields in the info table	**Corresponding response object fields**
Name	The value of the *"name"* attribute

“See Events” URL	The value of the *“url”* attribute
Image	The first value of “*images*” attribute
Address	You will need text address (*address.line, city.name, state.stateCode, state, coutryCode*) and coordinates (*location.latitude* and *location.longitude*)
Parking Info	The value of the “*parkingDetail*” attribute.
General Rule	The value of the “*generalRule*” attribute that is part of the “*generalInfo*” object.
Child Rule	The value of the “*childRule*” attribute that is part of the “*generalInfo*” object.

Table 3: Mapping the result from Venue Detail API into HTML table

NOTE: The Address field in Venue should be a concatenation of line1, city, and state fields of the venueDetail object, separated by commas. In case the API returns a result, which has one or more missing fields out of the above six, skip displaying that field in the table.

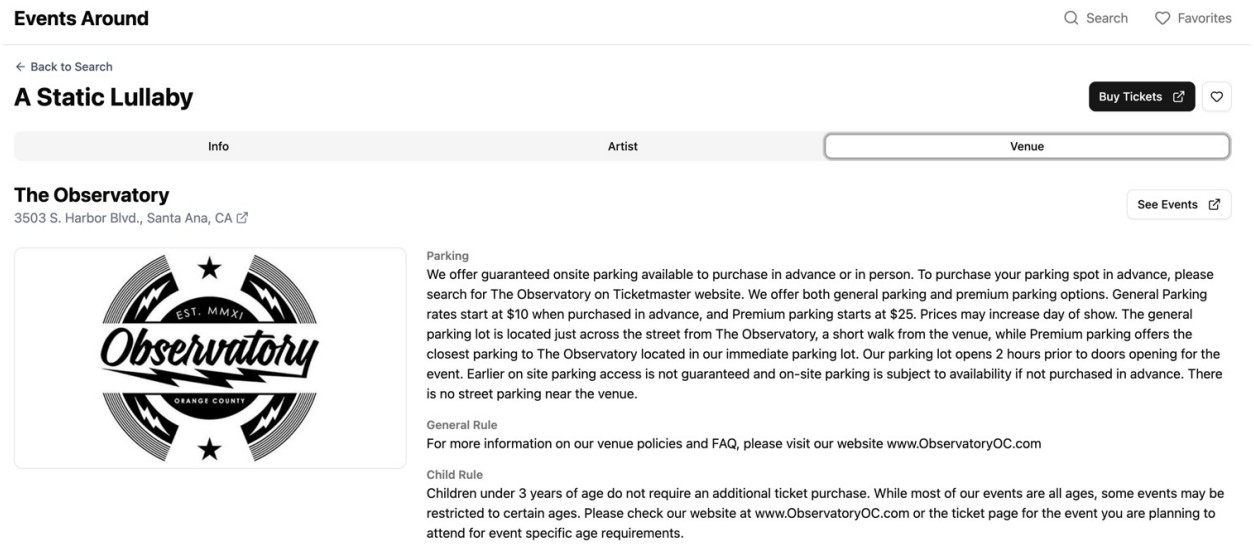


Figure 22: Example of an event having all mentioned information under Venue tab

Clicking on the address link should open Google Maps with location marker set at the coordinates from the response. Link should be opened in a new window.
“See Events” button should open a TicketMaster website with venue’s event list. Link should be opened in a new window.

5.4 Favorites

5.4.1 Favorite button

The Favorite button has two states: not favorite (transparent heart icon with black stroke) and favorite (red filled heart icon). Regardless of the location button is placed (event card, detail

page, favorite event card) button should show the correct state – for example, if user added event to favorites on the detail page, button should be filled on the search result card once user returns back to search. The state should also be retained after a page reload (if user closed the tab and later opened the application – your application should fetch the list of favorite events and display the correct state everywhere a Favorite button is used). Pressing a button should update the local state and send changes to the backend which in turns persists them in your database. Pressing a button also triggers a notification/toast/sonner, which is described in the next section.

5.4.2 Favorite notification (toast/sonner)

There should be 3 notifications:

1) *"... added to favorites!"*
   This notification is triggered, when user adds an event to favorites (by pressing a favorite button).
2) *"... removed from favorites!"*
   This notification is triggered when user removes an event from favorites (by pressing a favorite button). The notification should also have an undo button which reverts the action (by adding the event back to favorites) and trigger a "Re-added" notification.
3) *"... re-added to favorites!"*
   This notification is triggered only when user presses an undo button.

Please use sonner library or its port to the framework of your choice, this way you don't have to implement manually the following behaviors: displaying notifications in a stack view which is with unstacked on hover, swipe to close, appear/disappear/stack/unstack animations, auto-remove after timeout.

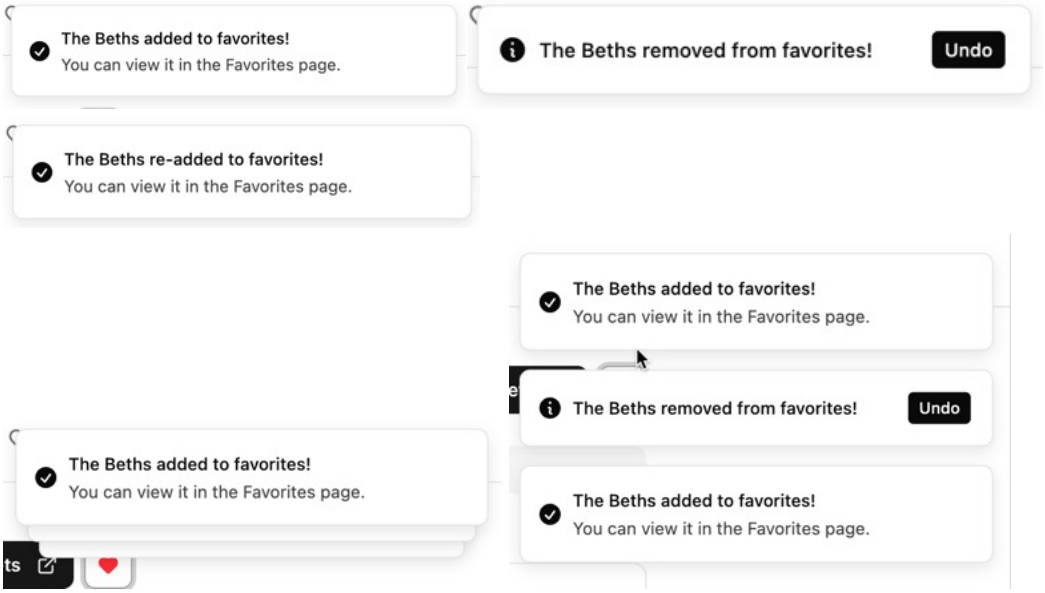


Figure 23: Example of different notifications and stack behavior

5.4.3 Favorite page

The Favorites route displays a list of events marked as “Favorites”. It fetches all the favorite events from a *MongoDB Atlas* database in Google Cloud (See section **5.5 Database**) and displays them in a grid view. The favorite event card is similar to event cards shown in the search results.

The events on the Favorites page are sorted in the order they are added to the favorites list. Please note if a user closes and re-opens the browser, its favorites list will still be there. If no favorites are available, display ‘No favorite events yet’ message. See **Figure 25.**

Note: All the CSS should be matched as shown in Images/Video.

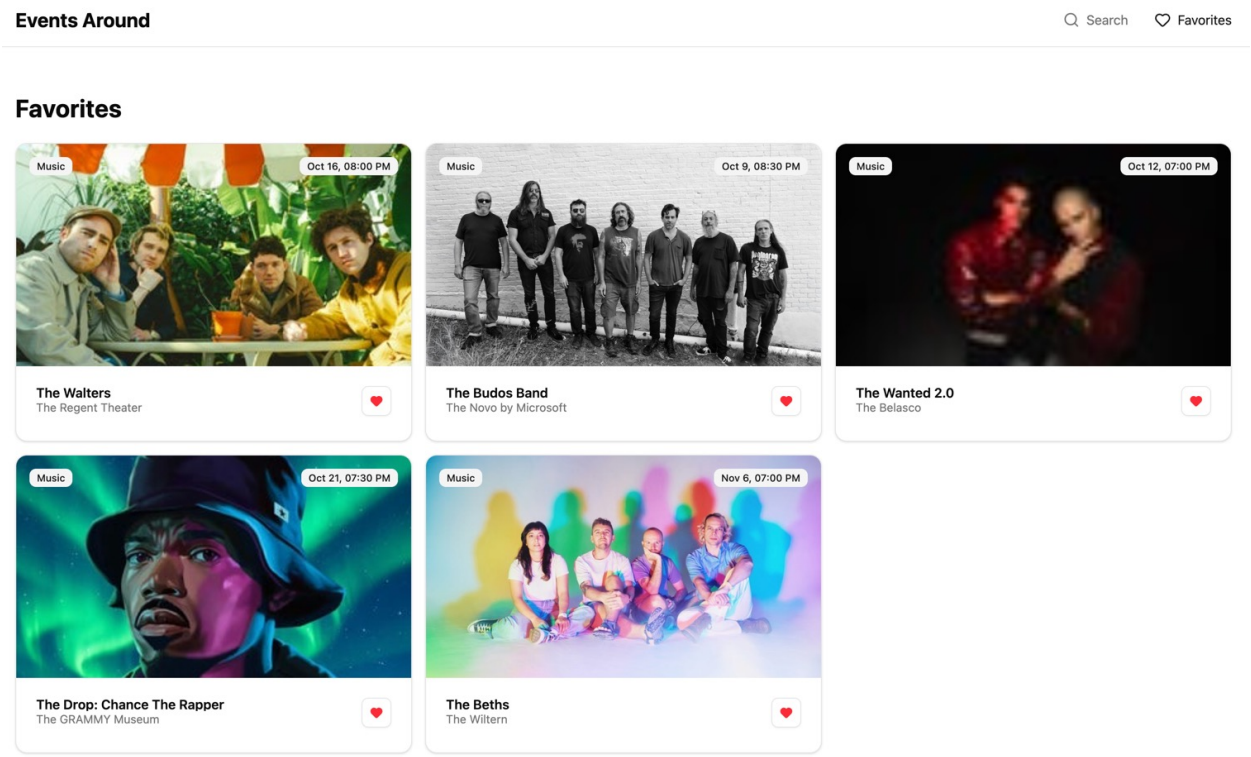


Figure 24: Example of list of Favorite events

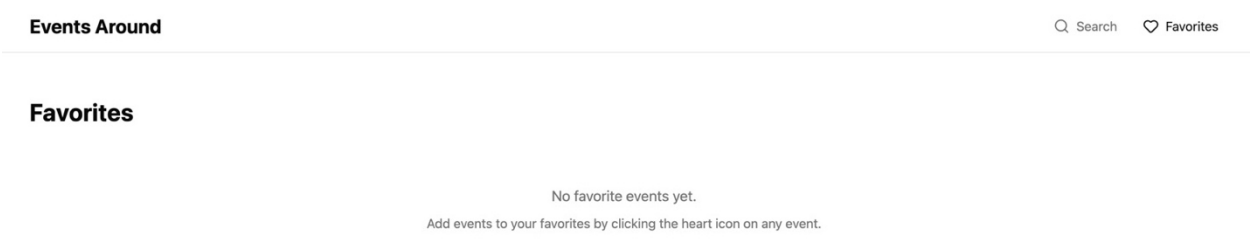


Figure 25: Example of empty list of Favorites.

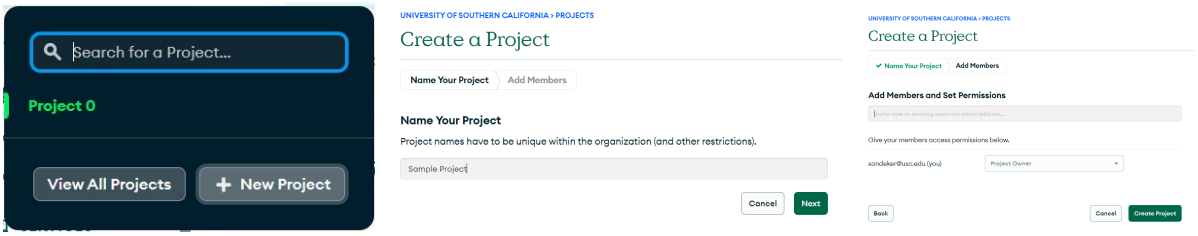
5.5 Database

In this assignment we will store users and their preferences in a database. For these purposes, we will use a NoSQL database, MongoDB. MongoDB is also available as a Cloud Service, called **MongoDB Atlas**.

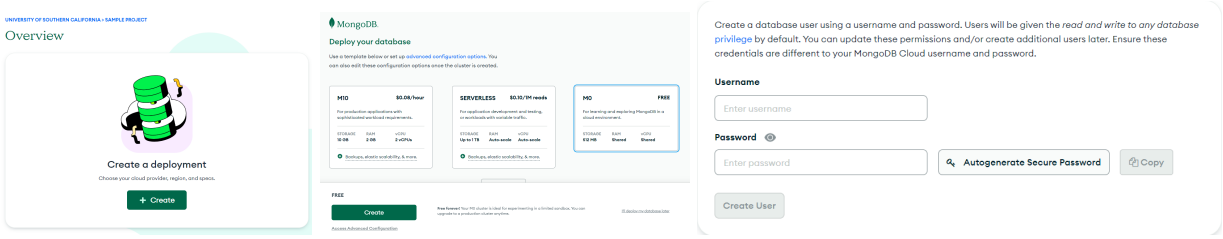
*MongoDB Atl*as is a source-available, cross-platform, document-oriented, DBMS. It is classified as a NoSQL DBMS (NoSQL Database Management System). A NoSQL DBMS is a non-relational database that is designed to handle large-scale, distributed, and flexible data storage. Unlike traditional SQL (relational) databases, NoSQL databases do not require fixed schemas, tables, or structured relationships.

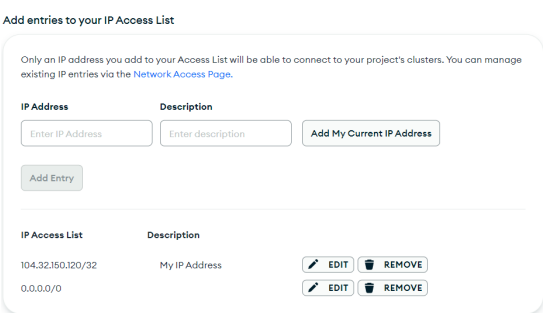
MongoDB Atlas uses JSON-like documents with optional schemas. For more information, see: https://www.mongodb.com/docs/

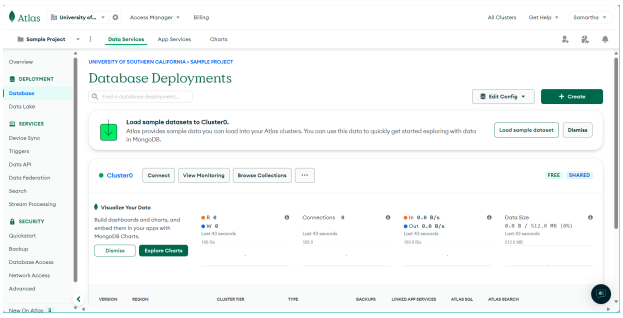
Also, see MongoDB on Google Cloud: https://www.mongodb.com/mongodb-on-google-cloud
Once you set up an account in MongoDB Atlas, you will have to create a project to store your databases. Below are the steps to set up a project and create a database.
Follow these steps to create a project in which you can store databases for your application.



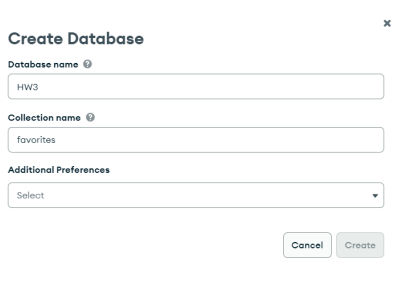
Follow these steps to create a deployment for the project by **creating a cluster** and providing user access and network access for the same. This would allow your application, running on a different server, to connect to MongoDB Atlas in the cloud.

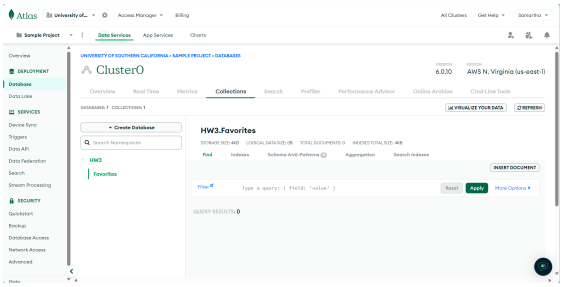




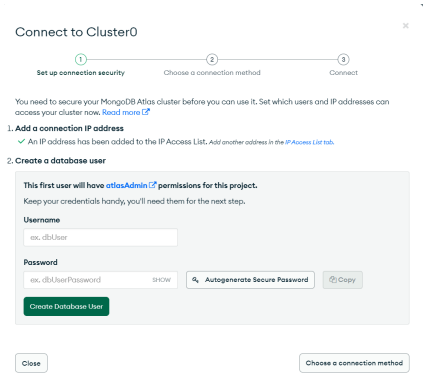


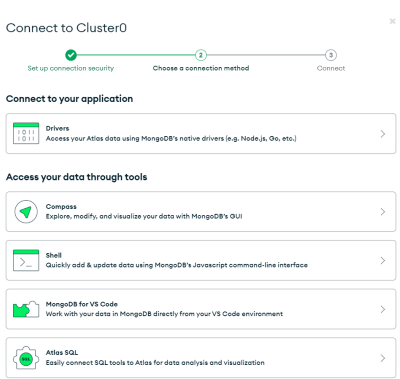
Follow these steps to **create a database** and a **collection** in that database. MongoDB stores data records as documents (specifically in BSON format) which are gathered in collections. A database stores one or more collections of documents.

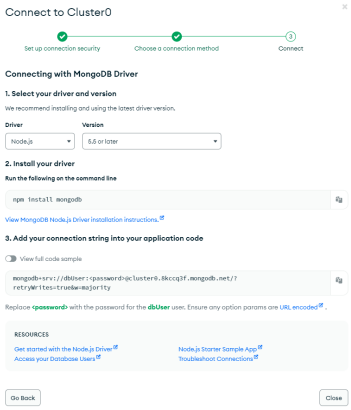




Follow these steps to provide the instructions on how to **connect to your MongoDB database** from your application.





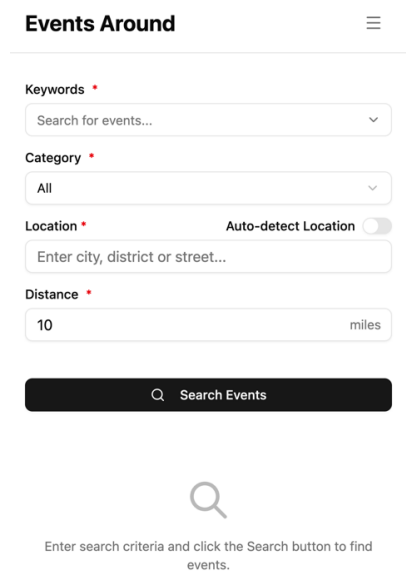


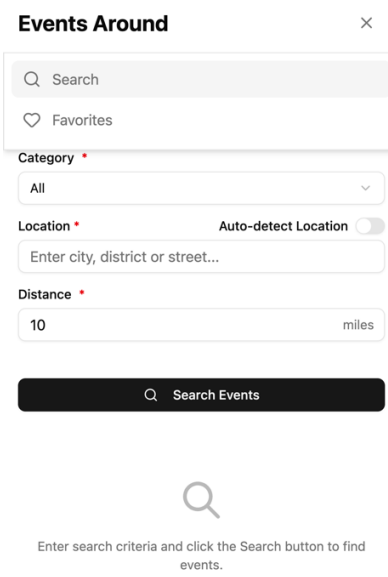
Once you have set up the database and the collection, you can connect to the database by adding the MongoDB node driver to your application. For more information on how to add the driver and run queries on the database, refer to [MongoDB Node Driver — Node.js.]

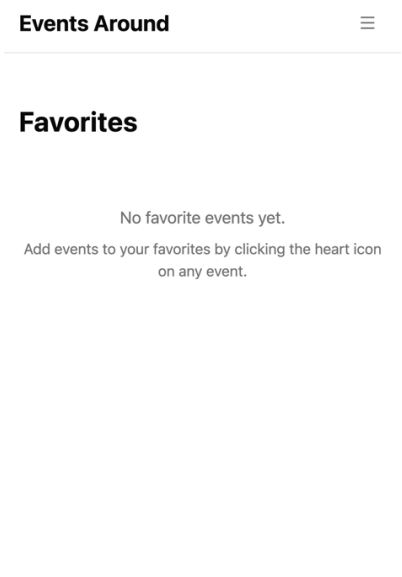
Make sure to set up accesses in the “Database Access” and “Network Access” side menu to give proper access to your client and allow certain IP addresses to access your database and collection(s). We recommend creating a dedicated user that will only have “ReadWrite” access to your collection(s) and set up an IP filtering such that it would match your IP for local development and the IP of cloud deployment. WE DO NOT RECOMMEND HAVING 0.0.0.0/0 entry – this will allow anyone to try and access your MongoDB instance.

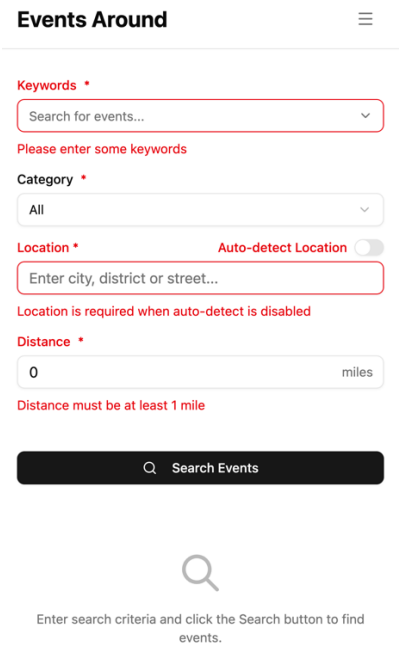
6. Responsive Design

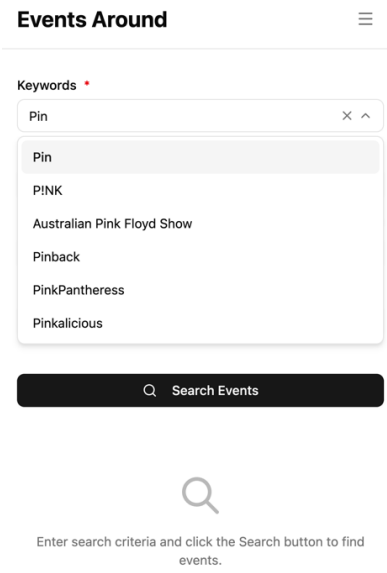
The webpage you develop must be responsive. One easy way to test responsive behavior is to use Google Chrome Responsive Design Mode in the Developer Console. Here are some snapshots for an **iPhone 14 Pro Max** using the Google Chrome Responsive Design Mode. All functions should work on mobile devices.

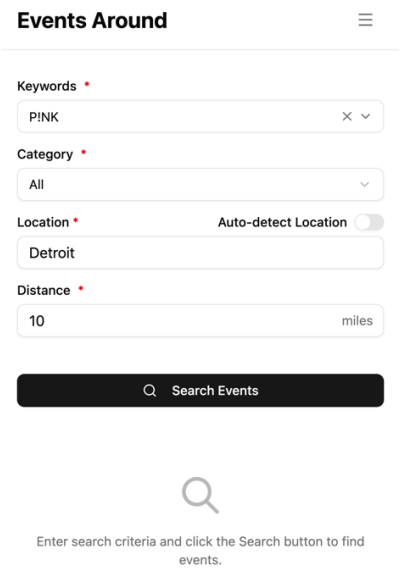

Events Around
Keywords *
Search for events...
Category *
All
Location *
Auto-detect Location
Enter city, district or street...
Distance *
10
miles
Search Events
Enter search criteria and click the Search button to find events.


Events Around
Search
Favorites
Category *
All
Location *
Auto-detect Location
Enter city, district or street...
Distance *
10
miles
Search Events
Enter search criteria and click the Search button to find events.


Events Around
Favorites
No favorite events yet.
Add events to your favorites by clicking the heart icon on any event.


Events Around
Keywords *
Search for events...
Please enter some keywords
Category *
All
Location *
Auto-detect Location
Enter city, district or street...
Location is required when auto-detect is disabled
Distance *
0
miles
Distance must be at least 1 mile
Search Events
Enter search criteria and click the Search button to find events.


Events Around
Keywords *
Pin
Pin
P!NK
Australian Pink Floyd Show
Pinback
PinkPantheress
Pinkalicious
Search Events
Enter search criteria and click the Search button to find events.


Events Around
Keywords *
P!NK
Category *
All
Location *
Auto-detect Location
Detroit
Distance *
10
miles
Search Events
Enter search criteria and click the Search button to find events.

Events Around

Keywords *
Los Angeles Lakers

Category *
All

Location Auto-detect Location
Location will be autodetected

Distance *
10 miles

Search Events

Enter search criteria and click the Search button to find events.

Events Around

Location will be autodetected

Distance *
10 miles

Search Events

Sports Oct 23, 07:00 PM

Los Angeles Lakers vs. Minnesota Timberwolves
Crypto.com Arena

Sports Nov 1, 06:30 PM

Los Angeles Lakers vs. Miami Heat
Crypto.com Arena

Events Around

← Back to Search

Los Angeles Lakers vs. Phoenix Suns

Buy Tickets

Info | Artist | Venue

Date
Nov 30, 07:00 PM

Artist/Team
Los Angeles Lakers, Phoenix Suns

Venue
Crypto.com Arena

Genres
Sports, Basketball, NBA, Group, Team

Ticket Status
On Sale

Share

Seatmap

Los Angeles Lakers vs. Phoenix Suns added to favorites!
You can view it in the Favorites page.

Los Angeles Lakers vs. Phoenix Suns

Buy Tickets

Info | Artist | Venue

Date
Nov 30, 07:00 PM

Artist/Team
Los Angeles Lakers, Phoenix Suns

Venue
Crypto.com Arena

Genres
Sports, Basketball, NBA, Group, Team

Ticket Status
On Sale

Share

Seatmap

Los Angeles Lakers vs. Phoenix Suns removed from favorites! Undo

← Back to Search

Los Angeles Lakers vs. Phoenix Suns

Buy Tickets

Info | Artist | Venue

Date
Nov 30, 07:00 PM

Artist/Team
Los Angeles Lakers, Phoenix Suns

Venue
Crypto.com Arena

Genres
Sports, Basketball, NBA, Group, Team

Ticket Status
On Sale

Share

Seatmap

Events Around

← Back to Search

The Beths

Buy Tickets

Info | Artist | Venue

Date
Nov 6, 07:00 PM

Artist/Team
The Beths, Bret McKenzie, Phoebe Rings

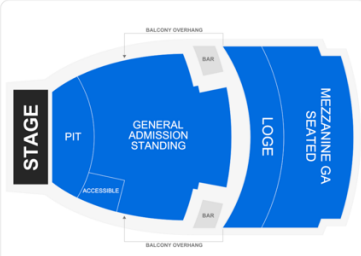
Venue
The Wiltern

Genres
Music, Rock, Pop

Ticket Status
On Sale

Share

Seatmap



Events Around

← Back to Search

The Beths

Buy Tickets

Info | Artist | Venue

The Beths

Followers: 175,756

Popularity: 54%

Genres: indie rock, power pop

Open in Spotify

Albums

Straight Line Was A Lie
2025-08-29
10 tracks

Expert In A Dying Field (Deluxe)
2023-09-15
20 tracks

Events Around

← Back to Search

The Beths

Buy Tickets

Info | Artist | Venue

The Wiltern

3790 Wilshire Blvd., Los Angeles, CA

See Events

Parking

We offer premium Wiltern Underground parking and upgrades on select shows. Otherwise, there are several paid parking lots in the area. The most convenient lot is the Ralphs parking structure, adjacent to the venue. Enter from Oxford street and only through the north entrance - not the Ralphs entrance. (We do not own or operate the Ralph's lot).

General Rule

General rules: please update to include this link: https://www.wiltern.com/visit

Child Rule

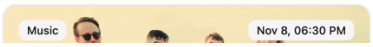
All patrons must have a ticket regardless of age. Some

Events Around

The Revivalists
The Novo by Microsoft

The Cult
Shrine Auditorium-CA

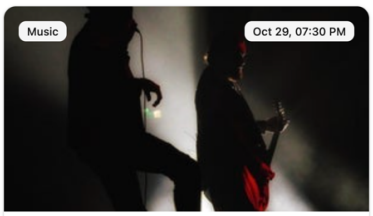
The Beths
The Wiltern



Events Around

Favorites

The Beths
The Wiltern



The Cult
Shrine Auditorium-CA

The Cult added to favorites!
You can view it in the Favorites page.

The Cult

Buy Tickets

Info | Artist | Venue

Date

Oct 29, 07:30 PM

Artist/Team

The Cult

Venue

Shrine Auditorium-CA

Genres

Music, Rock, Hard Rock

Ticket Status

On Sale

Share

Seatmap

7. API Documentation

7.1 Spotify API

To use the *Spotify API*, you need first to register a Spotify Account. Then create an application and get your client id and client secret.

https://developer.spotify.com/dashboard/#

Please refer to the documentation doc and the Spotify NodeJS libraries, documented at: https://github.com/spotify/spotify-web-api-ts-sdk

8. Libraries

- **Node-geohash -** https://github.com/sunng87/node-geohash for geo hash conversion
- **Angular Google Maps** - https://angular-maps.com/ This makes it easier to use Google Maps in Angular

You can use any additional Angular libraries and Node.js modules you like.

Shadcn components: badge, button, card, form, input, label, select, sonner, switch, tabs

Shadcn style: new-york

Icon library: lucide

9. Implementation Hints

9.1 Images

The only static image in this project is the favicon. Feel free to find any similar free icon online.

9.2 Icons

For icons, please use lucide icons. You may use lucide port to your framework of choice. For any missing icon you can find a substitution online or ask LLM to create one in svg format.

The following icons were used: Heart, FacebookIcon, TwitterIcon, Search, ExternalLink, ChevronUp/ChevronDown, X, Loader2, Check, Menu, Arrow Left.

9.3 Shadcn Components

The reference implementation uses the following components from shadcn:

- badge
- button
- card

- from
- input
- label
- select
- sonner
- switch
- tabs

9.4 Google App Engine

You should use the domain name of the GAE service you created in **Google Cloud Setup (NodeJS)** on D2L Brightspace to make the request. For example, if your server domain is called example.appspot.com, the JavaScript program will perform a GET request with keyword="xxx", and an example query of the following type will be generated, using *Google App Engine*:

https://example.appspot.com/searchEvents?keyword=xxx

When using *Google Cloud Run*, the URL will like this pattern:

https://<serviceName>-<projectHash>-<region>.run.app

as in:

https://gateway-12345-uc.a.run.app/searchEvents?keyword=xxx

Your URLs don't need to be the same as the ones above. You can use whatever paths and parameters you want. Please note that in addition to the link to your Assignment #3, you should also **provide a link like this URL in the table of your Node.JS backend**. When your grader clicks on this additional link, the response should return a JSON object with appropriate data.

9.5 AJAX call

You should send the request to the Node.js script(s) by calling an Ajax function. You **must use a GET method** to request the resource since you are required to provide this link to your homework list to let graders check whether the Node.js script code is running in the "cloud" on Google GAE or Cloud Run (see 9.4 above). Please refer to the grading guidelines for details.

9.6 CORS issues

To allow your frontend to call the backend without triggering CORS issues, it's usually easier to serve API endpoints under the same hostname with the `/api/...` prefix. This way, you don't need to modify your frontend configuration before deploying your app to Google Cloud. In this setup, your frontend can call `/api/...` directly without specifying a fully qualified URL (e.g., `fetch("/api/")`), as the frontend's origin will automatically be used.

9.7 Debugging

For local development and debugging, you can enable proxying all `/api` calls to your backend using vite *server.proxy* configuration. Vite will use this file to set up a proxy. Ensure your backend is running before making requests.

For Angular: please refer to the documentation here: https://v17.angular.io/guide/build#proxying-to-a-backend-server.

10. Assignment Submission

In your course Table of Assignments page (on GitHub Pages), update the Assignment 3 link to refer to your deployed website. Also, provide an additional link to one of your cloud query entry points, below the homepage link. Your project must be hosted on Google Cloud. Graders will verify that these links are indeed pointing to one of the listed cloud platforms.

Also, submit your source code files to D2L Brightspace as a ZIP archive. Include your frontend and backend source code, plus any additional files needed to build your app (e.g., yaml file). The timestamp of your last submission will be used to verify if you used any grace days. Make sure you don't upload the `node_modules` in the zip file and also make sure 'package.json' is added in the zip file.

11. Notes

- You can use **React or Vue.js** in lieu of **Angular**, but the look of the app must be the same, or penalties will be assessed. Also note that there will be no support on Piazza for React.
- You can use Bootstrap for this assignment, but can also use any other Responsive Design framework, especially if the code is AI generated.
- You can use Fastify, Hono or the Elysia frameworks instead of Express.
- The appearance of the webpage should be like the reference videos/screenshots as much as possible.

12. Additional References

****IMPORTANT**:**

- All discussions and explanations in Piazza related to this homework are part of the homework description and grading guidelines. So please review all Piazza threads, before finishing the assignment. If there is a "clarification" on Piazza that conflicts with this description and/or the grading guidelines, **Piazza always rules**. In most cases, the clarification is related to an error in the description, or missing information from the description, but **no additional functionality**.

- You can use FaaS like *Google Cloud Functions*, *or Cloud Run*, in lieu of building a monolithic application.
- You **should not call any of the Ticketmaster APIs directly from JavaScript**, bypassing the NodeJS proxy. Implementing any one of them in JavaScript instead of Node will result in a **4-point penalty.** Other APIs can be called from JavaScript.
- You may call the Google Maps Geocoding API directly from JavaScript.
- **APPEARANCE OF CARD VIEW should be as similar as possible to the reference video.**