Technical Documentation

Future Room for Expansion

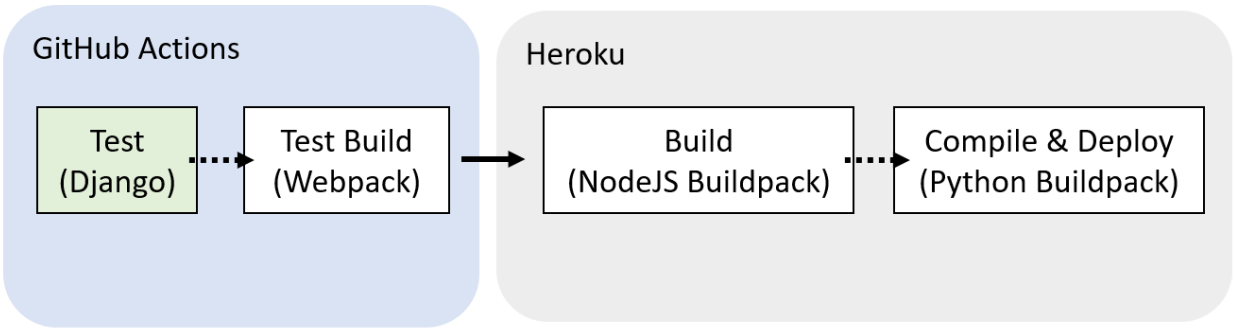
- Increase social media API integrations
    - get access to Clubhouse and LinkedIn API
        - Current WIP as Bobby Umar has reached out to both companies.
        - Once we have API access, we will need to build a new API-controller that maps the API data to fit our database schema, thus saving each event into a format the rest of the application is able to read.
        - For a working implementation, see the `TwitterController`
    - Integrate other social audio event platforms
        - requires getting API access, followed by implementing a new API-controller to process the data according to our database schema
        - For a working implementation, see the `TwitterController`
- Attach an official Thought Leadership Branding Club URL to the website.
    - go through the Heroku hosting platform and attach an more suitable URL to the application so it is easier to google.
    - Potentially integrate this web-page into the main Thought Leadership Branding Club website
- Improved Search filtering
    - would be useful to create a Google-style search which allows for keywords, phrases, wildcards, local operators, etc. instead of our current search filter which simply find keywords in common between our search parameter and each event listing.

- Add an “about” or “landing” page to inform new users about the purpose of the website
- Improve the styling of the EventCards:
  - Add a tab to the left of each card showing the platform, as illustrated in our Figma Mock-up. You can use the colour and logo definitions received from the `getPlatforms` endpoint
  - Colour-code category tags, based on the colour definitions retrieved from the `getCategories` internal endpoint
- Implement log-in and allow users to post about their own events through an input form
  - Allow users to save their favourite events to a calendar (in-app or the user’s external calendar)
  - Allow the user to save filter presets to facilitate repeated use of the app

Statement of Completed Work

- Implementation of Twitter API to get live Twitter spaces
- Testing of the back-end and database
- Support of mocked data for Twitter Spaces, LinkedIn Audio, and Clubhouse
- GitHub actions to test, build, and deploy the application
- Application hosted on Heroku
- Production database hosted on MongoDB Atlas
- Filtering functionality (live, categories, start/end times, platform)
- Share audio event functionality (Twitter, WhatsApp, link sharing)
- Basic keyword extraction and search functionality

Production Deployment (Heroku, MongoDB Atlas)



The overall deployment process is outlined above. There are two main segments on two different platforms: GitHub Actions and Heroku. The database does not require a deployment pipeline.

GitHub Actions is primarily responsible for testing, and orchestrating the deployment process. On any open Pull Request to the main branch, the “test” workflow will run (highlighted in green above). Then, on any push to the main branch, the “test, build, and deploy” workflow will run. This will run all 4 of the steps above.

The intention is that any errors in either the back end or the front end will be caught within GitHub Actions before Heroku is involved in the deployment process.

Relevant Environment Variables

GitHub

Key	**Description**
`HEROKU_API_KEY`	Heroku API key for deployment (must be the key for the Heroku account of the email).
`HEROKU_APP_NAME`	Heroku application name.
`HEROKU_EMAIL`	Heroku email (must correspond with the API key).

Heroku

Key	Description
`DJANGO_DEBUG`	Debug flag for Django, this MUST be set to 0 to ensure the application does not expose internal errors.
`DJANGO_SECRET_KEY`	Django's secret key (i.e. Django's salt) for various encryption uses).
`MONGODB_CONNECTION_STR`	Connection string for the MongoDB database. This can be generated from MongoDB Atlas.
`EXTERNAL_API_TWITTER_BEARER`	Twitter API key. This can be generated from the Twitter developer portal.

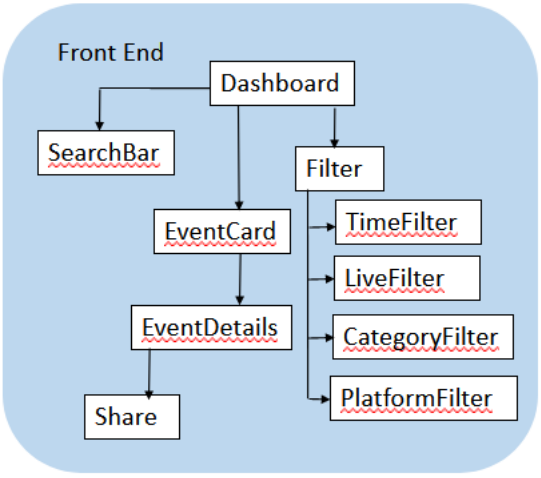
MongoDB Atlas

MongoDB Atlas is used to host the MongoDB database. See the links and credentials document for access.

On the deployed database, there are some additional indices (eventCategory, eventPlatform + eventId) to speed up queries and aggregation. Other indices (for the other fields) were attempted but were deemed to not be useful enough to warrant the additional storage space and query time requirements.

Front-end Components

You can find the Figma Mock-up here. Our front-end is currently served by the Django back-end. The basic hierarchy of components is as follows:



`Dashboard`, `EventCard`, and `EventDetails` are class components, but the team decided to use functional components moving forward as they are the current industry standard. The Dashboard component handles most of the communication with the back-end so that information about which events to display can be communicated between the filters and the event cards.

High-level description:

- `Dashboard` is our main container, housing all the components of the web-page

- `Filter` (and it's components) form the filter side-bar and update the Dashboard's `selected_*` states
- `SearchBar` is the search bar at the top of the web-page and handles search input.
- `EventCard` represents the event items displayed on the home page, and `EventDetails` is the pop-up that results from clicking on a card. `Share` handles the functionality of the share button on said pop-up.

Database Schema

> When including a new API connection, we must break down the event data into something the rest of the application can understand. To do this, see the following `events` collection which outlines the data structure that must be satisfied in order to save an event in the database, and subsequently, display it on the Dashboard.

> The schemas are presented in the JSON Schema format.

Database: tlbc

Collection: `events`

```
{
    "title": "Event",
    "description": "A social audio event.",
    "type": "object",
    "properties": {
        "_id": {
            "description": "MongoDB unique identifier for an event. (Automatically generated)",
            "type": "string"
        },
        "eventPlatformId": {
            "description": "Plaform specific ID. May not be unique across events.",
            "type": "string"
        },
        "eventPlatform": {
            "description": "Social audio platform for an event.",
            "type": "string"
        },
        "eventDateTime": {
            "description": "Start date and time for an event in the ISO8601 format.",
            "type": "string"
        },
        "eventName": {
            "description": "Name of an event.",
            "type": "string"
        },
        "eventDescription": {
            "description": "Description of the event.",
            "type": "string"
        },
        "eventHost": {
            "description": "Host information for an event.",
            "type": "object",
            "properties": {
                "hostUsername": {
                    "description": "Username of a host. Likely unique for a given social audio platform.",
                    "type": "string"
                },
                "hostDisplayName": {
                    "description": "Display name of a host.",
                    "type": "string"
                }
            },
            "required": ["hostUsername"]
        },
        "eventCategory": {
            "description": "Category for an event. This may differ from a platform's category.",
```

```
            "type": "string"
        },
        "eventLink": {
            "description": "URL to the event.",
            "type": "string"
        },
        "eventImage": {
            "description": "URL to an image for an event.",
            "type": "string"
        },
        "eventKeywords": {
            "description": "Keywords for an event.",
            "type": "array",
            "items": {
                "type": "string"
            },
            "minItems": 1,
            "uniqueItems": true
        },
        "eventLive": {
            "description": "Boolean flag representing whether or not this event is live.",
            "type": "boolean"
        },
        "createDateTime": {
            "description": "Date and time of document creation in the ISO8601 format.",
            "type": "string"
        },
        "updateDateTime": {
            "description": "Date and time of document update in the ISO8601 format.",
            "type": "string"
        }
    },
    "required": ["eventId", "eventPlatform", "eventDateTime", "eventName", "eventHost", "eventLink", "eventKeywords", "eventLive", "createD
}
```

Example:

```
{
  "_id": "123456789",  # something MongoDB generates, unique among all events
  "eventId": "9876",  # generated by platforms, unique among the platform's events
  "eventPlatform": "twitter",
  "eventDateTime": "2022-11-02T13:09:00Z",  # ISO 8601, prefer the "Z" for UTC timezone rather than a plus-minus offset
  ...,
  "eventHost": {
    "hostUsername": "asdf",  # may be unique among platforms (likely unique)
    "hostDisplayName": "A S D F"  # may not be defined
  },
  ...,
  "eventKeywords": ["stock", "apple"],
  ...
}
```

> To add another platform, one must create a new instance of said platform in the database following this schema:

Collection: `platforms`

```
{
    "title": "Platform",
    "description": "A social audio event platform.",
    "type": "object",
    "properties": {
        "_id": {
            "description": "Platform name in lowercase with no spaces and special characters.",
            "type": "string"
        }
        "platformDisplayName": {
            "description": "Display name for this platform.",
```

```
            "type": "string"
        },
        "platformIconLink": {
            "description": "Link to an icon for this platform.",
            "type": "string"
        },
        "platformColour": {
            "description": "Colour to represent this platform. Specified in hex.",
            "type": "string
        }
    },
    "required": ["_id", "platformDisplayName"]
}
```

> To add new categories, one must create a new instance of said category in the database following this schema:

Collection: `categories`

```
{
    "title": "Category",
    "description": "A social audio event category.",
    "type": "object",
    "properties": {
        "_id": {
            "description": "Category name in lowercase with no spaces and special characters.",
            "type": "string"
        }
        "categoryDisplayName": {
            "description": "Display name for this category",
            "type": "string"
        }
    },
    "required": ["_id", "categoryDisplayName"]
}
```

Collection: `watermarks`

```
{
    "title": "Watermark",
    "description": "Watermarks for the last updated time of the platforms",
    "type": "object",
    "properties": {
        "_id": {
            "description": "Platform name in lowercase with no spaces and special characters.",
            "type": "string"
        }
        "watermark": {
            "description": "Watermark for this platform.",
            "type": "string"
        }
    },
    "required": ["_id", "watermark"]
}
```

Endpoints

> When connecting to our backend API, one must call one of the following *endpoints* to make a request. All specifications for each of the endpoints are described out below…

Variation on ISO 8601 time format with no timezone (but ensure all times are in the UTC timezone): `YYYY-MM-DDThh:mm:ss`

- Example: `2022-10-31T23:37:30`

Event List

`/api/events/list`

- Methods: GET
- Description: Get a list of events (basic info)
- Query Parameters
  - `limit=<int>`
    - Maximum number of responses to return
  - `next=<str>`
    - Return events with a timestamp (ISO 8601) greater than or equal to `next`
  - `until=<str>`
    - Return events with a timestamp (ISO 8601) less than or equal to `until`
  - `platform=<str>` (for multiple, separate with commas, `...?platform=twitter,linkedin...`)
    - Include only these platforms in the response (case insensitive)
  - `category=<str>` (for multiple, separate with commas, `...?category=health,foodandlifestyle...`)
    - Include only these categories in the response (case insensitive)
  - `live=<str>`
    - Return events that are either live (`live=true`) or scheduled (`live=false`) [case insensitive]
    - If this parameter is not included, both live and scheduled events will be included
  - `query=<str>` (for multiple, separate with commas, `...?query=stocks,crypto...`)
    - Filter for events containing only these keywords (case insensitive)
- Response

```
{
  "next": "string used for the `next` parameter for pagination",
  "events": [
    {
      "_id": ...,
      "eventName": ...,
      "eventHost": {
        "hostUsername": ...,
        "hostDisplayName": ...
      },
      "eventCategory": ...,
      "eventPlatform": ...,
      "eventDateTime": ...
    },
    ...
  ]
}
```

Note that if the `query` string is specified, the returned events will also include two other keys: `keywordsCount` and `keywordsMatched`. `keywordsMatched` will be the specific keywords that matched in the query, and `keywordsCount` is the length of `keywordsMatched`.

Event Detail

`/api/events/detail/<str:event_id>`

- Methods: GET
- Description: Get detailed information for a single event
- Response
  - A single `event` as defined by Database Schema

Category List

`/api/categories/list`

- Methods: GET
- Description: Get a list of all categories
- Response (see Database Schema for all keys)

```
{
    "categories": [
        {"_id": ..., "categoryDisplayName": ..., ...},
        {"_id": ..., "categoryDisplayName": ..., ...},
        ...
    ]
}
```

Platform List

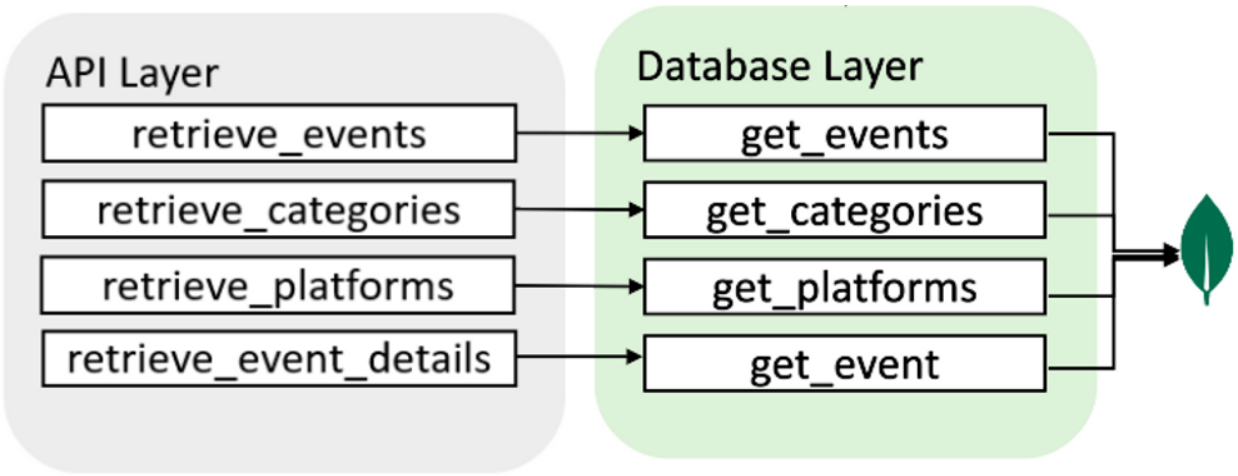
`/api/platforms/list`

- Methods: GET
- Description: Get a list of all supported platforms
- Response (see Database Schema for all keys)

```
{
    "platforms": [
        {"_id": ..., "platformDisplayName": ..., ...},
        {"_id": ..., "platformDisplayName": ..., ...},
        ....
    ]
}
```

Database Access Layer

The relationships between the backend API endpoints, which is referred to as API Layer, and the database endpoints, which is referred to as Database Layer, are visualized below:



Backend - Database

`core.database.get.get_event(_id: str) -> dict[str, Any]`

Retrieve all details of one event.

Params

- `_id`: string ID for an event

Returns

- Event information as a dictionary. See Database Schema for possible keys and values.

`core.database.get.get_events(limit: int = None, next: datetime.datetime = None, platforms: list[str] = None, categories: list[str] = None, search: list[str] = None) -> list[dict[str, Any]]`

Retrieve events matching the given criteria.

Params

- `limit`: maximum number of events to retrieve
- `next`: include only events with a start timestamp of `next` and onwards
- `until`: include only events with a start timestamp of before or at `until`
- `platforms`: include only these platforms
- `categories`: include only these categories
- `search`: list of keyword search terms

Returns

- All events with all details as a list. See Database Schema for possible keys and values of each list element.
- Note that if `search` is not `None`, each event will also have two additional keys: `keywordsCount` and `keywordsMatched`. `keywordsMatched` will be the specific keywords that matched in the query, and `keywordsCount` is the length of `keywordsMatched`.

`core.database.get.get_platforms() -> list[dict[str, Any]]`

Get all supported platforms and related information.

Returns

- All supported platforms with related info (display name, colours, icon, etc.)

`core.database.get.get_categories() -> list[dict[str, Any]]`

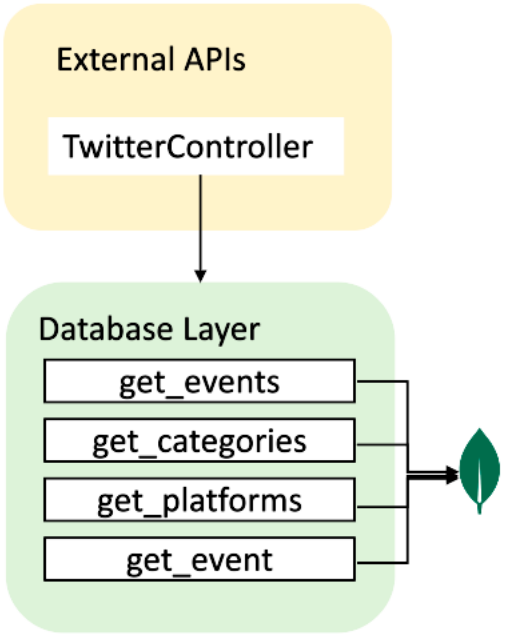
Get all supported categories and related information.

Returns

- All supported categories with related info (display name, etc.)

API - Database

The relationship between the Database Layer and the External API is visualized below:



`core.database.insert.insert_many(events: list[dict[str, Any]]) -> int`

Insert events into the database.

Params

- `events`: list of events to add. These events must abide by the database schema in Database Schema .

Returns

- Number of events successfully added.

`core.database.upsert_many(events: list[dict[str, Any]]) -> list[dict[str, Any]]`

Upsert many events into the database. All events MUST be the same platform. Additionally, every element of the `events` list MUST include two keys: “eventPlatform” and “eventId”.

Params

- `events`: events to upsert

Returns

- previous versions of the documents that were updated

Database Utilities

`core.database.utils.get_platform_watermark(platform: str) -> datetime.datetime`

Get the last updated time watermark for the given platform.

Params

- `platform`: platform to retrieve the watermark for

Returns

- Timestamp of the last database update for that platform

Category Data

These documents **MUST** be manually populated into the database and are also included in the `core/mock_data` folder. Create a new collection within the `tlbc` database called `categories` and insert the following document.

`categories` collection:

```
[
  {"_id": "technology", "categoryDisplayName": "Technology"},
  {"_id": "food", "categoryDisplayName": "Food"},
  {"_id": "animals", "categoryDisplayName": "Animals"},
  {"_id": "religion", "categoryDisplayName": "Religion"},
  {"_id": "physics", "categoryDisplayName": "Physics"},
  {"_id": "economy", "categoryDisplayName": "Economy"},
  {"_id": "business", "categoryDisplayName": "Business"},
  {"_id": "health", "categoryDisplayName": "Health"},
  {"_id": "entertainment", "categoryDisplayName": "Entertainment"},
  {"_id": "sports", "categoryDisplayName": "Sports"}
]
```

Platform Data

These documents **MUST** be manually populated into the database and are also included in the `core/mock_data` folder. Create a new collection within the `tlbc` database called `platforms` and insert the following document.

`platforms` collection:

```
[
  {"_id": "twitter", "platformDisplayName": "Twitter", "platformColour": "#1FBBEC"},
  {"_id": "clubhouse", "platformDisplayName": "Clubhouse", "platformColour": "#ECE41F"},
  {"_id": "linkedin", "platformDisplayName": "LinkedIn", "platformColour": "#0077B7"}
]
```