Welcome to your New Top Speed App!

We've created a database, website, app and api to be used for racers and organizers to communicate over the course of a tournament. This document will be broken into several sections covering the setup, maintenance and functionality of each component.

Here is a link to our trello board where we tracked tasks and have some tasks already planned out for future development:
Trello Board

Database

Database is a mySQL database that can be built and deleted through SQL query files

ER model and Fields:
https://viewer.diagrams.net/?tags=%7B%7D&highlight=0000ff&edit=_blank&layers=1&nav=1&title=AutoRacing_ER_MODEL#Uhttps%3A%2F%2Fdrive.google.com%2Fuc%3Fid%3D1BYQ4XT-rx1RO3hdE-d3WxXHeMHL_hfGl%26export%3Ddownload

Server

The Api, Database, and Website were all running on a low resource virtual server (Digital Ocean) at a subdomain owned by one of our group members. When setting up the project make sure to update anywhere that references the domain swe.cooperstandard.org or it will fail to work. The server os was Ubuntu version 20 long term support and we used apt to install all software packages. We configured the server to reject all unencrypted traffic, and used certbot to generate ssl certificates. The Website was hosted on the standard https port (443) and the api was on port 8080. All other ports besides 22 (for ssh) were blocked for incoming traffic through the default Ubuntu firewall. Because sensitive user data may be sent back and forth from the server we required encryption on all connections.

API

Running Locally

To make sure you have all dependencies installed, navigate to the api folder in a terminal and execute:

```
pip3 install -r requirements.txt
```

In order to connect to the remote database you will need to setup a SSH tunnel. To do so, open up terminal and enter the following command, replacing <your_ssh_username> with the username you use to SSH into the server and <server_address> with the address of the server:

```
ssh -f <your_ssh_username>@<server_address> -L 3307:127.0.0.1:3306 -N
```

Open the api.py file and replace the credentials and database name on line 24 with the correct values for your database. Then start the api with:

```
./api.py
```

The api will listen on 127.0.0.1 on port 5000, test by sending requests with curl or use the api-tests.py script in the test directory with:

```
./api-tests.py
```

After you are finished testing you will need to kill the SSH tunnel to the server. Enter ps aux | grep ssh | grep 3307 in terminal, then kill the process number shown with kill <pid>. Example shown below:

```
❯ ps aux | grep ssh | grep 3307
john  85601   0.0  0.0 408656656   1872   ??  Ss   11:21PM   0:00.01
ssh -f john@143.198.139.47 -L 3307:127.0.0.1:3306 -N
❯ kill 85601
```

Running on Server

Clone the api to your server. Navigate to the api subfolder and use the following command to install all dependencies:

```
pip3 install -r requirements.txt
```

Testing

Set the environment variable DEBUG to False and DB_URI to the URI for your database using the following commands:

```
export DEBUG=False
export DB_URI="your_database_URI_here"
```

Enter the path to your SSL certificate and key files on lines 98 and 99 of the api.py file. Replace the path on line 100 of the api.py file with your desired log file location (default is /var/log/api.log). Finally, start the api by navigating to the api folder and running:

```
./api.py
```

The api will listen on all interfaces on port 8080.

Production

The development server included with Flask is not suitable for use in production. Deploy using a production web server gateway interface as described in the official Flask documentation.

Endpoints

Examples for usage of all api endpoints can be found in the api-tests.py file in the test folder. Descriptions of all endpoints and their supported request types are listed below:

/register

- **POST**: creates a new user. Must include email, password, name, and accountType (either racer or raceOrganizer) in the request body. Returns the new user's racerID/orgName depending on account type.

/login

- **POST**: creates a new user. Must include email and password in the request body. Returns the user's racerID/orgName depending on account type and a JWT access token. This token **MUST** be included in the headers of all future requests!

/message

- **GET**: requires JWT header, returns all messages for the current user.
- **POST**: creates a new message from the passed in JSON value. Fields are title, body, and either racerID, heatID, or tourneyID.

/race_organizer/<organizer_name>

- **GET**: returns info for specified organizer.
- **PUT**: creates a new race organizer with specified name and JSON values (use /register for this).
- **PATCH**: updates specified race organizer with the fields passed in JSON. Only fields included in the request will be changed.

/racer/<racer_id>

- **GET**: returns info for the specified racer.
- **PUT**: creates a new racer with the specified id and JSON values (use /register for this).

- **PATCH**: updates the specified racer with the fields passed in JSON. Only fields included in the request will be changed.

/racer/<racer_id>/bikes

- **GET**: returns a list of bike JSON objects that belong to the specified racer.

/racer/<racer_id>/bike_racers

- **GET**: returns a list of bike_racer JSON objects associated with the specified racer.

/racer/<racer_id>/races?filter=<past/next/future>

- **GET**: returns all races this racer is involved in, subject to the provided filter. Filter options are past, next, and future. If no filter is provided all races, past and future, are returned.

/bike

- **POST**: creates a new bike with the values passed in JSON. returns JSON representation of the created bike.

/bike/<bike_id>

- **GET**: returns info for the specified bike.
- **PATCH**: updates the specified bike with the fields passed in JSON. Only fields included in the request will be changed.

/bike_racer/

- **POST**: creates a new bike racer with the values passed in JSON. returns JSON representation of the created bike racer.

/bike_racer/<bike_racer_id>

- **GET**: returns info for the specified bike racer.
- **PATCH**: updates the specified bike racer with the fields passed in JSON. Only fields included in the request will be changed.

/moto/

- **POST**: creates a new moto with the values passed in JSON. returns JSON representation of the created moto.

/moto/<moto_id>

- **GET**: returns info for the specified moto.

- **PATCH**: updates the specified moto with the fields passed in JSON. Only fields included in the request will be changed.

/heat

- **POST**: creates a new heat with the values passed in JSON. Returns JSON representation of the created heat.

/heat/<heat_id>

- **GET**: returns info for specified heat.
- **PATCH**: updates specified heat with the fields passed in JSON. Only fields included in the request will be changed.

/tourney

- **POST**: creates a new tourney with the values passed in JSON. returns JSON representation of the created tourney. If a list of classes is included, then qualifying heats for all classes will be created and also returned.

/tourney/<tourney_id>

- **GET**: returns info for specified tourney.
- **PATCH**: updates specified tourney with the fields passed in JSON. Only fields included in the request will be changed.

/tourney/<tourney_id>/classes

- **GET**: returns a list of all classes in the tourney.

/tourney/<tourney_id>/racers

- **GET**: returns a JSON structure where keys are the classes in the tourney and the values are a list containing JSON representations of all bike racers in the class.

/tourneys?filter=<current/future>

- **GET**: returns a list of tourneys. Setting filter to current will return all tourneys that have started, but have not finished. Setting filter to future will return all tourneys that have not started.

/join

- **POST**: requires a tourneyID, racerID, bikeID, and class JSON value. Creates a new bike racer using the specified racer and bike then creates a moto for that bike racer in the qualifying heat for the specified class in the specified tourney. Returns JSON representation of the new moto.

/season

- **POST**: creates a new season with the values passed in JSON. returns JSON representation of the created season.

/season/<season_id>

- **GET**: returns info for specified season.
- **PATCH**: updates specified season with the fields passed in JSON. Only fields included in the request will be changed.

Website

The Website is written in standard HTML and Javascript. We were hosting it statically on a barebones apache instance. It may not be an issue for setting up but the important configurations are:
Timeout 300
Keepalive On
MaxKeepAliveRequests 100
KeepAliveTimeout 20 #this is because of hardware limitations, lower is better
HostnameLookups Off

CSV upload file

The CSV File covers creating a tourney and patching a moto. Here is a picture of it.

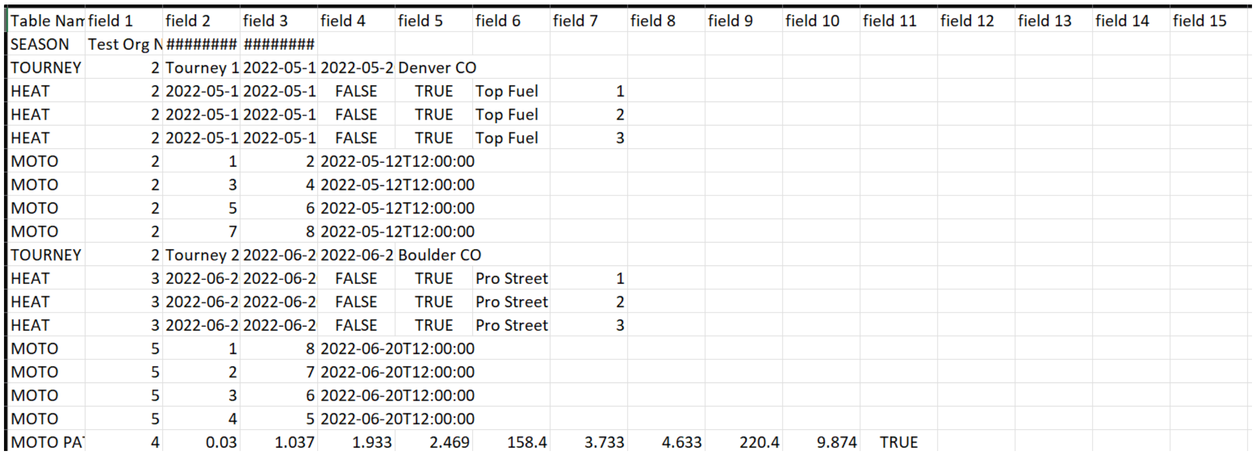


Table Nam	field 1	field 2	field 3	field 4	field 5	field 6	field 7	field 8	field 9	field 10	field 11	field 12	field 13	field 14	field 15
SEASON	Test Org N	########	########												
TOURNEY	2	Tourney 1	2022-05-1	2022-05-2	Denver CO										
HEAT	2	2022-05-1	2022-05-1	FALSE	TRUE	Top Fuel	1								
HEAT	2	2022-05-1	2022-05-1	FALSE	TRUE	Top Fuel	2								
HEAT	2	2022-05-1	2022-05-1	FALSE	TRUE	Top Fuel	3								
MOTO	2	1	2	2022-05-12T12:00:00											
MOTO	2	3	4	2022-05-12T12:00:00											
MOTO	2	5	6	2022-05-12T12:00:00											
MOTO	2	7	8	2022-05-12T12:00:00											
TOURNEY	2	Tourney 2	2022-06-2	2022-06-2	Boulder CO										
HEAT	3	2022-06-2	2022-06-2	FALSE	TRUE	Pro Street	1								
HEAT	3	2022-06-2	2022-06-2	FALSE	TRUE	Pro Street	2								
HEAT	3	2022-06-2	2022-06-2	FALSE	TRUE	Pro Street	3								
MOTO	5	1	8	2022-06-20T12:00:00											
MOTO	5	2	7	2022-06-20T12:00:00											
MOTO	5	3	6	2022-06-20T12:00:00											
MOTO	5	4	5	2022-06-20T12:00:00											
MOTO PA	4	0.03	1.037	1.933	2.469	158.4	3.733	4.633	220.4	9.874	TRUE				

Inputting Data into CSV

Season

-**Table Name**: SEASON

-**field 1**: Organization Name

-**field 2**: Start Date

-**field 3**: End Date

Tourney

-**Table Name**: TOURNEY

-**field 1**: Season ID

-**field 2**: Tourney Name

-**field 3**: Start Date (yyyy-mm-ddThh:mm:ss)

-**field 4**: End Date (yyyy-mm-ddThh:mm:ss)

-**field 5**: Location

Heat

-**Table Name**: HEAT

-**field 1**: Tourney ID

-**field 2**: Start Time (yyyy-mm-ddThh:mm:ss)

-**field 3**: End Time (yyyy-mm-ddThh:mm:ss)

-**field 4**: Is Delayed (True/False)

-**field 5**: Is Eliminated (True/False)

-**field 6**: Class Name

-**field 7**: Round Number

Moto

-**Table Name**: MOTO

-**field 1**: Heat ID

-**field 2**: Racer ID

-**field 3**: Opponent ID

-**field 4**: Start Time (yyyy-mm-ddThh:mm:ss)

Patch Moto

-**Table Name**: PATCH MOTO

-**field 1**: Dial Back

-**field 2**: Reaction Time

-**field 3**: 60 Time

-**field 4**: 330 Time

-**field 5**: 1/8 Time

-**field 6**: 1/8 MPH

-**field 7**: 1000 Time

-**field 8**: 1/4 Time

-**field 9**: 1/4 MPH

-**field 10**: Final Time

-**field 11**: Winner (True/False)

App

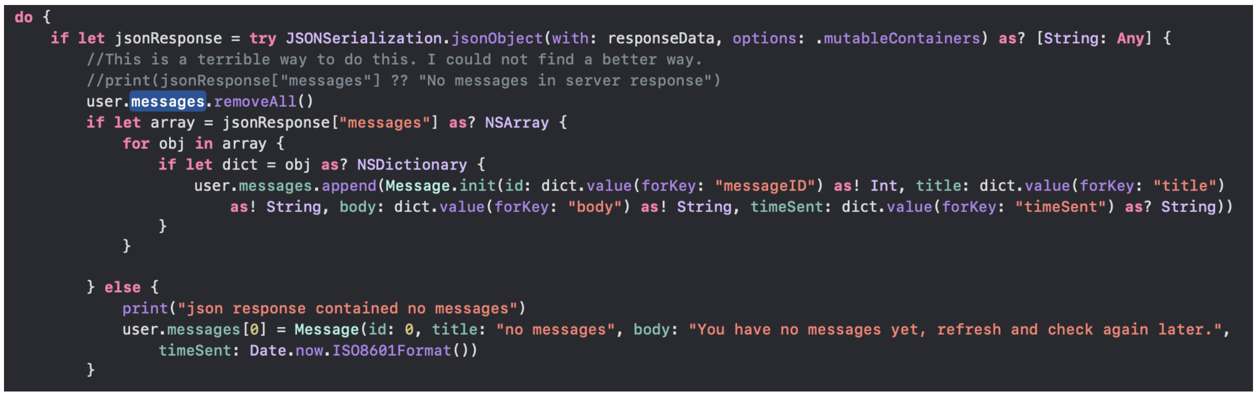
Download XCode on mac and locate the Top Speed xcode project from the zip file.

Quick IOS setup guide to run app on phone

Make sure to click on the top file that shows the apps build settings and make sure your user profile is for your computer.

We built the app using the Swift programming language, SwiftUI graphics framework, and other swift Standard Packages (like URLSession and Codable). Our minimum deployment version for IOS was 15.3 and we developed it using xcode version 13.4.
[This is Apple's documentation for SwiftUI]

The first View the user is presented with is the SignIn view. Once the user has provided valid credentials a series of api calls are completed to fill in the data for the User object. At the same time the user is transitioned to the MainMenu and from there they can access the different views to view data or submit new data. The api calls are run asynchronously and not all views have protection in place for missing/bad data but all requests log to the console when they succeed or if something unexpected happens. Besides the login view, all main views are assumed to be in a NavigationView which is rooted at MainMenu. Our Api Calls require a large amount of boiler plate but the actual action that takes place can be found in blocks like this (picture from the getMessages function in Handler.swift):



```swift
do {
    if let jsonResponse = try JSONSerialization.jsonObject(with: responseData, options: .mutableContainers) as? [String: Any] {
        //This is a terrible way to do this. I could not find a better way.
        //print(jsonResponse["messages"] ?? "No messages in server response")
        user.messages.removeAll()
        if let array = jsonResponse["messages"] as? NSArray {
            for obj in array {
                if let dict = obj as? NSDictionary {
                    user.messages.append(Message.init(id: dict.value(forKey: "messageID") as! Int, title: dict.value(forKey: "title")
                        as! String, body: dict.value(forKey: "body") as! String, timeSent: dict.value(forKey: "timeSent") as? String))
                }
            }

        } else {
            print("json response contained no messages")
            user.messages[0] = Message(id: 0, title: "no messages", body: "You have no messages yet, refresh and check again later.",
                timeSent: Date.now.ISO8601Format())
        }
```

All the other code in this function, which is not shown in the picture, sets up the http request and checks the response before trying to decode it. Hopefully Apple creates a better way to handle REST API’s sometime soon, but as of now this was the most reliable way we could find to get the functionality we needed.