Tiger Leagues Documentation

Release 1.0

Chege, Ivy, Rui, Obinna

Jan 15, 2019

CONTENTS

The source code for this project lives in https://github.com/dchege711/cos333_tiger_leagues

CHAPTER

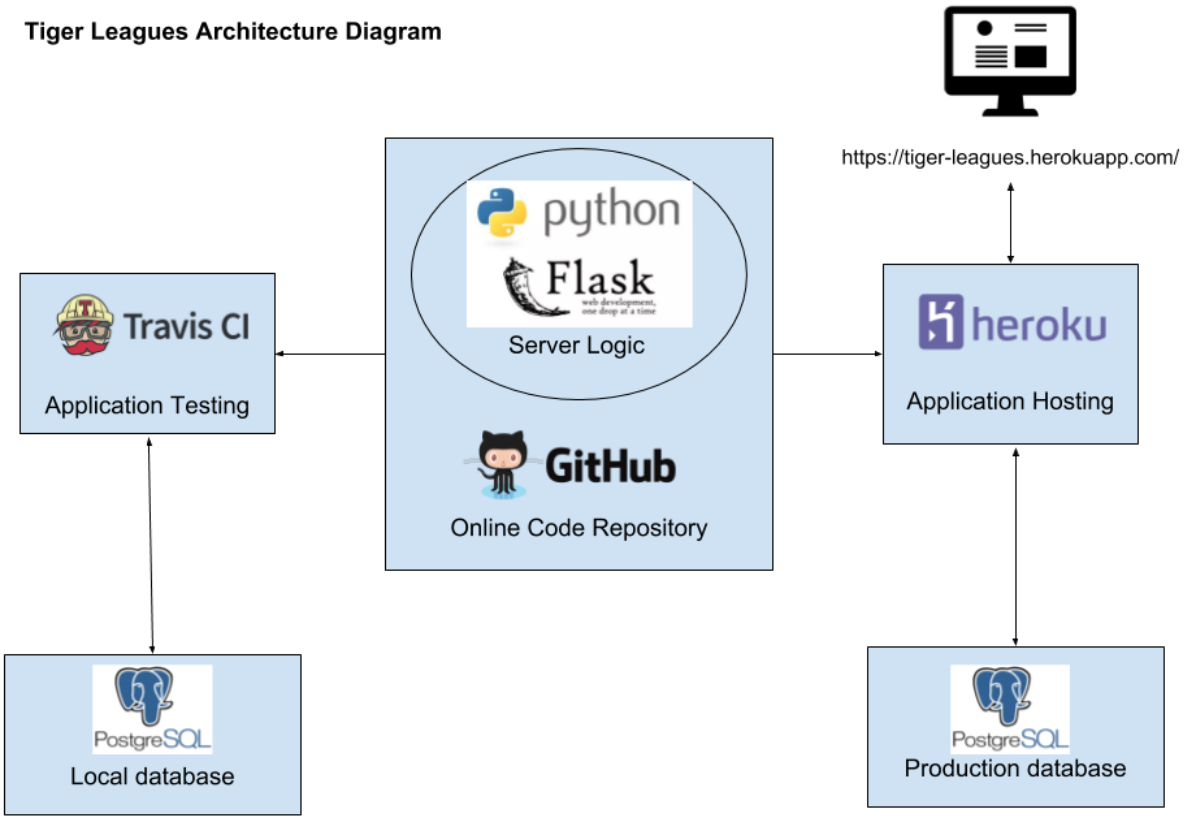
ONE

APP OVERVIEW

Tiger Leagues is a web application hosted at https://tiger-leagues.herokuapp.com/. It is used by Princeton students that wish to run a league amongst themselves. The application was created in lieu of managing leagues over Google Docs. Features include:

- Creating leagues and registering members
- Scheduling league games
- Keeping track of scores and leaderboards

1.1 Components



1.2 Getting Started

Clone the repository with:

```
$ git clone https://github.com/dchege711/cos333_tiger_leagues.git
$ cd cos333_tiger_leagues
```

Install the python packages (preferably in a new virtual environment):

```
$ pip install requirements.txt
```

We modelled the application after this flask tutorial. Installing Flask usually installs `ItsDangerous v1.0.0` as a prerequisite. However, Heroku cannot install v1.0.0. For this reason, `./requirements.txt` was manually updated to have `ItsDangerous==0.24`.

We're using a locally hosted database for development purposes. We found this tutorial useful when setting up PostgreSQL.

Set the values of the environment variables defined in config.py.

Run the application server:

```
$ ./run_flask_server
```

1.3 Architecture

Tiger Leagues uses the Model-View-Controller architectural pattern.

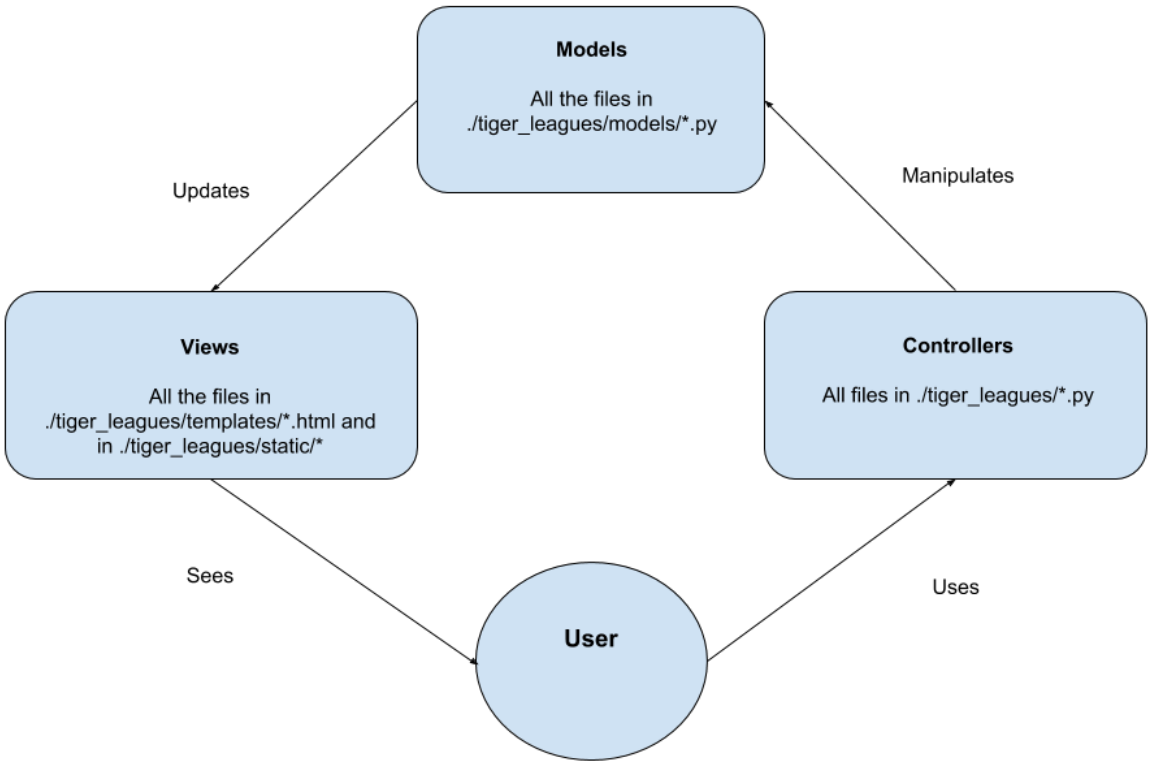


Diagram adapted from Wikipedia

By RegisFrey - Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=10298177

For more detailed documentation and design decisions made at each level see:

- Models
- Views
- Controllers

1.4 Testing

The tests are defined in the `tests` folder at the root of the project. We're using pytest for our testing. We have included a helper bash script (`run_tests`) to run the tests and provide coverage analysis.

We have also set up Travis CI to test entire builds. The Travis CI configuration is defined in the `.travis.yml` file at the root. We have set up `origin/master` as a protected branch, so make sure Travis CI greenlights any pull requests.

1.5 Generating the Documentation

The documentation for this project is modelled after Matplotlib's Tutorial which generates documentation with Sphinx.

`conf.py` sets up the settings used by Sphinx. The docs are built from the .md and .rst files found within this repository.

You generally need to run `$ make html` to build the documentation. The docs are built inside the `docs` folder, so that GitHub Pages can access them and serve them at https://dchege711.github.io/cos333_tiger_leagues

1.6 Additional Feature Requests

The following are remarks made by users. In case you're looking for features to implement, this is it Chief!

- You should add a feature to upload and share pictures and videos of the match to brag.
- We can have a puskas award (most beautiful goal) for the league at the end of the tourney
- In-depth stats, highlights, something to track playoff probability/scenarios toward the end would be pretty cool.
- Also a schedule for all the games.
- We can watch other ppl's matches right? Would be cool to have an audience

CHAPTER

TWO

MODELS

As described on Wikipedia, the model is the application's dynamic data structure, independent of the user interface. It directly manages the data, logic and rules of the application. We deviated a bit from the standard MVC architecture by validating our inputs at this level.

Here is a quick breakdown of where the higher-level application logic is handled:

Model	Application Logic
tiger_leagues.models.league_model	• Creating a new league • Recording requests to join a league • Updating league standings • Fetching league standings • Fetching league matches • Fetching player stats • Processing score reports submitted by players • Processing player requests to leave a league
tiger_leagues.models.admin_model	• Adding/removing players from a league • Allocating league divisions • Processing score reports submitted by admins • Deleting a league
tiger_leagues.models.user_model	• Fetch existing user profile • Update a user's profile • Post notifications to a user • Read user's notifications
tiger_leagues.models.exception	• Raise exceptions caused by errors encountered when accomplishing any of the above logic

2.1 Design Decisions

2.1.1 Application Context

Tiger Leagues can be ran in 3 different contexts. We used environment variables and *tiger_leagues.models.config* to switch between the different contexts:

- Development

Occurs when Tiger Leagues is being ran locally. We found it convenient to use a locally hosted database so that the dev can modify its content as they see fit.

- Travis CI

Our tests write and read data from the database. We found it convenient to use a PostgreSQL database that is provisioned by Travis CI. The database is set up by the `.travis.yml` file at the root of the repository.

- Heroku (Production)

If Tiger Leagues is running on Heroku, we use the database provided by Heroku.

2.1.2 League Rankings

Initially, we'd compute the rankings of players from the matches table. This was motivated by reducing redundancy in the database. However, we hypothesized that users will view the rankings way more frequently than update scores.

We therefore decided to add another table, `league_standings` that contains the most recent rankings that incorporate all the approved score reports. Although this creates some redundancy (we could determine the rankings from the score reports), it allows us to reduce repeated computation.

2.1.3 Keeping the User Updated

Since the users are not isolated, it's important to keep a user updated of any developments that involve them. For instance, a user might get their join request approved/denied by an admin. Or the score for a given match might have been reported by the other player and the admin approved of it.

We therefore developed a rudimentary notification system in which we post relevant updates to a user's mailbox. The system does not allow for responses. We leave that for future implementations of Tiger Leagues.

2.2 Models Documentation

2.2.1 tiger_leagues.models.db_model

db.py

A wrapper around the database used by the 'Tiger Leagues' app

class tiger_leagues.models.db_model.**Database**(*connection_uri=None*)

A wrapper around the database used by the 'Tiger Leagues' app.

Parameters **connection_uri** – str

Optional connection string for the database. If `None`, this defaults to the connection string set in `config.DATABASE_URL`.

disconnect()

Close the connection to the database. Should be called before exiting the script.

execute(*statement*, *values=None*, *dynamic_table_or_column_names=None*, *cursor_factory=<class 'psycopg2.extras.DictCursor'>*)

Parameters **statement** – str

The SQL query to run.

Parameters **values** – list

Values that the query's placeholders should be replaced with

Parameters **dynamic_table_or_column_names** – list

Names of tables/columns that should be substituted into the SQL statement

Parameters **cursor_factory** – psycopg2.extensions.cursor

The type of object that should be generated by calls to the `cursor()` method.

Returns `cursor`

The cursor after after executing the SQL query

Raise `psycopg2.errors`

If the SQL transaction fails, the transaction is rolled back. The most recently executed query is printed to `sys.stderr`. The error is then raised.

execute_many(*sql_query*, *values*, *dynamic_table_or_column_names=None*, *cursor_factory=<class 'psycopg2.extras.DictCursor'>*)

Execute many related SQL queries, e.g. update several rows of a table.

Parameters **sql_query** – str

The SQL query to run. It must contain a single *%s* placeholder

Parameters **values** – iterable

Each item should be a value that can be substituted when composing a SQL query

Parameters **dynamic_table_or_column_names** – list

Names of tables/columns that should be substituted into the SQL statement

Parameters **cursor_factory** – psycopg2.extensions.cursor

The type of object that should be generated by calls to the `cursor()` method.

Returns `cursor`

The cursor after after executing the SQL query

Raise `psycopg2.errors`

If the SQL transaction fails, the transaction is rolled back. The most recently executed query is printed to `sys.stderr`. The error is then raised.

iterator(*cursor*)

An alternative to having the `x = cursor.fetchone()` ` ... ` `while x is not None` dance when iterating through cursor's results.

Parameters **cursor** – psycopg2.cursor

The cursor after after executing the SQL query

Yield `Row`

A row fetched from the cursor.

Warn `DepracationWarning`

Unlike sqlite3, psycopg2 provides an iterable cursor, so this method is unnecessary baggage.

launch()

Initialize the tables if they do not exist yet.

2.2.2 tiger_leagues.models.config

config.py

The central source for variables that span the entire application. As a rule of thumb, if you find yourself using *os.environ*, you should probably include the variable here instead.

Expected environment variables: `TIGER_LEAGUES_ENVIRONMENT`, `TIGER_LEAGUES_POSTGRESQL_DBNAME`, `TIGER_LEAGUES_POSTGRESQL_USERNAME`, `TIGER_LEAGUES_POSTGRESQL_PASSWORD`

2.2.3 tiger_leagues.models.league_model

league.py

Exposes a blueprint that handles requests made to */league/** endpoint

`tiger_leagues.models.league_model.`**`create_league`**(*league_info*, *creator_user_id*)

Parameters **`league_info`** – dict

Expected keys: `league_name`, `description`, `points_per_win`, `points_per_draw`, `points_per_loss`, `registration_deadline`, `additional_questions`.

Parameters **`creator_user_profile`** – int

The ID of the user creating this league

Returns `dict`

`success` is set to `True` only if the league was created. If `success` is `False`, the `message` field will contain a decriptive error message. Otherwise, the `message` field will be an `int` representing the league ID

`tiger_leagues.models.league_model.`**`get_league_info`**(*league_id*)

Parameters **`league_id`** – int

The ID of this league

Returns `dict`

Keys: `league_id`, `league_id`, `league_name`, `description`, `points_per_win`, `points_per_draw`, `points_per_loss`, `league_status`, `additional_questions`, `registration_deadline`, `num_games_per_period`, `length_period_in_days`, `max_num_players`

Raise `TigerLeaguesException`

If the league_id is not found in the database.

`tiger_leagues.models.league_model.`**`get_league_info_if_joinable`**(*league_id*)

Parameters **`league_id`** – int

The ID of the league

Returns `dict`

If `succcess` is `False`, `message` will contain a descriptive error message. If `success` is `True`, `message` will contain the league information needed to join the league.

`tiger_leagues.models.league_model.`**`get_league_standings`**(*league_id*)

Parameters **`league_id`** – int

The ID of the league

Returns dict[list[dict]]

The sorted league standings. The outermost dict is keyed by the division ID. The list[dict] is sorted by points and then by goal difference. This innermost is keyed by wins, losses, draws, games_played, goals_for, goals_allowed, goal_diff, points, rank, rank_delta

If the league doesn't exist, the dict will be empty.

tiger_leagues.models.league_model.**get_leagues_not_yet_joined**(*user_profile*)

Parameters **user_profile** – dict

Expected keys: associated_leagues

Returns List[DictRow]

Each item is keyed by league_id, league_name, registration_deadline and description

tiger_leagues.models.league_model.**get_matches_in_current_window**(*league_id*, *num_periods_before=3*, *num_periods_after=3*, *user_id=None*)

Parameters **league_id** – int

The ID of the league

Parameters **num_periods_before** – int (or infinity)

The fetched matches will have deadlines that are at least on/later than today - num_days_between_games * num_periods_before If the value is inf, then all matches that have deadlines earlier than today will be included in the results.

Parameters **num_periods_after** – int (or infinity)

The fetched matches will have deadlines that are at least on/earlier than today + num_days_between_games * num_periods_before If the value is inf, then all matches that have deadlines later than today will be included in the results.

Parameters **user_id** – int

If set, only return matches that are associated with this user ID

Returns List[DictRow]

A list of all the matches within the current time window. These are the matches that are about to be played or have been played. Keys include: `match_id` `user_1_id`, `user_2_id`, `league_id`, `division_id`,`score_user_1`,`score_user_2`,`status`,`deadline`

tiger_leagues.models.league_model.**get_player_comparison**(*league_id*, *user_1_id*, *user_2_id*)

Parameters **league_id** – int

The ID of the associated league

Parameters **user_1_id** – int

The ID of the first user. By convention, user 1 is the initiator of the request.

Parameters **user_2_id** – int

The ID of the second user

Returns dict

Keyed by `success` and `message`. If `success` is `True`, `message` will be a dict keyed by `rank`, `points`, `mutual_opponents`, `head_to_head`, `player_form`

`tiger_leagues.models.league_model.`**`get_players_current_matches`**(*user_id*, *league_id*, *num_periods_before=4*, *num_periods_after=4*)

Unlike `get_matches_in_current_window()`, this method resolves the opponent names as well as adding convenient fields such as `my_score`, `opponent_score`, `opponent_id`, `opponent_name`.

Parameters **`user_id`** – int

The ID of the player (equivalent to user)

Parameters **`league_id`** – int

The ID of the league

Parameters **`num_periods_before`** – int (or infinity)

The fetched matches will have deadlines that are at least on/later than `today - num_days_between_games * num_periods_before` If the value is `inf`, then all matches that have deadlines earlier than today will be included in the results.

Parameters **`num_periods_after`** – int (or infinity)

The fetched matches will have deadlines that are at least on/earlier than `today + num_days_between_games * num_periods_before` If the value is `inf`, then all matches that have deadlines later than today will be included in the results.

Returns `List[dict]`

The player's current matches ordered by the deadline. Each dict is keyed by `match_id`, `league_id`, `division_id`, `status`, `deadline`, `my_score`, `opponent_id`, `opponent_score`, `opponent_name`

`tiger_leagues.models.league_model.`**`get_players_league_stats`**(*league_id*, *user_id*, *matches=None*, *k=5*)

Parameters **`league_id`** – int

The ID of the associated league

Parameters **`user_id`** – int

The ID of the user

Parameters **`matches`** – `list[dict]`

A list of matches to calculate the player's form from. The matches should be ordered such that the most recent matches appear last in the list. If `NoneType`, this method queries the database for such matches.

Parameters **`k`** – int

The max number of matches to calculate a player's form from.

Returns `dict`

Keyed by `success` and `message`. If `success` is `True`, `message` will be a dict keyed by `rank`, `points`, `player_form` If `success` is `False`, `message` will contain a descriptive error message.

`tiger_leagues.models.league_model.`**`get_previous_responses`**(*league_id*, *user_profile*)

Parameters **`league_id`** – int

The ID of the league

Parameters **user_profile** – dict

Expected keys: `associated_leagues, user_id`

Returns `NoneType`

If the user has not tried to join this league before

Returns `DictRow`

The responses that the user previously entered while trying to join this league.

`tiger_leagues.models.league_model.`**process_join_league_request**(*league_id*, *user_profile*, *submitted_data*)

Parameters **league_id** – int

The ID of this league

Parameters **user_profile** – dict

Expected keys: `user_id, league_ids`

Returns `dict`

If `succcess` is `False`, `message` will contain a descriptive error message. If `success` is `True`, `message` will contain a dict containing the updated user profile.

`tiger_leagues.models.league_model.`**process_leave_league_request**(*league_id*, *user_profile*)

Parameters **league_id** – int

The ID of the league

Parameters **user_profile** – dict

Expected keys: `user_id, league_ids`

Returns `bool`:

`True` if the user was successfully removed from the league, `False` otherwise

`tiger_leagues.models.league_model.`**process_player_score_report**(*user_id*, *score_details*)

Parameters **user_id** – int

The ID of the user submitting the score report

Parameters **score_details** – dict

Expected keys: `my_score`, `opponent_score`, `match_id`.

Returns `` `dict` ``

Keys: `` `success` ``, `` `message` ``. If `` `success` `` is `` `True` ``, `` `message` `` contains the status of the match after the score has been processed. Otherwise, `` `message` `` contains an explanation of what went wrong.

`tiger_leagues.models.league_model.`**process_update_league_responses**(*league_id*, *user_profile*, *submitted_data*)

Parameters **league_id** – int

The ID of this league

Parameters **user_profile** – dict

Expected keys: `user_id, league_ids`

Returns `dict`

If `succcess` is `False`, `message` will contain a descriptive error message. If `success` is `True`, `message` will contain a dict containing the updated user profile.

`tiger_leagues.models.league_model.`**`update_league_standings`**(*league_id*, *division_id*)

Compute the new league standings and persist them into the database.

Parameters **`league_id`** – int

The ID of the league

Parameters **`division_id`** – int

The ID of the division within the league of interest

Returns `NoneType`

This method affects the state of the database. It doesn't return anything. To fetch the standings, call `get_league_standings()` instead.

2.2.4 tiger_leagues.models.admin_model

admin_model.py

Exposes functions that are used by the controller for the */admin/** endpoint

`tiger_leagues.models.admin_model.`**`allocate_league_divisions`**(*league_id*, *desired_allocation_config*)

Parameters **`league_id`** – int

The ID of the league

Parameters **`desired_allocation_config`** – dict

Options to use when allocating the divisions. Keys may include `num_games_per_period`, `length_period_in_days`, `completion_deadline`

Returns `dict`

If `success` is `False`, `message` contains a string describing what went wrong. Otherwise, `message` is a dict keyed by `divisions` and `end_date`

`tiger_leagues.models.admin_model.`**`approve_match`**(*score_info*, *admin_user_id*)

Parameters **`score_info`** – dict

Expected keys: `score_user_1`, `score_user_2`, `match_id`

Parameters **`admin_user_id`** – int

The ID of the admin approving the match's results

Returns `dict`

Keys: `success`, `message`. If `success` is `True`, `message` has the updated status of the match. Otherwise, `message` explains why the update failed.

`tiger_leagues.models.admin_model.`**`delete_league`**(*league_id*)

Parameters **`league_id`** – int

The ID of the league

Returns dict

Keys: success, message. If success is True, message has a confirmation message. Otherwise, message explains why the deletion failed.

tiger_leagues.models.admin_model.**fixture_generator**(*user_ids*)

Parameters **user_ids** – List[int]

A list of the IDs of users who are supposed to play each other.

Returns List[List[List]]

The innermost list has 2 elements (the IDs of the players involved in a game). The middle list has a collection of all the games being played at a particular timeslot. The outermost list encompasses all the games that will be played between all the users.

tiger_leagues.models.admin_model.**generate_league_fixtures**(*league_id*, *div_allocations*, *start_date=None*)

Parameters **league_id** – int

The ID of the league

Parameters **div_allocations** – dict

The keys are the division IDs. Each value is a dict keyed by name and user_id representing a player associated with the league.

Parameters **start_date** – date

The earliest games' time window will start from this date. Defaults to tomorrow

Returns dict

If success is False, message will have a description of why the call failed. Otherwise, message will contain a string confirming that the fixtures were generated.

tiger_leagues.models.admin_model.**get_current_matches**(*league_id*)

Parameters **league_id** – int

The ID of the league

Returns List[DictRow]

A list of all matches in the current time block. Keys include match_id, league_id, user_1_id, user_2_id, division_id, score_user_1, score_user_2, status, user_1_name, user_2_name

tiger_leagues.models.admin_model.**get_join_league_requests**(*league_id*)

Parameters **league_id** – int

The ID of the league

Returns List[DictCursor]

A row for each user who submitted a request to join this league.

tiger_leagues.models.admin_model.**get_registration_stats**(*league_id*)

Parameters **league_id** – int

The ID of the league

Returns dict

The keys are various join statuses and the values are their frequency.

tiger_leagues.models.admin_model.**update_join_league_requests**(*league_id*, *league_statuses*)

Parameters **league_id** – int

The ID of the league

Parameters **league_statuses** – dict

The keys are user_ids and the values are any of the supported status strings

Returns dict

If success is set, message will contain a user_id->status matching. Otherwise, message will contain an error description.

2.2.5 tiger_leagues.models.user_model

user_model.py

Exposes functions that are used by the controller for the */user/** endpoint

tiger_leagues.models.user_model.**get_user**(*net_id*, *user_id=None*)

Parameters **net_id** – str

The Princeton Net ID of the user

Parameters **user_id** – int

The ID of the user as assigned in Tiger Leagues

Returns dict

A representation of the user as stored in the database. Keys include: user_id, name, net_id, email, phone_num, room, league_ids, associated_leagues, unread_notifications

Returns NoneType

If there is no user in the database with the provided net id

tiger_leagues.models.user_model.**read_notifications**(*user_id*, *notification_status=None*)

Parameters **user_id** – int

The ID of the associated user

Parameters **notification_status** – str

The status of the notifications that are to be read. If None, this defaults to notifications that have not been archived.

Returns cursor

An iterable cursor where each item keyed by notification_id, notification_status, notification_text, created_at, league_name.

tiger_leagues.models.user_model.**send_notification**(*user_id*, *notification*)

Send a notification to this user

Parameters **user_id** – int

The ID of the associated user

Parameters **notification** – dict

Expected keys include: `league_id, notification_text`

Returns `int`

The notification ID if the notification is successfully delivered to the user's mailbox.

Returns `NoneType`

If the method failed

`tiger_leagues.models.user_model.`**update_notification_status**(*user_id*, *notification_obj*)

Parameters **user_id** – int

The ID of the user making this request

Parameters **notification_obj** – dict

Expected keys: `notification_id`, `notification_status`

Returns `dict`

Keyed by `success` and `message`. If `success` is `False`, `message` contains a description of why the request failed. If `success` is `True`, `message` contains the new status of the notification.

`tiger_leagues.models.user_model.`**update_user_profile**(*user_profile*, *net_id*, *submitted_data*)

Parameters **user_profile** – dict

A representation of the user, usually obtained from `get_user(net_id)`. If set to `None`, a new user will be created and added to the database.

Parameters **net_id** – str

The Princeton Net ID of the user

Parameters **submitted_data** – dict

Keys may include *name*, *email*, *phone_num*, *room*

Returns dict

The updated user profile

2.2.6 tiger_leagues.models.exception

exception.py

Allows for error pages/responses with custom exception messages.

exception `tiger_leagues.models.exception.`**TigerLeaguesException**(*message*, *status_code=400*, *jsonify=True*)

A special exception for errors that arise due to constraints that we set on the application, for instance, a user may not access the league panel for a league in which they lack an admin status, etc.

Parameters **message** – str

human readable string explaining the problem

Parameters **status_code** – int

To specify the error code in the response. Like 400, 404, 500, etc.

Parameters **jsonify** – bool

Set the `jsonify` attribute of the exception. The error handler can then check this value to decide how to convey the error to the user.

to_dict()

Returns `dict`

A dict representation of the exception

`tiger_leagues.models.exception.`**validate_values**(*data_obj*, *constraints*, *jsonify=True*)

Helper function for validating JSON input

Parameters **data_obj** – dict

A key-value pairing that needs to be validated

Parameters **constraints** – list[tuple]

Each tuple has 5 items. In order, they are: key (str), cast_function (function), l_limit (value), u_limit (value), error_msg (str)

Parameters **jsonify** – bool

If `True`, the raised `TigerLeaguesException` will have its jsonify attribute set.

Raises `TigerLeaguesException`

If any of the keys don't exist or any of the values fail to meet the constraint.

CHAPTER

THREE

VIEWS

As described on Wikipedia, a view is any output representation of information. In our case, our views were all HTML files. We also have supporting stylesheets, images and JavaScript in `./tiger_leagues/static/*`

3.1 Design Decisions

3.1.1 Uniform Design

Our application has several pages, so it's important to keep their design uniform. Using the Jinja templates supported natively by Flask, we inherited templates whenever we felt that a group of pages should share some design.

3.2 Views Documentation

3.2.1 base.html

Serves as the overall template for all other HTML files. The header and footer (and any other persistent content) should be added here.

3.2.2 error.html

Used to render a custom error page.

3.2.3 admin/*

The HTML files found here correspond to different pages that are relevant to admins, e.g.

- `admin_league_panel.html` shows different actions that an admin can take
- `manage_members.html` shows pages for managing league members.
- `start_league.html` allows the admin to allocate league divisions and generate fixtures.
- `admin_league_homepage.html` allows admins to approve pending scores.
- `delete_league.html` allows admins to delete the league.

3.2.4 auth/*

Contains HTML files related to the authentication process, e.g.

- `login.html` provides a link to Princeton's Central Authentication System.

Since we're using Princeton's CAS, other auth-related pages such as resetting a password, validating an email address, etc, are not necessary.

3.2.5 league/*

Contains HTML files related to a league from the viewpoint of a non-admin member.

- `browse.html` shows leagues that a user can request to join.
- `create_league.html` allows a user to create a new league.
- `join_league.html` allows a user to request to join an existing league.
- `update_responses.html` allows a user to update the responses that they had submitted to the league.
- `league_base.html` provides an inheritable template that has league header information at the top.
- `league_homepage.html` shows the current standings and upcoming matches of the logged in user.
- `member_stats/league_comparison_base.html` provides an inheritable template for displaying a player(s) stats within a league.
- `member_stats/league_side_by_side_stats.html` provides a side-by-side comparison of the logged in user and any other comparable player.
- `member_stats/league_single_player_stats.html` provides league stats for a single player (usually happens when user tries to view themselves, or a player who is not in the same division)

3.2.6 user/*

Contains HTML files related to a user's account.

- `user_profile.html` allows a user to view and/or update their site-wide profile.
- `user_notifications.html` allows a user to read the notifications that have been sent to their mailbox.

CHAPTER

FOUR

CONTROLLERS

As described on Wikipedia, the contoller receives the input, optionally validates it and then passes the input to the model.

Unlike the typical MVC model, we validated our most of our input in the models. We did this because our test suite focused on the models.

Here is a quick breakdown of the input handled by the controllers:

Controller	Input to Relay to the Model
tiger_leagues.auth (This controller uses *tiger_leagues.cas_client* to complete its tasks)	• Results of CAS authentication for login • Request to log out the user
tiger_leagues.league	• Creating a new league • Recording requests to join a league • Updating league standings • Fetching league standings • Fetching league matches • Fetching player stats • Processing score reports submitted by players • Processing player requests to leave a league
tiger_leagues.admin (This controller also checks that the logged in user has admin privileges)	• Adding/removing players from a league • Allocating league divisions • Processing score reports submitted by admins • Deleting a league
tiger_leagues.user	• Fetch existing user profile • Update a user's profile • Post notifications to a user • Read user's notifications
tiger_leagues.decorators	Used as middleware • Confirm that user is logged in • Refresh a user's notifications

4.1 Design Decisions

4.1.1 Use of Decorators

We extensively used decorators, as defined in *tiger_leagues.decorators*, to enforce access control, e.g. only logged in users can join a league, only admins can accept/reject league members, etc.

4.1.2 Graceful Error Handling

Since we defined a custom exception class, *tiger_leagues.models.exception*, we were able to set an error handler for any such exception. This allows us to gracefully show helpful error pages/ responses instead of the default ones.

4.2 Controllers Documentation

4.2.1 tiger_leagues.auth

auth.py

Handles authentication-related requests e.g. `login`, `logout`. Exposes a blueprint that handles requests made to the *auth* endpoint

`tiger_leagues.auth.`**`cas_login`**`()`

Log in users through CAS. At the end of the CAS-related stuff, the rest of the application expects to find a user object set in the session object.

Note that the contents of the session are public, but immutable. Please exclude values that you would not like the world to see. If sensitive data is needed, leave it to the caller to query the database themselves.

Returns `flask.Response(code=302)`

A redirect to the account creation page for new users or the homepage for any of the leagues that a returning user is associated with.

`tiger_leagues.auth.`**`cas_logout`**`()`

Log out the currently logged in user.

Returns `flask.Response(code=302)`

Redirect to the login page.

`tiger_leagues.auth.`**`index`**`()`

Returns `flask.Response(mimetype='text/HTML')`

Render the login page if the person isn't logged in, otherwise render a homepage for any of the leagues that they're involved in.

4.2.2 tiger_leagues.cas_client

cas_client.py

A convenient wrapper around the central authentication system.

@authors: Scott Karlin, Alex Halderman, Brian Kernighan, Bob Dondero

@modified: Ported to Python 3.7 & Flask by Chege Gitau

class tiger_leagues.cas_client.**CASClient**(*url='https://fed.princeton.edu/cas/'*)

A convenient wrapper around the central authentication system.

authenticate(*request*, *redirect*, *session*)

Authenticate the remote user.

Parameters **request** – flask.Request

A request that occurs as part of the CAS authentication process.

Parameters **redirect** – flask.redirect

A function that, if called, returns a 3xx response

Parameters **session** – flask.session

A session object whose values can be accessed by the rest of the application. If the authentication is successful, the username attribute will be set.

Returns str

If the user has been successfully authenticated, return their username

Returns flask.Response(code=302)

If the user has not been successfully authenticated, redirect them to the CAS server's login page.

strip_ticket(*request*)

Parameters **request** – flask.Request

A request that occurs as part of the CAS authentication process.

Returns str

The URL of the current request after stripping out the *ticket* parameter added by the CAS server.

validate(*ticket*, *request*)

Validate a login ticket by contacting the CAS server.

Parameters **request** – str

A ticket that can be validated by CAS. Once a user authenticates themselves with CAS, CAS makes a GET request to the application. This GET request contains a ticket as one of its parameters.

Parameters **request** – flask.Request

A request that occurs as part of the CAS authentication process.

Returns str

The user's username if valid

Returns NoneType

Returned if the user is invalid

4.2.3 tiger_leagues.admin

admin.py

Exposes a blueprint that handles requests made to */admin/** endpoint.

The blueprint is then registered in the __init__.py file and made available to the rest of the Flask application

tiger_leagues.admin.**admin_status_required**()

A decorator function that asserts that a user has admin privileges for the requested URL. This function is automatically called before any of the functions in the `admin` module are executed. See http://flask.pocoo.org/docs/1.0/api/#flask.Flask.before_request

Returns `flask.Response(code=302)`

A redirect to the login page if the user hasn't logged in yet.

Returns `flask.Response(code=302)`

A redirect to an exception page if the user doesn't have admin privileges in the league associated with this request.

Returns `None`

If the user has admin privileges for the current league, the request will then be passed on to the next function on the chain, typically the handler function for the request.

tiger_leagues.admin.**allocate_league_divisions**(*league_id*)

Parameters **league_id** – `int`

The ID of the league associated with this request

Returns `flask.Response(mimetype=application/json)`

A JSON object containing allocations of players in a league into divisions

tiger_leagues.admin.**approve_scores**(*league_id*)

Parameters **league_id** – `int`

The ID of the league associated with this request

Returns `flask.Response(mimetype=text/html)`

If responding to a GET request, render a HTML page that allows the admin to approve any reported scores.

Returns `flask.Response(mimetype=application/json)`

If responding to a POST request, approve the scores as reported in the body of the POST request. Return a JSON object that confirms that the scores updated on the server.

tiger_leagues.admin.**delete_league**(*league_id*)

Parameters **league_id** – `int`

The ID of the league associated with this request

Returns `flask.Response(mimetype=text/html)`

If responding to a GET request, render a HTML page that prompts the admin to delete the league, or abort the deletion

Returns `flask.Response(mimetype=application/json)`

If responding to a POST request, delete the league as specified in the POST request's body. Return a JSON object that confirms that the league was indeed deleted from the server.

tiger_leagues.admin.**league_has_started**()

A decorator function that asserts that a league has already started. Called before approve_scores and any other functions that should only take place with a started league.

Returns `flask.Response(code=302)`

A redirect to an exception page if the league has already started.

Returns None

If the league has not yet started, the request will then be passed on to the next function on the chain, typically the handler function for the request.

tiger_leagues.admin.**league_homepage**(*league_id*)

Parameters **league_id** – int

The ID of the league associated with this request

Returns flask.Response(mimetype='text/HTML')

Render a page with links to admin actions such as 'Approve Members'

tiger_leagues.admin.**league_not_started**()

A decorator function that asserts that a league has not yet started. This function is automatically called before any of the functions in the admin module are executed. See http://flask.pocoo.org/docs/1.0/api/#flask.Flask.before_request

Returns flask.Response(code=302)

A redirect to an exception page if the league has already started.

Returns None

If the league has not yet started, the request will then be passed on to the next function on the chain, typically the handler function for the request.

tiger_leagues.admin.**league_requests**(*league_id*)

Parameters **league_id** – int

The ID of the league associated with this request

Returns flask.Response(mimetype='text/HTML')

If responding to a GET request, render a template such that an admin can view the requests to join the league and can choose to accept or reject the join requests

Returns flask.Response(mimetype=application/json)

If responding to a POST request, update the join status of the users as instructed in the POST body. The JSON contains the keys message and success

tiger_leagues.admin.**manage_members**(*league_id*)

Parameters **league_id** – int

The ID of the league associated with this request

Returns flask.Response(mimetype='text/HTML')

If responding to a GET request, render a template such that an admin can view the requests to join the league and can choose to accept or reject the join requests

Returns flask.Response(mimetype=application/json)

If responding to a POST request, update the join status of the users as instructed in the POST body. The JSON contains the keys message and success

tiger_leagues.admin.**start_league**(*league_id*)

Parameters **league_id** – int

The ID of the league associated with this request

Returns flask.Response(mimetype='text/HTML')

If responding to a GET request, render a template for setting the league configuration, e.g. frequency of matches

Returns `flask.Response(mimetype=application/json)`

If responding to a POST request, generate the league fixtures. Return a JSON response contains the keys `success` and `message`

4.2.4 tiger_leagues.league

league.py

Exposes a blueprint that handles requests made to */league/** endpoint

`tiger_leagues.league.`**`browse_leagues`**`()`

Returns `flask.Response(mimetype='text/html')`

Render a page with a list of leagues that the user can request to join.

`tiger_leagues.league.`**`create_league`**`()`

Returns `flask.Response(mimetype='text/html')`

If responding to a GET request, render a template that can be used to create a new league.

Returns `flask.Response(mimetype='application/json')`

If responding to a POST request, return a JSON object confirming whether the league was created. The JSON sent in the POST request should have these keys: `league_name`, `description`, `points_per_win`, `points_per_draw`, `points_per_loss`, `registration_deadline` and `additional_questions`

`tiger_leagues.league.`**`index`**`()`

A user that already has an account will be redirected here. The user details will be present in the session object.

Returns `flask.Response(mimetype='text/HTML')`

If the user is a part of any leagues, render that league's homepage.

Returns `flask.Response(code=302)`

If the user is not a part of any league, redirect them to a page that allows them to browse available leagues.

`tiger_leagues.league.`**`join_league`**`(`*`league_id`*`)`

Parameters **`league_id`** – `int`

The ID of the league associated with this request

Returns `flask.Response(mimetype='text/html')`

Render the form that needs to be filled by users that wish to join this league.

Returns `flask.Response(mimetype='text/html')`

Process the form submitted by the user who wants to join this league. Return a JSON object that confirms the status of the join request.

`tiger_leagues.league.`**`league_homepage`**`(`*`league_id`*`)`

Parameters **`league_id`** – `int`

The ID of the league associated with this request

Returns `flask.Response(mimetype='text/html')`

Render a template for the provided league and the associated user. The template includes information such as `standings, media_feed, score_reports, upcoming_matches`, etc.

`tiger_leagues.league.`**`league_member`**(*league_id*, *other_user_id*)

Parameters **`league_id`** – `int`

The ID of the league associated with this request

Parameters **`other_user_id`** – `int`

The ID of the user whose data should be fetched.

Returns `flask.Response(mimetype='text/html')`

If `other_user_id == current_user_id`, the page shows the currently logged in user's stats.

If `other_user_id` does not belong to the same division as the currently logged in user, the stats of this other player are shown.

If the two users are in the same division, then the page shows both player's stats in the league in a side-by-side fashion.

`tiger_leagues.league.`**`leave_league`**(*league_id*)

Parameters **`league_id`** – `int`

The ID of the league associated with this request

Returns `flask.Response(mimetype='application/json')`

The JSON object contains a confirmation that the user was removed from the league.

`tiger_leagues.league.`**`process_score_submit`**(*league_id*)

Persist the score submitted by the user. The body of the POST object should have the following keys: *my_score, opponent_score, match_id*

Parameters **`league_id`** – `int`

The ID of the league associated with this request

Returns `flask.Response(mimetype='application/json')`

The JSON object contains the keys `success` and `message` whose values set appropriately.

`tiger_leagues.league.`**`update_responses`**(*league_id*)

Parameters **`league_id`** – `int`

The ID of the league associated with this request

Returns `flask.Response(mimetype='text/html')`

Render the form that needs to be filled by users that wish to update their responses to league-specific questions.

Returns `flask.Response(mimetype='text/html')`

Process the form submitted by the user. Return a JSON object that confirms the status of the submission.

4.2.5 tiger_leagues.user

user.py

Exposes a blueprint that handles requests made to */user/** endpoint

`tiger_leagues.user.`**`display_user_profile`**()

Returns `flask.Response(mimetype-'text/html')`

Render a template that contains user information such as: net_id, preferred_name, preferred_email, phone_number, room_number, associated_leagues

tiger_leagues.user.**modify_notification_status**()

Returns flask.Response(mimetype='application/json')

The JSON object is keyed by success and message

tiger_leagues.user.**update_user_profile**()

Returns flask.Response(mimetype='text/html')

Update the information stored about a user. Render a template that contains user information such as: net_id, preferred_name, preferred_email, phone_number, room_number, associated_leagues

tiger_leagues.user.**view_notifications**()

Returns flask.Response(mimetype='text/html')

Render the user's pending messages

4.2.6 tiger_leagues.decorators

decorators.py

A decorator is a function that wraps and replaces another function. If there's a functionality that you wish to extend to multiple functions, you should probably add the functionality as a decorator.

http://flask.pocoo.org/docs/1.0/patterns/viewdecorators/

tiger_leagues.decorators.**login_required**(*f*)

A decorator function that is used to confirm that a user is logged in before viewing/using certain URLs.

http://flask.pocoo.org/docs/1.0/patterns/viewdecorators/#login-required-decorator

Parameters **f** – function

A function that should be accessed only by authenticated users.

Returns flask.Response(code=302)

If the user isn't logged in, redirect them to the application's login page.

Returns function

If the user is logged in, return a function that is equal to the one that was passed as a parameter

tiger_leagues.decorators.**refresh_user_profile**(*f*)

A decorator function that is updates the user object stored in the session object. This is helpful when keeping the user up to date.

http://flask.pocoo.org/docs/1.0/patterns/viewdecorators/#login-required-decorator

Parameters **f** – function

A function that would benefit from an updated user object

Returns function

The incoming function is always returned as updating the user object is a side effect.

4.2.7 tiger_leagues.wsgi

wsgi.py

Expose Flask application object as the WSGI application. The WSGI app will then be ran by a WSGI server. Flask's built-in server is not suitable for production.

In our case, in `../Procfile`, we ask `gunicorn` to use the `app` object that is exposed in the `tiger_leagues` module.

http://flask.pocoo.org/docs/1.0/deploying/

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

t