This project:

Frozen Four is a tournament-style simulation game which allows the user to input eight teams of their choosing to compete against each other and eventually decide a winner. The game relies heavily on the passing of team data stored in a redis database to decide the winners of each round. The user will be put through a series of error-checks if they try to do anything they shouldn't be for the game. The game also renders a wide variety of html forms to improve the visual experience of the game.

Services:

History: This service is in charge of fetching all of the teams that have been entered by the user and rendering links to view more information about them. This data is rendered from the team service which hands them all of the teams fetched from the redis database. From this, it parses the data and renders links to another endpoint in the team service, displaying all of the information relating to that team.

Accessories: This service just handles simple routing to the credits and homepage html. For credits it just displays some basic text and the homepage is routed as the default from nginx. This page renders a few buttons that link to the tournament service, history service, and then the credits page from itself.

Add-team: This service is responsible for the majority of the application logic, servicing two endpoints: insertTeam and simulateTournament. Insert team is a post route which is hit from the team service which renders an html form with inputs provided by the user. These inputs are then passed to this endpoint where they're put into a structured dictionary and added to the database. The logic for this endpoint is somewhat complicated, but to simplify, essentially if the team entered is the first one, then it does an insert for the entire dictionary. Then, everytime this endpoint is hit after, it modifies the original dictionary by fetching the data from the redis instance, decoding the serialized data, and then passing the dictionary data provided by the user back into the database. Finally, it renders an html form verifying that the user provided their data and makes it easy to add another team. Simulate Tournament does exactly what it sounds like, it handles all of the simulation logic. First, using the random library, it decides the winning seeds from 8, 4, and 2 teams. Then, it fetches the dictionary from the db. Using the randomly chosen numbers, it fetches the winning teams and adds them each to their own dictionary. So one for semi-winners, one for finalists, and one for the actual winner of the whole tournament. From there, it either does the logic and renders a button for the user to go back, or it doesn't and tells the user as well.

Team: This service is responsible for most of the routing for the team data. It renders the addTeam form which then makes the post call to add-team, it also allows the routing from history to display the data for a given team. It does so by accessing redis and finding the stored key-value pair for that given team_name passed in as a parameter to the endpoint. Finally, it has several fetch functions with their entire job being to fetch the teams from the redis database

and pass them to the tournament service pages. These endpoints fetch the requested data for each round, de-serialize it and then store it in a utf-8 format (to avoid json) and pass that data back to the tournament service.

Tournament: This service handles all of the tournament pages and logic. The endpoints each make calls to their respective fetch functions from the team service, from there it returns that data and gets de-serialized. Once the teams for that given round are back to proper python dictionaries, that data gets passed into the forms. If the data doesn't exist for the given form, it won't actually render it. The other part of this is the default tournament page will show all of the tournament teams that have been entered up to the eight team limit, as they get added. So the user gets direct verification on if the team is added properly.

How to use:

Using the basic docker commands, all the user has to do is unzip the file into their directory and start it up using docker-compose build and then docker-compose run.

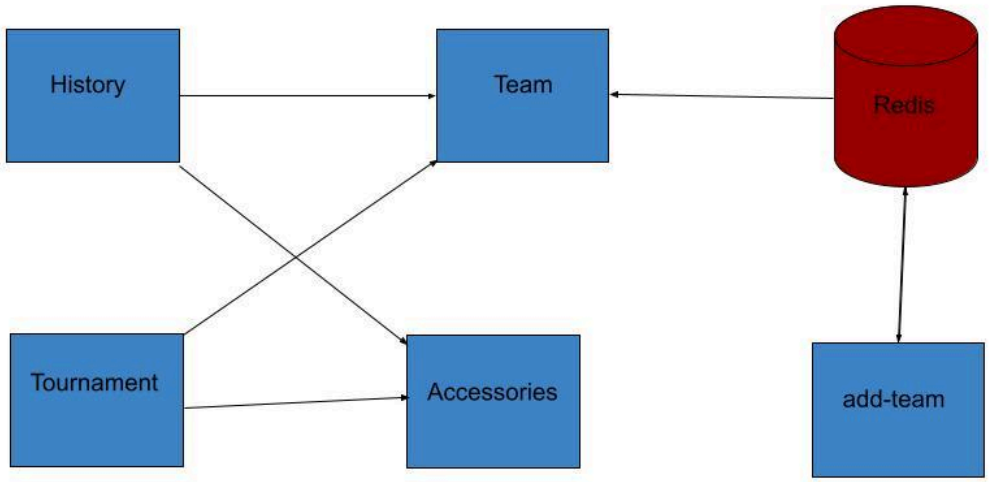
From there, the user should navigate to the start tournament page and enter eight teams. It doesn't require the inputs so if a user was lazy they could get away with not adding teams (I didn't get around to checking for that) but the best experience would be to at least add a team_name so they can see the tournament logic. From there, the user should add eight teams and then go back to the tournament page. Then, they can hit the simulate tournament button, go back to that page and then move forwards through the pages to see what teams won in each round. At the end the user will see a winner from those original eight teams. If the user wants to see more data about the teams they entered, they can go back to the home page and click history => view team and it will provide them with all of the team data they added.

Sources:

I leaned heavily on the slides for Cloud Computing, especially the ones from the last few weeks. I also used the random library docs, as well as the flask docs, the jinja docs (for templating), and ESPECIALLY the redis docs as I had no experience with storing serialized data before this project. I also used stack overflow a bit to debug although I don't have direct links to the threads I used for this project.

https://redis.io/docs/latest/
https://flask.palletsprojects.com/en/stable/
https://jinja.palletsprojects.com/en/stable/
https://docs.python.org/3/library/random.html

Graphic of Application Structure:



Screenshots of each route:

Screenshots of each route are included under the screenshots directory.