Electronic Golf League Scheduler

Design Document

Team Number: sddec21-03

Client: Tina Prouty

Advisor: Mai Zheng

Team Email Address: sddec21-03@iastate.edu

Team Website: https://sddec21-03.sd.ece.iastate.edu

Project Repository: https://git.ece.iastate.edu/sd/sddec21-03

Application Link:https://golf-league-scheduler.netlify.app/#/

Executive Summary

Development Standards & Practices Used

- Agile development
- Responsive Web Development
- Continuous integration
- Code reuse
- Data privacy
- Software unit testing
- User interface testing

Summary of Requirements

- Dynamic scheduling system of golf hole assignments for both individuals and teams
- Account for absences and change in partners within schedule
- Notify users of hole assignments and tee times via push notification
- Store data related to what team played one another, starting hole assignment, and score
- Fast load times and no delays in API requests
- Minimal cost or no cost to client
- Low technical maintenance
- User friendly interface

Applicable Courses from Iowa State University Curriculum

List all Iowa State University courses whose contents were applicable to your project.

- Software Engineering 309: Software Development Practices
- Software Engineering 319: Construction of User Interfaces
- Software Engineering 329: Software Project Management
- Software Engineering 339: Software Architecture and Design
- Computer Science 363: Introduction to Database Management Systems

New Skills/Knowledge acquired that was not taught in courses

- ReactJS
- Node.js
- Material UI
- Netifly
- Amazon Web Services
- Docker

Table of Contents

1 Introduction

1.1 Acknowledgement

We want to acknowledge Tina Prouty for the opportunity to work on this project. Thank you for your time and willingness to work with us. This project allowed us to work with industry-leading technology. Also, this project taught us to interact with a client and take an idea from conception to production.

Mai Zheng, we would like to thank you for your time and guidance during this project. We appreciate your technical expertise and feedback.

1.2 Problem and Project Statement

Problem Statement:

The women's golf league at Honey Creek course has gained more members than the current system can handle, an excel spreadsheet. Our client Tina Prouty would like to have an application that can enter individuals/team names and create hole assignments for them. If there is an absence by a team member, the absent golfer must be replaced, and the system must record the new golfer. If a team is missing altogether, then the application must reschedule the starting hole assignments. Each team member who golfed, the score they finished the course with, and their starting hole assignment from week to week must be recorded until the end of the golf league season. Currently, there are no free or low-cost options out there for a similar application. The available options do not have all the necessary features desired by the client.

Solution Approach:

The solution to this problem was developing a web application. A web application that requires minimal maintenance and operates per the requirements. Amazon Web Services manages the application's backend since it provides a free tier and free to the client. The application is deployed on Netfily, another free service. This solution provides a low setup cost and minimal monthly fees.

1.3 Operational Environment

Netfily hosts the web application, and Amazon Web Services manages the database. The user will have little to no interaction with these services, making the application maintenance-free. If the client needs to interact with either of these services, she will be able to consult the development team since they have provided their emails.

1.4 Requirements

Functional Requirements

- Enter individual's and team members' names.
- Create an initial schedule for the hole assignments.
- Account for absences by individuals/teams and able to enter replacements as needed.
- Schedule will dynamically switch based on individual/team absences.
- Keep track of golf scores, what team played, and the starting hole assignments throughout the duration of the league.

Economic Requirements

- Minimal monthly cost or no monthly cost.

User Interface Requirements

- Simple to use.
- Features are clearly defined.
- Design is modern and professional.

Version Control Requirements

- Each commit must contain a message with a detailed description for what is being pushed.
- New features will be implemented on feature branches.
- Feature branches being merged to master must go through a code review done by another team member.

Software Testing Requirements

- Unit Testing will be used to ensure the API is functioning properly.
- Manual user testing must be performed for every Merge Request before it can be completed.
- User Acceptance Testing will be used to ensure requirements are met.

Software Security Requirements

- Application will use modern security practices to prevent any known security flaws.
- Developer team and client are the only ones to have access to root accounts for third party services.

1.5 Intended Users and Uses

The end-user is the person who is administering the golf league. The administrator will have full access to the application, which includes changing hole assignments and teams.

1.6 Assumptions and Limitations

Assumptions:

- Information from previous technology will be used (excel document).
- Website will be accessed from mobile and desktop computers.
- There will be only one user using the application, essentially an administrator.
- The application will only be used for the women's golf league.
- Application will only be used for one golf course (Honey Creek).
- Two players on each team, able to be changed based on member absences.
- Automatic rescheduling.

Limitations:

- Must follow standard design principles.
- Users must be able to access the application from the golf course main website.
- The application must be low maintenance to the client.
- Budget constraints, the cost should be little to no expense to the user.

1.7 Expected End Product and Deliverables

The expected end product is a responsive web application. Amazon Web Services will host the database. The benefit of this is that it will allow the server to be up at all times, and the application will not need a dedicated server that is managed by the development team.

The team delivered an end product that is in production.

2 Design

2.1 Previous Work And Literature

There are a few competing applications in the same space as our product, with PlayPass and GolfSoftware being the most prevalent options. Both options are free or cheap and offer the ability to schedule a tournament or round robin league for a group of golfers. There are a few points that separate this project from these applications.

This project is built with a specific user set in mind, which allows the team to fine tune certain choices to deliver a more comprehensive product. The first point is the inclusion of automatic pairing shuffling and reassignment. Neither of the leading products offer this option, any changes in personnel would require a manual regeneration of the tournament.

This project also differentiates itself by tracking the weekly scores, winners, and standings of each round in the league. GolfSoftware tracks which golfer in a pairing wins, but not scores or any side competitions such as closest to the pin.

Finally, this project lets groups compete as a team. Neither of the competing products offer this capability.

2.2 Design Thinking

There were a large number of ways to accomplish this task, being that there was no prior application to build off of or integrate with. While helpful in a sense, it posed a different challenge of narrowing down our options to a select few that were simpler to analyze and compare. To start, we selected a few requirements that would play a large part in the decision making process, such as:

- Cost needs to be kept low.
- Project needs to be easily extensible, so that the team can add new features if opportunities for improvement arise.
- The application must be user friendly for a non-technical audience.

With those requirements laid out, we were able to remove options such as providers that have a monthly cost for hosting or do not have a free tier. It was helpful that the amount of traffic expected to flow through the application is low, meaning that it is likely that it will stay within the free tier of some platforms.

Possible development technologies for the project were compared and selected based on the familiarity of the tool/language/platform of the team, simplicity, and cost. Cost was the dominant feature in the decision making process.

2.3 Proposed Design

IEEE defines multiple software engineering standards. Of those standards there are ones that are more applicable to our project. The types of standards we defined as being applicable to our project were customer interaction, product development, and testing. IEEE has a standard called *'Standard for Application and Management of the Systems Engineering Process'*. In this standard, it is described how to meet the customers requirements, needs and constraints.

The next standard that is applicable to our project is the *'Software Life Cycle Processes'*, this engineering standard describes the framework for software lifecycle management. This standard goes into detail describing how to develop, maintain, and operate the software. Unit testing is going to be largely incorporated into our project. The standard that relates to unit testing is *'Standard for Software Unit Testing'*, in this standard it describes best practices and approaches to implement unit testing for our project.

There are a few software development standards that played a large role in the development and evaluation of the various application designs presented below.

API - Microservices/Serverless

Each microservice runs on a AWS Lambda instance to provide low-latency operation, even during cold start situations. AWS DynamoDB is used as the main database for the application. To facilitate this, the team utilized the Serverless Framework with NodeJS to manage infrastructure as code and reduce the complexity of deploying to AWS and track all changes to infrastructure inside of git for version control.

Each service contains a single domain area. By doing so, each service is very loosely coupled to other services, allowing for rapid iteration between services without causing breaking changes elsewhere. Listed below are the services that the team will implemented

Services:

- Auth: Handles authentication and authorization for users.
    - Login
    - Logout
- Users: Tracks individual details and performance for a golfer throughout the season.
    - Personal details
    - Score
    - Season standing
- Teams: Groups of users that make up individual teams. Handles team shuffling and result tracking.
    - Season standing
    - Current members
- Outings: Handles scheduling and team pairings for each golf outing.
    - Schedule
        - Team
        - Starting Hole
    - Upcoming outings and when they start
- Leagues: Overall league standings and organization.
    - Team Leaderboard
    - Contest Winners

Pros:

- Utilizing a serverless architecture with microservices aids in keeping costs low by only running the application when absolutely necessary.
- Microservice applications are extensible by design. As the project progresses, the team will have the ability to add features if requested without the need to alter other aspects of the project.
- HTTP Request validation handled automatically by AWS.
- Simpler testing through mocking other services.
- Auto-provisioning of resources from AWS.

Cons:

- Local development requires a lot of setup and can be problematic if not configured properly.
- Steeper learning curve for communication between services.
- Continuous Integration and Deployment is complicated.

Application UI - Web Application

A web application provides a cross platform and device-agnostic user interface for the project. The team used ReactJS as the application framework to aid in development, with MaterialUI as the design foundation to ensure that the application is visually appealing. React Hooks was preferred over class based components during development.

The project is broken up into individual features, with a main component to handle routing, application context, and state management. Each feature is structured similarly to the overall application, with a main component and sub-directories for internal components. Each directory holds these files:

- ComponentName.js - Markup and state management for the component.

The web application uses Netlify for free hosting.

Pros:

- No need for installation or updating once the application is deployed.
- Works with all devices.
- Free hosting.

Cons:

- An internet connection is required to use the application.

2.4 Technology Considerations

AWS was chosen over Azure and Google Cloud Platform due to the generous free tier for their services. AWS costs for the application are expected to be less than one dollar a month.

DynamoDB was chosen over other SQL and NoSQL database options due to the AWS free tier. Its low latency for reads and writes also makes it ideal for serverless applications where runtime is what monthly costs are based on.

NodeJS was chosen over other Lambda runtimes due to its speed during cold-starts, which we expect to be the majority of interactions with the API. Jest will be used for unit testing.

Netlify hosting was chosen over AWS Cloudfront for the web application due to the simplicity of deployment and domain configuration.

2.5 Development Process

The team utilized a Kanban development process for organizing and selecting tasks. A Kanban process was selected for a few reasons. Firstly, allowed the team to add more items and features to the backlog as the project progresses. This helped with prioritization and let the team extend the feature set of the end product without overreaching. Secondly, due to Kanban being a looser form of Agile development than the strict sprint-based method of Scrum, the teams varying schedules and workloads throughout the semester can be accounted for more efficiently.

2.6 Design Plan

The design for this project was to separate the backend and frontend into a web application and REST API.

Web Application

The user interface for the project is a Single Page Application written in React. Each page of the application is contained within its own directory, with all components pertaining to the page held within that directory. In doing so, adding new pages to the application was trivial and did not affect the operation of other pages.

Interface components were split by functionality on a case by case basis. Whenever a component gained more functionality than originally intended, the developer broke it up into smaller components for readability.

REST API

The backend for the project was written in NodeJS, with each endpoint (GET, POST, UPDATE, DELETE) being hosted on it's own AWS Lambda instance. Prior to reaching its destination lambda, a request passes through an AWS API Gateway.

The API Gateway does the following:

- Validates the request body and responds with an error message if it is not properly formatted.
- Attaches the requestor's authorization credentials so the endpoint can return user-specific data.
- Pass the request to the correct lambda endpoint.

Each service, shown in Figure 1 as a box of nodes, will have its own table in the DynamoDB database schema. DynamoDB is distributed by design, so there is no need to have multiple instances running.

AWS API Gateway, Lambda, and DynamoDB will run within their respective free tiers. The only exception is that API Gateway will have a negligible cost (< $0.50) after the first year of use.

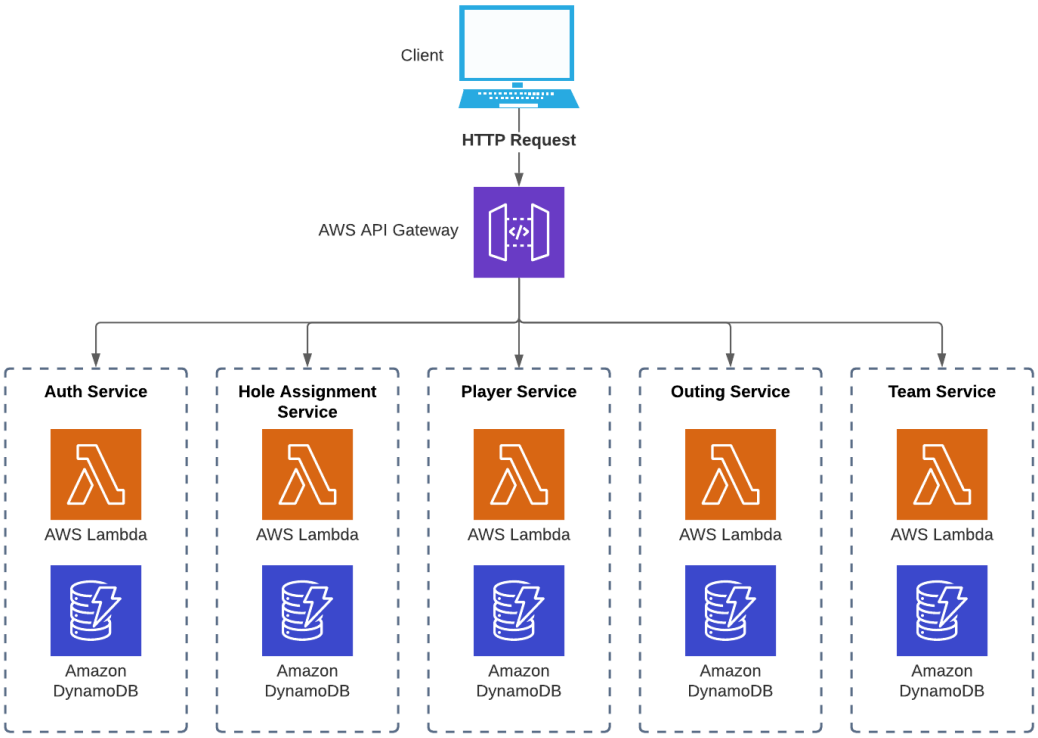

Fig. 1 - High level backend architecture

3 Implementation

3.1 Project Management

To help manage the project the team utilized Trello and Slack. Each tool served a pivotal part in the success of this project.

Slack was the communication tool used by the team. This allowed for easy communication between team members. We had separate channels for communication. The channels we utilized were: general, code-review, meetings, and documentation.

Trello was used to manage tasks within the project. Each task was broken down to address a specific requirement or defect. The task then had a label to specify if it was a backend or frontend task. There were four columns on the Trello board. The “Backlog” column specified tasks that were unassigned and able to be worked on. The next column was “In Progress”, this column was to show the tasks that were currently being worked on. Once a task is in this column the developer working on this should assign themselves to the task. After the implementation of the task is done and a MR is open the task should be moved to the “In Review” column. Once the code is merged the task is then moved to the “Completed” column.

3.2 Development Approach

The first step in development was to take the requirements outlined by the clients then apply them to wireframe mockup. This was a fast and flexible approach to designing the user interface. It allowed for all members of the development team to be in agreement on how the user interface should appear.

The front-end developers began to build the user interface based on the wireframes. There was strict enforcement of style to ensure consistency. Since the backend development was still in progress the front-end developers used artificial data to continue with development.

The backend developers started to develop each service. Every service had a request for creating, reading, updating, and deleting. Postman was used to ensure implementation was being done correctly.

Once the backend was developed the front-end developers began to implement the API calls to the backend. This allowed the application to begin mocking the end product. At this point the user interface was able to support the CRUD features of the backend.

3.3 Scheduling Algorithm

Scheduling hole assignments for a golf outing is done on two different occasions. When an outing is first created, an initial schedule will be generated, using every team in the database. Once players have been checked-in on the day of the outing, an administrator has the ability to shuffle the players to form as many full teams as possible. This reduces the length of the day overall, and ensures that as many players get the chance to play with a partner as possible.

The implementation for initial schedule generation is relatively straightforward. The list of teams in the database is placed in a randomized order. The first team is then paired with the second, the third team with the fourth, and so on until all teams have a pairing.

Shuffling teams is a more involved process. To keep outings as short as possible, the shuffling algorithm must favor early hole assignments over those occurring later in the day. It also must provide reference so that each substituted team member will still have their score counted toward their original team's collective score for the season.

Due to these requirements, it was determined that a custom algorithm would be required for this implementation. (See Appendix II for a pseudo-code representation of the scheduling algorithm)

3.4 Security Concerns/Countermeasures

Security was not the highest priority for this project. Some basic measures were made to ensure that no extra costs would be incurred over time, and that data could not be stolen from the application. Due to the low risk associated with any data from the application leaking, efforts were directed towards other aspects of the application.

That said, there are some notes to be made on security for the application.

- All communication between the API and web client is sent over HTTPS to ensure safe transmission of data.
- The web client is password protected with an admin password to lower chances of data being edited unwittingly.
- All backend services are placed behind a VPC on AWS to ensure separation from the public domain. A public API Gateway routes traffic through to the private application if traffic is properly formatted and from the correct source.

3.5 Deployment/Project Hosting

Deployments were done manually for both the API and web client.

Hosting for the web client was done via Netlify. Netlify provides hosting that delivers static content from edge servers across the goal for free. This ensures that not only will the application load quickly when accessed, it will also never incur costs in doing so.

The API was hosted on AWS, utilizing their generous free tier pertaining to the API Gateway and Lambda platforms. To help with deployments and resource provisioning, the team decided on using the Serverless framework to handle the heavy lifting in those aspects. Each new deployment removes old versions of the application before deploying to maintain a predictable platform usage level and stay under the free tier limits.

The application uses DynamoDB for it's primary database. DynamoDB is a cloud-native database that is provided by AWS. This removes the need to create database instances and simplifies use.

3.6 Design Evolution From Last Semester

The main architecture of the application follows the originally proposed design. The biggest changes that were made fell into the topic of git workflow and Continuous Integration and Deployment.

The team had planned to run tests and deploy the application using a pipeline within Gitlab. Issues arose around the fact that ISU's Gitlab is self-hosted, which most platforms do not support without upgrading to a Pro plan (Netlify in this particular case).

Tests had to be run locally by Merge Request reviewers due to issues running a kubernetes cluster for the team within Gitlab. With this being the default option for Gitlab CI, the team opted to not utilize those Gitlab features.

4 Testing

4.1 Unit Testing

Unit testing was executed within each microservice in isolation. Business logic and data access were tested, resulting in 89% test coverage for all business logic.

A local DynamoDB database was used to verify the correctness of all database queries. The database was cleared and re-seeded before executing each unit test. In doing so, each test was done in isolation, with no chance of interference from other tests.

4.2 Integration Testing

A Postman collection was created to test all API endpoints for the application. By doing so, the team had confidence that each endpoint was behaving as it should be when changes were being made to the web application. Every test was required to run successfully before a Merge Request could be completed.

4.3 Interface Testing

Interface testing was done manually by each member of the team both during development and by a peer during Code Review.

5 Closing Material

5.1 Conclusion

The result of two semesters of development of this Golf League Scheduler has led to a product that the team stands behind and hopes will be of great use to our client. We are happy to help with any issues that arise next Summer when it starts to be used, and have provided our contact information to the client

5.2 References

Important Engineering Software/Hardware Design Standards. Software/Hardware Design Standards. (n.d.). http://users.encs.concordia.ca/~ecewebdv/EDS/Software/std_list.htm.

Appendix

I - Operation Manual

A live version of the Golf League Scheduler is available at https://golf-league-scheduler.netlify.app/.

The Admin Password for the demo is:

- **password**

Notes about the demo:

- The demo is currently seeded with example data, which will be removed after the semester has finished.
- Each page may load data slowly the first time. AWS Lambda instances sit inactive until invoked to reduce cost. Once started, the instance will shut down after 15 minutes of inactivity.

II - Code Snippets

- Schedule Shuffling Algorithm Pseudo-code

```
fillMissingSlots():
    // empty lists that will contain hole assignments, corresponding to
    // what should be done with that hole assignment

    // teams and players that need a new team/pairing

    i ← 0
    j ← length of allHoleAssignments
    while (j < i):
        // store players and teams that need to be moved
        orphanedPlayers, orphanedTeams

        // find next hole assignment with absent players
        // - must have at least one absent player
        i ← index of next hole needing players

        // find next hole assignment to pull players from
        // - must have at least one absent player
        j ← index of next hole with players than can be moved

        // make sure that i and j have not crossed after the previous loops
        if (j > i):
            // the i'th hole assignment will be retained
            // the j'th hole assignment will be deleted

        // grab orphaned team from allHoleAssignments[j] if possible
        orphanedTeams ← any teams with both players confirmed

        // grab orphaned players from allHoleAssignments[j] if possible
        orphanedPlayers ← any players without a confirmed partner

    // loop through hole assignments that will be retained,
    //     but need slots filled
    foreach (hole assignment in toUpdate):
        // replace a team if one is missing
        // replace any missing players on either team

    while (there are still orphaned players or teams):
        // create new hole assignment object

        // place any remaining orphaned teams on the new hole assignments

        // place any remaining orphaned players on new teams

        // update toCreate to later save the new hole assignment

    // complete database operations for the toUpdate, toDelete, and toCreate lists
```