



Capstone Project: Ft_Transcendence

Group Members' Full Names:

Jafar Alwaheidi

Joe Shaju

Zahra Syyida

Shaik Mazin Bokhari

Huong Bui Vu

Group Members' Intra Logins:

jalwahei

jperinch

zsyyida

sbokhari

hbui-vu

30th September 2024

Abstract

This Capstone project aims to develop a full-stack web application based on the famous Pong game, utilising various modern technologies to improve user experience and security. Utilizing Bootstrap for the frontend, Django for the backend, and PostgreSQL for database management, the application ensures solid functionality and scalability. Key features include secure user authentication with two-factor authentication (2FA), remote authentication via the 42 login system, and advanced monitoring and logging capabilities through the ELK stack and Prometheus. This project serves as a practical demonstration of implementing modern web application standards.

Contents

Abstract	i
List of Figures	v
List of Tables	vi
List of Abbreviations	vii
1 Introduction	1
1.1 Introduction	1
2 Software Development Life Cycle (SDLC)	2
2.1 Planning requirements	2
2.2 Designing the system architecture	2
2.3 Implementing the code	3
2.4 Testing the application	3
2.5 Future improvements	3
3 Requirement Analysis	4
3.1 Module Picks	4
3.1.1 Web	4
3.1.2 User Management	4
3.1.3 Cybersecurity	5
3.1.4 DevOps	5
3.1.5 Accessibility	5
3.2 Gathering Requirements	5
3.3 Functional Requirements	5
3.3.1 Game Functionalities	5
3.3.2 User Management	6
3.3.3 Authentication	7
3.4 Non-Functional Requirements	7
3.4.1 Hardware	7
3.4.2 Application Serving	7
3.4.3 Database	7
3.4.4 Performance	7
3.4.5 Compliance	8
3.5 Technical Requirements	8

3.5.1	Hardware	8
3.5.2	Software	8
3.5.3	Application Serving	8
3.5.4	Database	9
3.5.5	Logging	9
3.5.6	Monitoring	9
3.5.7	Containerization	9
3.5.8	Security	9
3.5.9	Integration	10
4	Design	11
4.1	Backend Design	11
4.2	Security Design	11
4.3	DevOps Design	12
4.4	Frontend Design	13
5	Implementation	15
5.1	Adopting Agile for Our Development Process	15
5.1.1	Trello	16
5.1.2	Code Reviews, Git Workflow, Git Conflicts, and Coding Standards	16
5.1.3	Gantt Chart	16
5.1.4	Risk Management Register	17
5.2	Web Modules Implementation	19
5.2.1	Major: Use the Django Framework as Backend	19
5.2.2	Minor: Use the Bootstrap Toolkit as the Front-end Framework	21
5.2.3	Minor: Use PostgreSQL as the Database for the Backend	21
5.3	User Management Modules Implementation	21
5.3.1	Major: Implement Standard User Management, Authentication, and Users Across Tournaments	22
5.3.2	Major: Implement OAuth 2.0 Authentication with 42	22
5.4	Cybersecurity Modules Implementation	22
5.4.1	Major: Implement WAF/ModSecurity with Hardened Configuration and HashiCorp Vault for Secrets Management	23
5.4.2	Major: Implement Two-Factor Authentication (2FA) and JWT	24
5.4.2.1	JWT (JSON Web Tokens)	25
5.5	DevOps Modules Implementation	27
5.5.1	Major: Set up infrastructure for log management with ELK stack (Elasticsearch, Logstash, Kibana)	27
5.5.2	Minor: Set up a monitoring system using Prometheus and Grafana	30
5.5.3	Major: Design the backend as microservices in Docker	31
5.5.4	Accessibility Modules Implementation	32
5.5.4.1	Minor: Add support for expanding Browser Compatibility	32
5.5.4.2	Minor: Server-Side Rendering (SSR) Integration	33
6	Testing	35
6.1	Performance Tests	35
6.2	Reliability Tests	35
6.3	Security Tests	36

6.4	Enacting Changes After Testing	36
7	Evolution	37
7.1	Enhance Logging	37
7.1.1	Add Logs for Logging and Monitoring Services	37
7.1.2	Refine Logstash Filters	37
7.1.3	Automated Logging Alerts	38
7.2	Centralizing Secrets Management	38
7.2.1	Auto Unsealing	38
7.2.2	Automating SSL Certificate Management	38
7.2.3	Leverage Vault’s Transit Engine	39
7.2.4	Dynamic Secrets	39
7.2.5	Customized Access Control	39
7.3	Take the Game Online	39
7.4	Allow for Bigger Tournaments	39
7.5	Add Another Game	40
	Conclusion	41
	References	42
	Appendix A Wireframes and User Journey	44

List of Figures

4.1	Deployment and DevOps Architecture	13
4.2	Flowchart	14
5.1	Agile Process Flow	16
5.2	Gantt Chart - Part 1	17
5.3	Gantt Chart - Part 2	17
5.4	Authentication flow	25
5.5	Index Summary	29
5.6	Logs Visualization	29
5.7	Policy Summary	30
5.8	Grafana Overview	31
5.9	SSR Overview	34
A.1	Signup & Login	45
A.2	Game & History	46
A.3	Tournament	47
A.4	Enabling 2FA	48
A.5	Adding Friends	49
A.6	View friend Profile	50

List of Tables

5.1 Risk Matrix	18
5.2 Risk Register	19

List of Abbreviations

2FA	Two-Factor Authentication
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
AWS	Amazon Web Services
CPU	Central Processing Unit
CRS	Core Rule Set
CSR	Client-Side Rendering
CSRF	Cross-Site Request Forgery
CSS	Cascading Style Sheets
CURL	Client URL
DevOps	Development Operations
DRF	Django REST Framework
ELK	Elasticsearch, Logstash, Kibana
Git	Global Information Tracker
HTML	Hyper Text Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
ILM	Index Lifecycle Management
JWT	JSON Web Token
KV	Key Value
MTV	Model-Template-View
MVC	Model-View-Controller
OAuth	Open Authorization
ORM	Object-Relational Mapping
OTP	One-Time Password
OWASP	Open Web Application Security Project
PKI	Public Key Infrastructure
QR	Quick Response
REST	Representational State Transfer

RESTful	REST-Compliant Systems
SDLC	Software Development Life Cycle
SEO	Search Engine Optimization
SPA	Single Page Application
SQL	Structured Query Language
SSL	Secure Sockets Layer
SSR	Server-Side Rendering
TLS	Transport Layer Security
TOTP	Time-Based One-Time Password
UI	User Interface
URL	Uniform Resource Locator
WAF	Web Application Firewall
XSS	Cross-Site Scripting

Chapter 1

Introduction

1.1 Introduction

transcendence marks the final project in Ecole 42 core curriculum, where students must develop a full-stack web application that features a Pong game. The project requires students to integrate various modules into both the front and back end of the application, allowing for the incorporation of various technologies and features.

Our team selected Django as the back-end framework, Bootstrap for the front-end, and PostgreSQL as the database, ensuring a solid foundation for the application's functionality. To enhance user experience, we integrated expanding browser compatibility, server-side rendering, standard user management, and remote authentication via the 42 login system. For security, we installed ModSecurity as an additional firewall layer and HashiCorp Vault to securely store sensitive environmental variables and keys. Furthermore, we implemented secure user authentication with two-factor authentication (2FA) and JSON Web Tokens (JWT). On the DevOps side, we integrated advanced monitoring and logging tools to optimize performance and streamline operations. We utilized the ELK stack (Elasticsearch, Logstash, Kibana) for comprehensive log management and implemented Prometheus and Grafana for real-time tracking of application performance. Additionally, we containerized the entire application using Docker microservices, allowing for efficient scaling and simplified deployment. In the following chapters, we will explore each of these modules in more detail and how they shape the functionality and architecture of our application.

Chapter 2

Software Development Life Cycle (SDLC)

The Software Development Life Cycle (SDLC) provides a structured, step-by-step approach to building and deploying software systems, making it a valuable tool for software development teams. It encompasses six key stages: planning the application's requirements, designing the system architecture, implementing the code, testing the application, deploying the solution, and maintaining the project over time. We used the SDLC framework to efficiently manage our resources and timeline, ensuring that our project adhered to 42's guidelines and was completed within the set deadlines.

2.1 Planning requirements

In this phase, we focused on defining what the system needs to achieve and identifying any constraints for our project. We considered the key requirements of each module and set clear goals for the project. After selecting the modules, we decided on each module's priority and discussed them with other students to get more insight on how we might implement them. We also documented both the functional and technical requirements to create a solid foundation for the development process.

2.2 Designing the system architecture

After completing the requirements analysis, we moved into the design phase, where we planned both the frontend and backend of the system. In this stage, we focused on how the various components of the program would interact with each other to create

a seamless user experience. We also defined the technical specifications, detailing how each module would function and communicate with the others, laying the groundwork for the system's architecture and overall performance.

2.3 Implementing the code

After determining the program architecture, we began the implementation process. We worked in stages based on each module's priority from our requirements analysis, and assigned modules to each team member to work on. We used a variety of tools such as github, a gantt chart, a risk register document, trello, and the agile framework to manage our work.

2.4 Testing the application

We periodically tested the code after each stage of code implementation to ensure that the code was running properly before moving on to the next stage. This involved a variety of tests, including unit testing to validate individual components, integration tests to assess how different parts of the system worked together, and user tests to gather feedback from real users. This testing strategy allowed us to identify and resolve bugs early, ensuring a smoother development process as we built the code.

2.5 Future improvements

After deployment, we need to focus on system maintenance and consider how we can update the game. This includes monitoring performance, addressing any issues that arise, and implementing enhancements based on user feedback. We should also plan for regular updates to introduce new features, fix bugs, and ensure the game remains engaging and enjoyable for players.

Chapter 3

Requirement Analysis

Initially, each team member independently reviewed the project guidelines to determine which modules aligned with their interests. Following this, we gathered as a team to discuss our picks and assign the modules. Knowing that we wanted to also complete the bonus, we decided to take on the 7 major modules and 5 minor modules as follows:

3.1 Module Picks

The project employs a layered architecture, with key components structured as follows:

3.1.1 Web

Major: Use the Django framework as backend

Minor: Use the Bootstrap toolkit as the front-end framework

Minor: Use PostgreSQL as the database for the backend

3.1.2 User Management

Major: Implement standard user management, authentication, and users across tournaments

Major: Implement OAuth 2.0 authentication with 42

3.1.3 Cybersecurity

Major: Implement WAF/ModSecurity with Hardened Configuration and HashiCorp Vault for Secrets Management.

Major: Implement Two-Factor Authentication (2FA) and JWT.

3.1.4 DevOps

Major: Set up infrastructure for log management with ELK stack (Elasticsearch, Logstash, Kibana)

Minor: Set up a monitoring system using Prometheus and Grafana

Major: Design the backend as microservices in Docker

3.1.5 Accessibility

Minor: Add support for expanding Browser Compatibility.

Minor: Server-Side Rendering (SSR) Integration.

3.2 Gathering Requirements

To make sure we fully understood the project requirements, we carefully read through the rubrics and noted the key requirements for each module. We also spoke with students who had already completed the project to get their insight on the modules and advice on how they tackled each one. On top of that, we watched YouTube videos of games made by students from other 42 campuses to get ideas for the game's look and possible features.

3.3 Functional Requirements

Functional requirements are the specific tasks and features the project must have in order to meet the user's expectations and needs, specifying the software's tasks, user functionalities, interactions, constraints, and use cases.

3.3.1 Game Functionalities

Tournament Mode:

Task: Users can sign up for a tournament with random matchmaking.

Interaction: Four users join a tournament, and two users play head-to-head.

Constraint: Only four users can be in a group.

Use Case: Players join the tournament based on a random matchmaking algorithm.

Two-Player Matches:

Task: Two users play against each other in a match.

Interaction: Users interact by controlling paddles and competing in real-time.

Use Case: Two users face each other in a real-time game.

3.3.2 User Management

User Registration:

Task: Users can register an account and create a profile.

Interaction: Users input their nickname, upload an avatar, and create a secure password.

Constraint: Username and email must be unique. Passwords must meet complexity requirements (1 capital letter, 1 digit, 1 special character, etc.).

Use Case: A new user registers and sets up a profile.

User Login and Session Management:

Task: Users log in and manage their profiles (avatar, password).

Interaction: Users can view match history, change avatars, or update settings.

Constraint: Only one user can be logged in at a time (local game).

Use Case: A returning user logs in and updates their account settings.

Match History:

Task: Users can view their match history and friends' match histories.

Interaction: Users navigate to the match history section and view detailed stats.

Use Case: A player views detailed statistics of past matches and friends' match history.

Password Management:

Task: Users can change their password.

Interaction: Password changes require meeting strict complexity rules.

Constraint: Passwords must be between 8–128 characters with specific criteria.

Use Case: A user changes their password.

Friend Management:

Task: Users can add friends and accept friend requests.

Interaction: Users can view and interact with friends, and see their match histories.

Use Case: A user views another registered user's match history.

3.3.3 Authentication

Two-Factor Authentication (2FA):

Task: Users can enable or disable 2FA for added security.

Interaction: 2FA setup is available in settings.

Use Case: A user enables or disables two-factor authentication.

42 Authentication Integration:

Task: Users can log in using 42 OAuth for authentication.

Interaction: Users choose between logging in with local credentials or 42 authentication.

Use Case: A user logs in using 42 authentication.

3.4 Non-Functional Requirements

Non-Functional Requirements are criteria that specify the operation and qualities of a system that affects how well a system performs its tasks.

3.4.1 Hardware

Networking: Latency is not really an issue on local device.

Storage: Scalable storage solutions for database persistence and logging systems to store game history, user data, and application logs.

3.4.2 Application Serving

Gunicorn: Ensures high performance.

Nginx: Handles load balancing efficiently.

3.4.3 Database

Passwords must be securely hashed before storage in the database.

3.4.4 Performance

- **Microservices:** The application can be updated service by service.
- **Scalability:** The application is designed to scale vertically by upgrading hardware resources.

- **Load Balancing:** Nginx ensures efficient load distribution across multiple application instances, preventing bottlenecks and improving response times.

3.4.5 Compliance

- **Audit Logs:** ELK will be used to maintain audit logs of user actions and system changes for regulatory compliance and security audits.
- **Encryption:** Sensitive data will be encrypted both at rest (via PostgreSQL) and in transit (via SSL/TLS), meeting industry standards for data security.

3.5 Technical Requirements

Technical Requirements outline the essential constraints and specifications necessary for the project, ensuring the system's functionality across hardware, software, security, integration, performance, and compliance. It focuses on the tools and components used in the system.

3.5.1 Hardware

Servers: The application will be deployed on on-device servers with sufficient resources (CPU, memory, storage) to handle web traffic, database queries, and background tasks.

3.5.2 Software

Frontend: Built using HTML, CSS, and Bootstrap to provide a responsive and user-friendly interface.

Backend: Developed with Django, responsible for business logic, database interactions, and handling real-time user requests.

3.5.3 Application Serving

Gunicorn: Serves dynamic content and manages incoming requests for the Django application.

Nginx: Acts as a reverse proxy and serves static files (CSS, JavaScript, images).

3.5.4 Database

PostgreSQL is the primary database, managing and storing key information:

- Two-Factor Authentication (2FA) keys and status.
- User profile details (avatars, nicknames).
- Game history (wins, losses, timestamps).
- User status (online/offline), friends, and securely hashed passwords.

3.5.5 Logging

The ELK Stack (Elasticsearch, Logstash, Kibana) will be integrated for comprehensive logging, analysis, and visualization of system and application logs.

3.5.6 Monitoring

Prometheus: Collects real-time metrics related to application performance, resource usage, and user activity.

Grafana: Provides a visual dashboard for monitoring system performance and generating alerts for critical metrics.

3.5.7 Containerization

Docker: Used to containerize the application and its services, enabling consistent deployments across environments, from development to production.

3.5.8 Security

ModSecurity: A web application firewall that protects against common vulnerabilities like SQL injection and cross-site scripting (XSS).

HashiCorp Vault: Manages sensitive information such as API keys and credentials, ensuring encryption, controlled access, and audit logging for security compliance.

3.5.9 Integration

- **Prometheus and Grafana:** Seamlessly integrated for monitoring, collecting system metrics, and providing real-time performance insights.
- **ELK Stack:** Works in conjunction with application logging to centralize and process logs for easier troubleshooting and compliance auditing.
- **Docker:** Simplifies integration across different environments by containerizing all services (Django app, PostgreSQL, ELK stack) for consistent performance.

Chapter 4

Design

4.1 Backend Design

After a client sends a request to our web application, the request follows a structured pathway: The request passes through a firewall before it is passed to Nginx, which handles static assets like images, CSS, and JavaScript directly. For dynamic requests that require server-side processing, Nginx forwards them to Gunicorn, the application server. Nginx also manages SSL encryption and decryption for the web application for secure communication over HTTPS.

When Gunicorn receives a forwarded request, it processes it via the WSGI interface, handing it off to Django. Django processes the request, accessing the PostgreSQL database if necessary to retrieve user data such as authentication details, profiles, or friend interactions. Django communicates with the frontend through RESTful APIs, managing tasks like user authentication and data retrieval.

Once Django has processed the request, it returns the response to Gunicorn, which in turn sends it back to Nginx. Finally, Nginx delivers the response back to the client.

4.2 Security Design

We implemented a range of security measures to safeguard different aspects of our software system, ensuring comprehensive protection across all components. Client requests pass through the ModSecurity web application firewall, which screens for malicious content before forwarding the request to Nginx for further handling. Communication between containers is secured with HTTPS, including those without an exposed port, ensuring that data exchanged between services remains encrypted and protected.

Hashicorp Vault is used to securely store sensitive environmental variables and the Fernet key used for two-factor authentication. User passwords are hashed by Django before they are stored in the database. Users can choose to securely register and/or log in using OAuth2 with 42's login system. JWT (JSON Web Tokens) are used for secure session management. Users can also choose to enable two-factor authentication for their account after registration.

4.3 DevOps Design

The system architecture is designed using Docker to implement a microservices approach, where each service runs in its own isolated container, communicating with other containers via a shared Docker network.

Logging for the Nginx, ModSecurity, Vault, and the web application services is managed by the ELK stack (Elasticsearch, Logstash, Kibana, and Filebeat). Filebeat collects log data and sends it to Logstash for processing, which then forwards it to Elasticsearch for storage and indexing. The logs can be visually explored and analyzed via the Kibana dashboard on port 5601.

Prometheus is used to monitor the health and performance of all key services, including Django, Nginx, the ELK stack, PostgreSQL, and Vault. Monitoring data is displayed in real-time through customizable visual dashboards accessible via Grafana on port 3000.

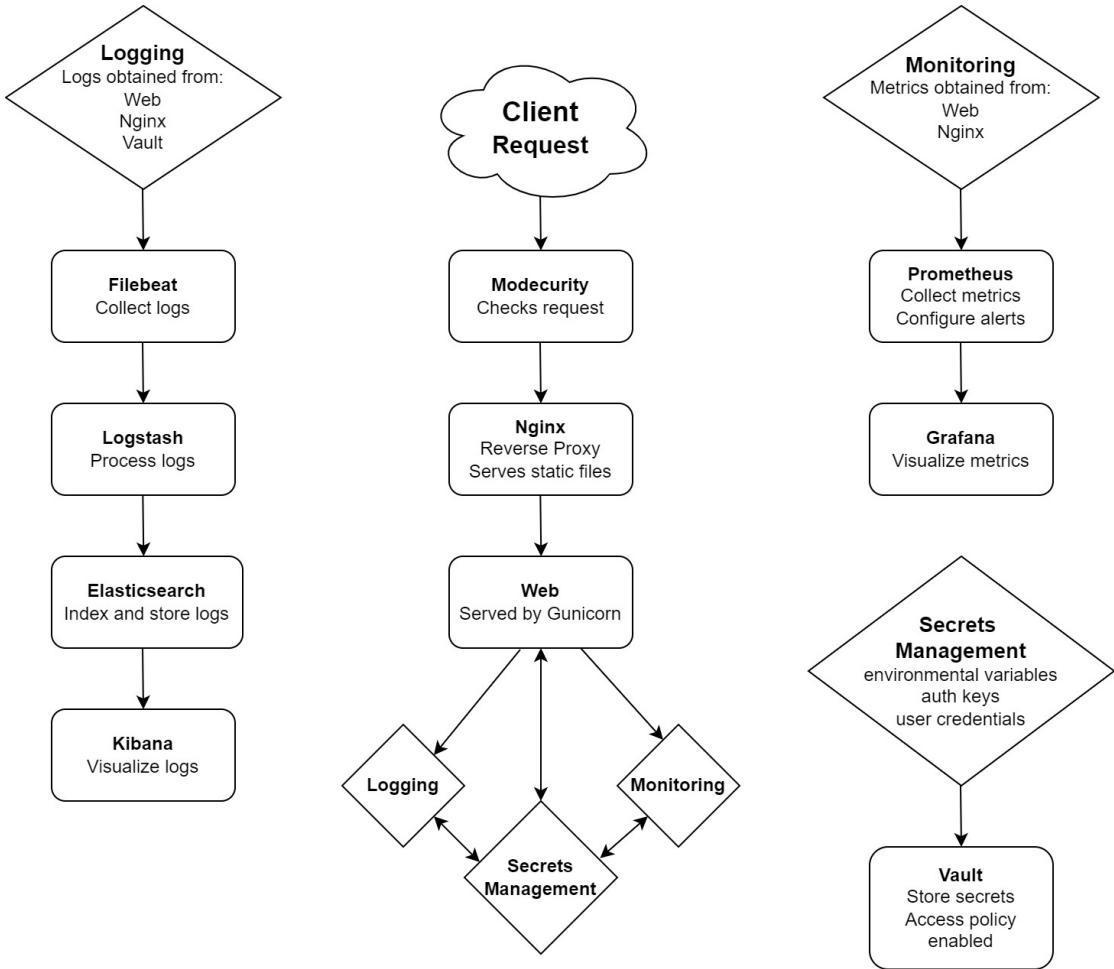


FIGURE 4.1: Deployment and DevOps Architecture

4.4 Frontend Design

The front end of the web application was developed with the following considerations:

1. The front end was developed using HTML for structure, CSS for styling, and Bootstrap for responsive design. Bootstrap's built-in components helped in quickly building a visually appealing interface while ensuring compatibility across different browsers.
2. The web application features a simple and intuitive layout. The header includes a navigation bar that allows users to access different parts of the site such as home, game, developer team and Menu. The main content area is divided into various sections depending on the user's interaction, such as viewing game statistics or updating their avatar. For details see the flowchart below;

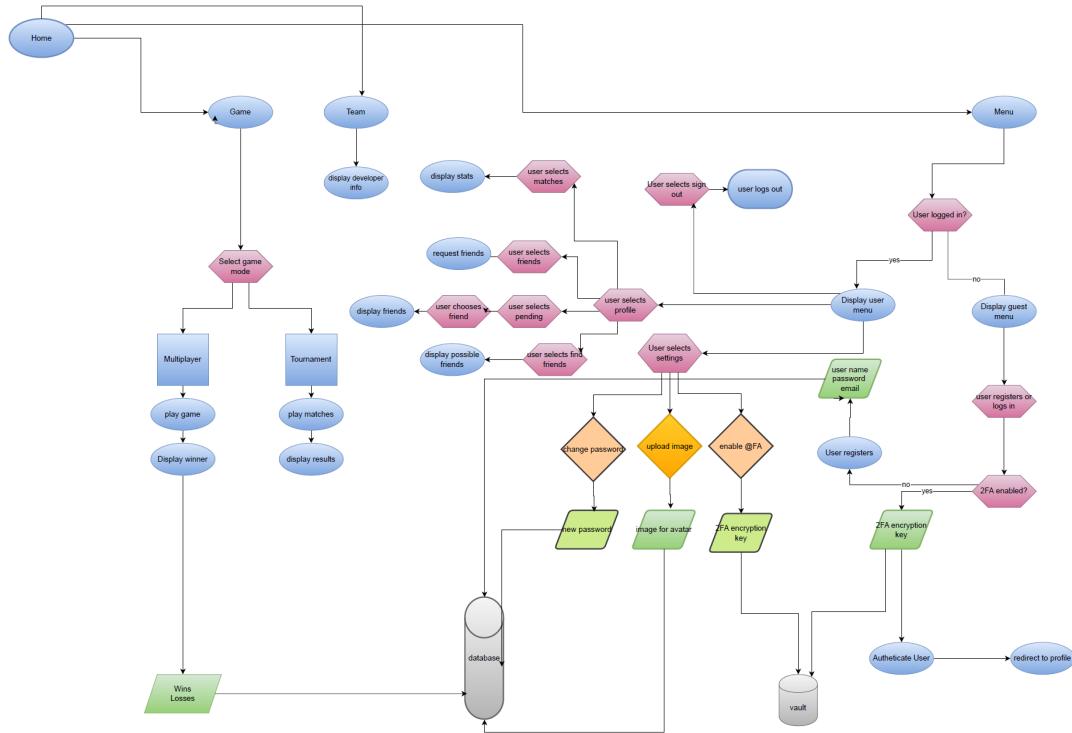


FIGURE 4.2: Flowchart

3. User experience was a key focus during the front-end design. The interface was designed to be intuitive, allowing users to easily navigate between their profile, game, and settings. Feedback mechanisms such as form validation and success/error messages were implemented to guide users during interactions like updating their profile or enabling two-factor authentication.
4. The website uses a modern, minimalist design with a focus on user-friendliness. A dark theme was chosen for the game environment, contrasting with bright colors for buttons and background. The font family used is “Arial,” known for its readability and modern look, ensuring clear text display across all devices.
5. The Pong game is seamlessly integrated into the main content area, utilizing an HTML5 <canvas> element for rendering the game environment. Players interact with the game through keyboard inputs, with on-screen prompts guiding them through the two different game modes: multiplayer or tournaments. The interface ensures a smooth transition between playing and viewing match results.
6. We have included the wireframes as representations of each application screen in Appendix A.

Chapter 5

Implementation

The testing phase is essential for validating the software's reliability, security, and overall quality. This chapter discusses the various testing methods utilized, the tools implemented, and the strategies employed to ensure the system meets both functional and non-functional requirements.

5.1 Adopting Agile for Our Development Process

Agile is a flexible, iterative software development methodology that focuses on collaboration, adaptability, and continuous feedback. Our team used the Agile methodology to maintain better organization and control over our project. We broke the project into smaller, manageable tasks or sprints, and held regular bi-weekly meetings on Google Meet to update the team on progress and address any adjustments to the project scope as challenges arose. As deadlines approached, we increased communication to several times a week to stay on track and ensure that tasks were completed on time.

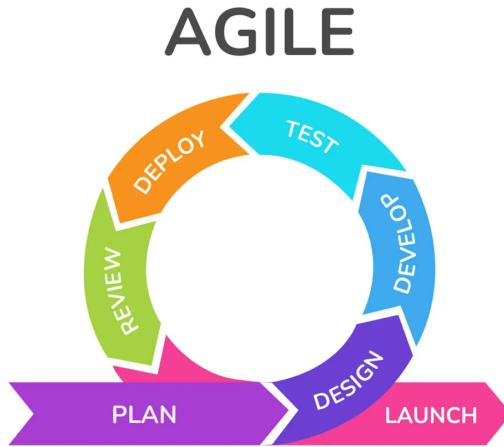


FIGURE 5.1: Agile Process Flow

5.1.1 Trello

To manage our tasks effectively, we utilized Trello as our project management tool. Each team member created task cards representing their responsibilities, which we organized into columns indicating different stages of progress, such as “To Do,” “In Progress,” and “Done.” This visual layout made it easy for us to see who was working on what and allowed us to quickly identify any bottlenecks.

5.1.2 Code Reviews, Git Workflow, Git Conflicts, and Coding Standards

In order to maintain code quality, we established GitHub ground rules for the team. At least two members were required to review and approve any code changes before merging them into the main branch. The members would thoroughly test the code prior to merging to ensure the stability of the master branch. Additionally, each person would work in a separate Git branch to avoid disrupting others’ tasks. In the event of merge conflicts, we opted to rebase our branches to keep our commit history clean and linear.

5.1.3 Gantt Chart

We developed a Gantt chart as a visual tool to map out the project timeline, highlighting key tasks, their durations, and team member responsibilities. By organizing tasks by module, we were able to maintain a clear view of each module’s progress and completion. This chart helped the team in tracking overall progress, managing deadlines,

and identifying task dependencies, ensuring the project stayed on course and met its objectives.

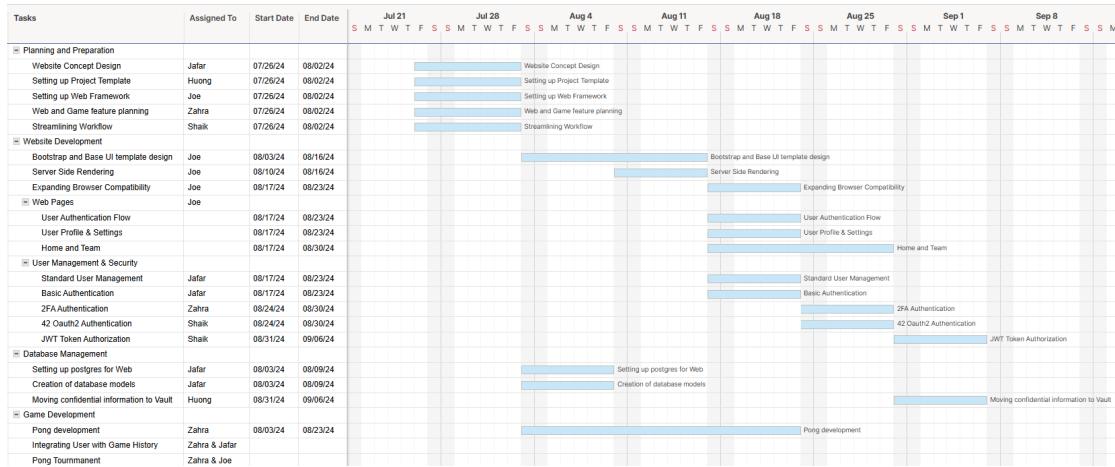


FIGURE 5.2: Gantt Chart - Part 1

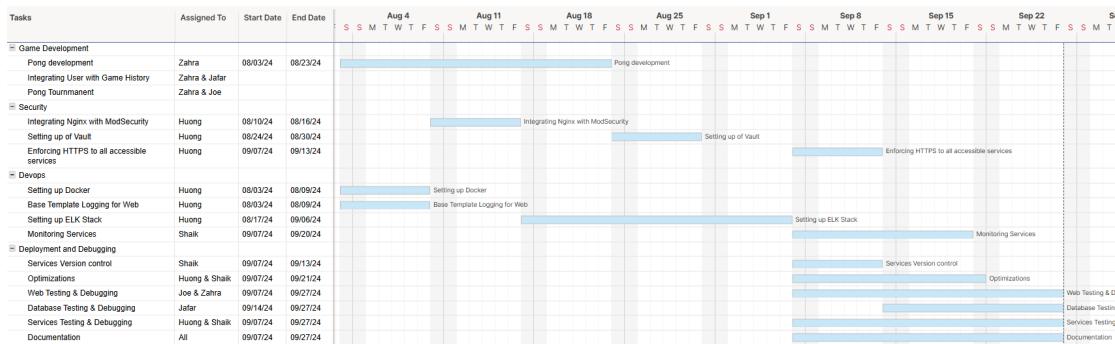


FIGURE 5.3: Gantt Chart - Part 2

5.1.4 Risk Management Register

As part of our project management process, we created a risk register and risk matrix to identify and analyze potential risks so that we could address them early on. It includes information such as the probability of each risk, the possible impact of the risk on the project, and a planned solution. By monitoring each risk on the matrix, we were able to tackle any issues as they arose and keep the project running efficiently.

		Consequence				
		Negligible 1	Minor 2	Moderate 3	Major 4	Catastrophic 5
Likelihood	5 Almost certain	Moderate 5	High 10	Extreme 15	Extreme 20	Extreme 25
	4 Likely	Moderate 4	High 8	High 12	Extreme 16	Extreme 20
	3 Possible	Low 3	Moderate 6	High 9	High 12	Extreme 15
	2 Unlikely	Low 2	Moderate 4	Moderate 6	High 8	High 10
	1 Rare	Low 1	Low 2	Low 3	Moderate 4	Moderate 5

TABLE 5.1: Risk Matrix

Risk	Impact Description	Impact Level	Probability	Severity	Mitigation	Responsibility
Risk Description	What happens if risk isn't mitigated	Rating: 1 - Low 5 - High	Rating: 1 - Low 5 - High	Impact x Probability	What to do to reduce or eliminate the risk	Who is Responsible?
Git merge conflict	when two different work on the same file conflicts may cause loss in progress or error in the code	2	5	10	Coordinate who will work on relevant files Branching Utilizing Git Rebase for easier and cleaner commit history	Shaik
Fault in Django Backend	Fatal Error during website's runtime	5	1	5	Monitoring and Alert with Prometheus Container set to always restart Vigorous Testing Recording of logs	Jafar, Joe
Fault in Database	Unable to serve services dependant on postgres including the website	5	1	5	Set to always update and migrate the database on startup Container set to always restart	Jafar
Improper website load on other browser	Website does not load appropriately on other browser	1	1	1	Make CSS to conform with common browser standards Test with commonly used browsers	Joe
Breaking changes from updates	Updates to python packages, docker images may introduce breaking changes	4	3	12	Set to use specific versions of packages and images to avoid breaking changes	Huong, Shaik
Unauthurized Access to the site	Unauthurized access to senstive pages Exposure to User data External Attack	5	2	10	Use JWT to secure sensitive pages Add 2FA to secure user account's ModSecurity to protect the control connections Encryption of User Password HTTPS only connections CSRF protection	Zahra, Huong

TABLE 5.2: Risk Register

5.2 Web Modules Implementation

Django was chosen as the backend framework for its rapid development capabilities, security features, and scalability. It follows the Model-Template-View (MTV) architecture and includes tools like an ORM, template engine, and built-in admin interface. For the frontend, Bootstrap was selected for its responsive design and ease of use in creating mobile-first applications. PostgreSQL serves as the database, offering advanced SQL capabilities and seamless integration with Django, supporting both structured and unstructured data for scalability and reliability.

5.2.1 Major: Use the Django Framework as Backend

Django is a high-level Python web framework designed for rapid development of secure and maintainable websites. It serves as a high-level backend framework designed to accelerate web development by handling many common tasks out-of-the-box, such as URL

routing, request/response handling, and authentication. It operates using the Model-View-Controller (MVC) design pattern, where the Model manages data, the View renders the user interface, and the Controller (Django's Views) handles the logic. Django's scalability, modularity, and security features like protection against CSRF and SQL injection make it ideal for developing dynamic, robust, and secure web applications. Its flexibility allows it to support a range of backend functionalities, including user management, and more.

Model-Template-View (MTV) Architecture: Django follows the MTV architecture, which is similar to the Model-View-Controller (MVC) pattern. It separates the code into three interconnected components:

- **Model:** Defines the data structure and provides an API for accessing database records.
- **Template:** Handles the presentation layer, rendering HTML based on the model data.
- **View:** Contains logic to process data and pass it to the template for rendering.

ORM (Object-Relational Mapping): Django's ORM allows interaction with databases using Python code instead of SQL. This abstraction enables developers to use multiple databases (e.g., PostgreSQL) without changing the codebase.

Admin Interface: Django auto-generates an admin interface based on model definitions, offering a customizable dashboard for database management.

URL Routing: Django's routing mechanism maps URL patterns to view functions, allowing modular and clean URL designs.

Template Engine: Django's template engine dynamically generates HTML and includes features such as template inheritance, filters, and context processors for efficient HTML management.

Forms: Django simplifies form handling with automatic rendering and validation, ensuring protection against CSRF attacks.

Authentication and Authorization: Django provides robust user authentication and authorization systems, integrated with the admin interface.

Security Features: Django includes protection against SQL injection, Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), and Clickjacking, with built-in support for SSL/HTTPS.

Why We Chose Django: Django was chosen for its rapid development capabilities, built-in security features, and scalability. Its robust community and Python's ecosystem further strengthen its appeal, ensuring smooth integration and development.

5.2.2 Minor: Use the Bootstrap Toolkit as the Front-end Framework

Bootstrap is a widely used frontend framework for designing responsive and mobile-first web applications.

Key Features:

- Responsive grid system with a flexible 12-column layout
- Pre-styled components (buttons, forms, navbars, modals)
- Sass for customization
- JavaScript plugins for interactive elements
- Built-in icon library and typography consistency

Why We Chose Bootstrap: Bootstrap enables rapid prototyping and ensures a consistent user experience across different devices and browsers, making it ideal for our project.

5.2.3 Minor: Use PostgreSQL as the Database for the Backend

PostgreSQL is an advanced, open-source relational database system known for its strong standards compliance and feature set, making it a popular choice for Django projects. It offers powerful SQL querying capabilities, including support for complex queries, indexing, and foreign keys. PostgreSQL integrates smoothly with Django's ORM, allowing developers to define data models in Python and interact with the database using high-level abstractions. It also supports JSON and full-text search, enabling both structured and unstructured data handling. This makes PostgreSQL a scalable and reliable choice for backend systems, especially those requiring high-performance data handling.

5.3 User Management Modules Implementation

The user management implementation includes standard features such as user authentication, profile management, and account settings across tournaments. OAuth 2.0 with

42 authentication is integrated, allowing users to sign up or log in using their 42 accounts, with secure token exchange and automatic account linking. This ensures a seamless and secure user experience.

5.3.1 Major: Implement Standard User Management, Authentication, and Users Across Tournaments

This module implements secure user registration, login, and profile management to ensure seamless participation in tournaments. Key functionalities include:

- Users can securely register and log in.
- Selection of a unique display name for tournament participation.
- Users can update their profile information and upload an avatar, with a default avatar if none is provided.
- Users can add friends and view their online status.
- User profiles display stats like wins and losses, with match history including 1v1 games, dates, and relevant details.

5.3.2 Major: Implement OAuth 2.0 Authentication with 42

The web server has been set up to authenticate users to their 42 Account using OAuth2. Account linking is done based on their email address and is handled automatically. Users can also sign up using their 42 accounts for seamless registration. The exchange of tokens is securely handled with Cross-Site Request Forgery protection.

5.4 Cybersecurity Modules Implementation

This section details the implementation of various cybersecurity measures within the project. It begins with an overview of major implementations, including a Web Application Firewall (WAF) using ModSecurity for enhanced protection against common web vulnerabilities and HashiCorp Vault for secure secrets management. Following this, the implementation of Two-Factor Authentication (2FA) and JSON Web Tokens (JWT) is discussed, highlighting their roles in improving user security and ensuring robust authentication processes. Each subsection provides an in-depth look at the methodologies and technologies used to fortify the application against potential threats.

5.4.1 Major: Implement WAF/ModSecurity with Hardened Configuration and HashiCorp Vault for Secrets Management

What is Modsecurity? Modsecurity is an open-source web application firewall (WAF) that provides real-time monitoring, logging, and access control for web applications. It is designed to protect web applications from a variety of attacks by filtering and monitoring traffic between a web application and the Internet.

What Can Modsecurity Do? Modsecurity can inspect incoming requests and outgoing responses to detect and block malicious activities like SQL injection, cross-site scripting, and other common web vulnerabilities. It operates on a rule-based system, where security rules define what types of traffic should be allowed or blocked. Users can create custom rules and/or use pre-defined rule sets like the OWASP Core Rule Set (CRS). It can be used with various web servers like Apache and Nginx, and protects web applications from common vulnerabilities such as:

- SQL injection
- Cross-site scripting (XSS)
- Command injection
- Path traversal attacks

Implementation: A Docker container was set up with ModSecurity integrated with the OWASP Core Rule Set (CRS) and Nginx. A custom Nginx configuration file, `transcendence.conf`, was added to the `/etc/nginx/conf.d` directory, while a custom rules file, `custom_rules.conf`, was placed in the `/etc/modsecurity.d/owasp-crs/rules` directory. Client requests are filtered by ModSecurity before being processed by the Nginx server.

What is HashiCorp Vault? HashiCorp Vault is an open-source tool designed for securely storing and managing sensitive information such as secrets, API keys, passwords, and cryptographic keys. It provides a unified interface to access secrets while ensuring that sensitive data is kept secure and only accessible to authorized users and applications.

Implementation: Two Docker containers were set up for Vault. The first container installs Vault using the official image and generates a self-signed X.509 SSL/TLS certificate, stored in a shared volume accessible by other services needing secure communication with Vault. The second container initializes Vault, generating unseal keys and a root token. It activates the KV secrets engine, creates a policy token, and stores sensitive environment variables.

5.4.2 Major: Implement Two-Factor Authentication (2FA) and JWT

Importance of Two-Factor Authentication The Django admin panel is a valuable tool for managing the backend of an application; however, it poses a risk of unauthorized access, particularly if a user's device or credentials become compromised. Two-factor authentication (2FA) enhances security and mitigates these risks, making it highly recommended.

Components of Two-Factor Authentication A standard two-factor authentication system generally includes the following components:

1. **Opt-In for 2FA:** Users should have the ability to opt into 2FA for their accounts. This enables users to set up their preferred 2FA method without needing additional support from developers or superusers.
2. **OTP Delivery Method:** Users are required to provide a one-time password (OTP) each time they attempt to log in. Common OTP delivery methods include SMS, email, or an authenticator app like Google Authenticator or Authy. Google Authenticator was chosen as it does not rely on external networks and utilizes a time-based OTP that lasts only 30 seconds, making interception by hackers more difficult. Furthermore, it does not require email addresses or phone numbers, limiting exposure in case of data leaks.
3. **User Authentication:** The system must first verify the user's identity through standard authentication (username and password) before prompting for the OTP.
4. **OTP Verification:** Once identified, the user is prompted to enter the OTP. The system then verifies the OTP and grants or denies access based on the result.
5. **User Authorization:** After setting up 2FA, users must enter a valid OTP for each login attempt.
6. **Encryption Key:** A single encryption key is generated when the system is initialized using Fernet. This encryption key is typically stored securely in an environment variable or a secrets management service (e.g., HashiCorp Vault). In this project, the key is stored in HashiCorp Vault to ensure sensitive data remains secure. The same encryption key is used by the cipher object of each user for encrypting their 2FA secret key. A cipher object, generated from the encryption key, is then used to encrypt and decrypt each user's individual two-factor key, which is utilized to generate a QR code and time-based one-time password (TOTP).

Visualization of the 2FA Process:

When a user enables 2FA, they typically scan a QR code provided by the service using an authenticator app like Google Authenticator. This QR code contains a shared secret key, stored both in the service and the authenticator app. The app then begins generating OTPs, which rotate every few seconds. When the user enters the OTP during login, the service uses the current timestamp and the shared secret key to validate the OTP, ensuring secure access to the system.

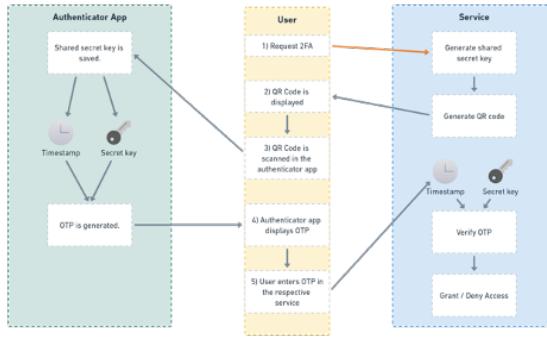


FIGURE 5.4: Authentication flow

5.4.2.1 JWT (JSON Web Tokens)

Using JWT (JSON Web Tokens) in Django, especially with the Django REST Framework (DRF), is a highly effective way to manage authentication and secure APIs. JWT offers a stateless, compact, and scalable solution that enhances performance and security for web and mobile applications.

Why JWT is a Good Idea for Django Projects:

Stateless Authentication: Unlike traditional session-based authentication, where user sessions are stored on the server, JWT allows authentication to be stateless. Once a token is issued, the server does not need to store any user information or session data, reducing memory usage and improving scalability, especially in distributed systems where multiple servers handle user requests.

Improved Scalability: In a microservices architecture or when deploying your app on multiple servers, JWT shines by eliminating the need for centralized session storage. Every request contains the token, which the server can verify independently, making it easier to scale applications horizontally without worrying about session synchronization between servers.

Efficient API Calls: JWT embeds the user's authentication data within the token itself, which is sent in the header of each HTTP request. This allows the server to process the request faster by simply decoding the token, enhancing API efficiency for client-server communication.

Enhanced Security: JWT tokens are signed and optionally encrypted, ensuring that they cannot be tampered with. Additionally, they are time-limited, automatically expiring after a specified period, which reduces the risk of token theft. JWT can also include a refresh token mechanism to renew tokens safely without requiring the user to re-authenticate frequently.

Compact Format: JWT tokens are compact, making them ideal for environments where bandwidth is limited, such as mobile applications. The small size of JWT allows for faster transmission over the network compared to session cookies or OAuth tokens.

How JWT Works in Django

1. **User Authentication:** The user sends their credentials (username and password) to a login endpoint. After validation, the server issues a JWT containing user-specific claims (such as user ID, roles, or permissions).
2. **Token Structure:** The JWT consists of three parts: a header, payload, and signature. The header contains metadata about the algorithm used, the payload includes user data (claims), and the signature ensures the token's authenticity.
3. **Access Token:** Once the token is issued, it is sent back to the client, which stores it (typically in local storage or session storage). This token is then included in the Authorization header of subsequent HTTP requests in the form `Authorization: Bearer <token>`.
4. **Token Validation:** When a request is received, the server extracts the token from the header and verifies its signature and expiration. If the token is valid, the server processes the request based on the claims included in the token.
5. **Refresh Token:** To avoid constant re-authentication, a refresh token is often issued alongside the access token. The refresh token can be used to obtain a new access token when the old one expires, keeping the session alive without exposing the user's credentials.

5.5 DevOps Modules Implementation

5.5.1 Major: Set up infrastructure for log management with ELK stack (Elasticsearch, Logstash, Kibana)

What is the ELK stack? The ELK stack is a popular set of open-source tools - Elasticsearch, Logstash, Kibana, and Filebeat - used for log and data management, search, and visualization.

- **Elasticsearch:** A distributed search and analytics engine that can index and search data. Real-time indexing of data enables fast, full-text searches and provides powerful querying capabilities, allowing users to search across a variety of fields.
- **Logstash:** A data pipeline that ingests, processes, and forwards logs or event data to Elasticsearch. It collects logs, metrics, and other data from multiple sources. It can process data in different formats, filter it, transform it, enrich it, and clean it before sending it to a system like Elasticsearch.
- **Kibana:** A visualization tool that allows users to explore, analyze, and create dashboards from data stored in Elasticsearch. It provides a graphical interface that allows users to search, view, and interact with data through dashboards, charts, maps, and other visualizations. Users can set up alerts and create scheduled reports.
- **Filebeat:** A lightweight log shipper that collects log files and forwards them to services like Logstash. It has modules that allow for easy collection and parsing of logs like Nginx and MySQL. It monitors log files, detects changes, and ships to a central storage or analysis system. It includes built-in buffering and retry mechanisms to ensure logs are delivered reliably even in the event of network outages or system failures.

Implementation While the ELK stack can be utilized for various purposes, we use it exclusively for logging in this project. We deployed individual containers for each of the four services, along with a sidecar container called elk-setup to set up the services.

Setup Using built-in tools from Elasticsearch and API requests, the elk-setup container does the following:

1. Generates SSL/TLS certificates for each ELK service using Elasticsearch's built-in certutil function. After the SSL certificates are created, it waits for the main

Elasticsearch container to start up and for the Vault container to unseal, activate the kv engine, and add the secret environment variables.

2. Retrieves the passwords for Elasticsearch, Kibana, and Logstash from Vault.
3. Activates the kibana_system user by establishing the kibana_system password. The kibana_system user is a built-in user with a fixed set of privileges that allows Kibana to connect and communicate with Elasticsearch.
4. Creates a role and user for Logstash, assigning the necessary privileges to manage the transcendence logs and oversee Index Lifecycle Management (ILM) for each log.
5. Defines an ILM policy to manage log rotation and retention, specifying when logs should roll over and when older logs can be safely deleted.

Memory Management Given that Elasticsearch is memory-intensive, we configured its memory usage with `ES_JAVA_OPTS="-Xms1g -Xmx1g"` and set `bootstrap.memory.lock` to false. While this is not a recommended practice, it was necessary due to the limited physical memory available on the 42 Mac, which required allowing memory swapping to disk. We set the minimum and maximum heap size to 1 GB to prevent the program from halting unexpectedly during operation. These memory configurations were also applied to Kibana and Logstash to ensure consistent performance across the services.

Integration All four services are configured to communicate securely via TLS/SSL connections. Filebeat collects logs from various sources, including the web application (Django), Nginx, ModSecurity, and Vault. We enabled logging in each container and created a shared volume for Filebeat to access. In the web application, loggers were set up to capture specific events such as user activity, authentication attempts, database interactions, and error tracking. Filebeat then forwards these logs to Logstash for processing and data cleansing. Logstash processes the incoming logs and sends them to Elasticsearch, where they are stored and indexed. Finally, Kibana interacts with Elasticsearch to create insights and visualizations of the log data.

The screenshot shows the Elastic Stack Management interface. The top navigation bar includes the elastic logo, a search bar, and links for Stack Management, Index Management, and Indices. A banner at the top encourages users to help improve the stack by enabling usage collection. On the left, a sidebar lists various management categories like Ingest, Data, and Alerts. The main content area is titled "Index Management" and contains tabs for Indices, Data Streams, Index Templates, Component Templates, and Enrich Policies. Below the tabs is a search bar and filters for Lifecycle status and phase. A prominent "Create index" button is visible. A table lists several Elasticsearch indices with columns for Name, Health, Status, Primaries, Replicas, Docs count, Storage size, and Data stream. The indices listed are: transcendence-logs-2024.09.28-000001, transcendence-nginx-error-logs-2024.09.28-000001, transcendence-nginx-logs-2024.09.28-000001, transcendence-vault-logs-2024.09.28-000001, and transcendence-web-logs-2024.09.28-000001.

FIGURE 5.5: Index Summary

The screenshot shows the Elastic Stack interface with the following details:

- Header:** elastic, Find apps, content, and more., New, Open, Share, Alerts, Inspect, Save.
- Left sidebar:** Discover, Help us improve the Elastic Stack, Usage collection is enabled, Dismiss, Available fields (timestamp, @version, agent.version, data_stream.dataset, data_stream.name_space, data_stream.type, ecs.version, host.name, input.type, log_source, log_file.device_id, log_file.inode, log_file.path, log_offset, tags, transcendence_message), Empty fields.
- Top navigation:** transcendence-web-logs-2024..., Filter your data using KQL syntax, Refresh.
- Main area:** Documents (39) table with columns: Document, Field statistics. The table lists 39 documents, each containing timestamp, @version, and log-source information.

FIGURE 5.6: Logs Visualization

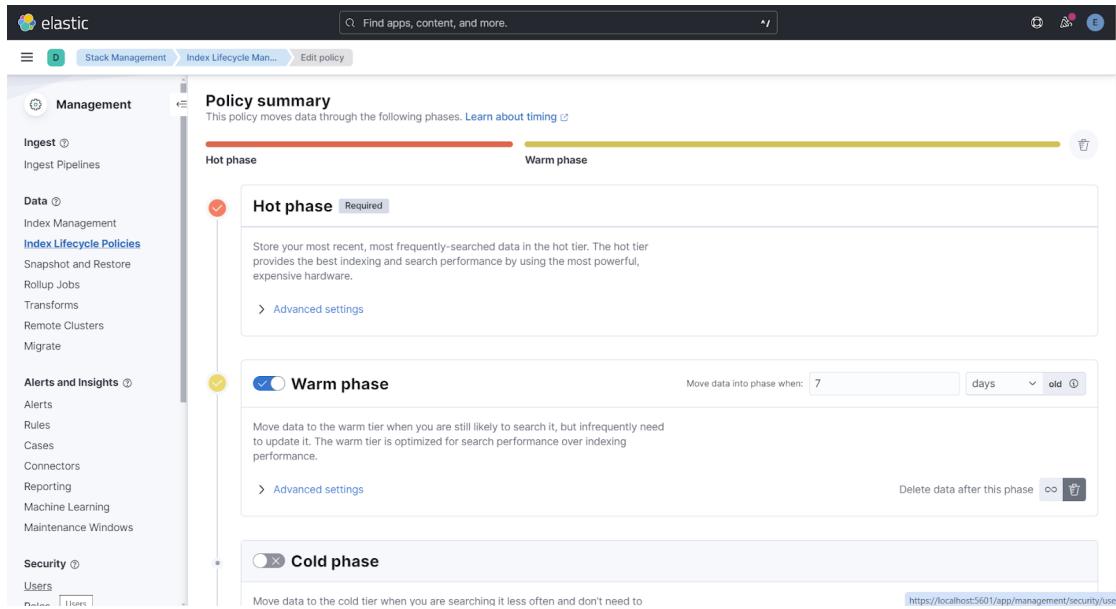


FIGURE 5.7: Policy Summary

5.5.2 Minor: Set up a monitoring system using Prometheus and Grafana

Monitoring System Setting up Prometheus to routinely check each of the services running and attain useful metrics that are helpful in monitoring the state of each service. It can help us know the activity and type of the activity for a service, thus allowing us to understand how the experience of users is on the website.

We can set up alerting services to notify us of scenarios where an error occurs in any of the services that need urgent attention. Prometheus checks the status of the website running on Django, Nginx, Filebeat, Grafana, Kibana, Logstash, PostgreSQL, and Vault.

Grafana helps visualize all the statistics collected by Prometheus. The system has been set up to allow users of different roles to access features permitted within their roles, thus securing log data.



FIGURE 5.8: Grafana Overview

5.5.3 Major: Design the backend as microservices in Docker

What is Docker and Microservices? Docker is an open-source platform that simplifies the deployment, scaling, and management of applications by packaging them into lightweight, portable containers. These containers include everything the app needs to run—code, dependencies, and environment settings—ensuring consistent behavior across any environment, from local development to production. When designing microservices in Docker, each microservice is isolated in its own container, which allows for independent development, scaling, and deployment.

Why Design as Microservices?

- Allows for independent scaling of each service depending on that service's workload. By scaling only necessary services, we reduce operational costs by using computational resources more efficiently.
- Allows for faster development. Different teams/people can work on different services in parallel.
- Fault isolation: If one part of the system fails, it won't take down the whole program.
- Different services can be built with different frameworks, programming languages, environments, and more based on their needs.
- We can more easily update or refactor services independently, making it easier to adopt new technology without rewriting the whole system.

- Easier to maintain, test, understand, and modify the codebase.
- Improve security by isolating services, limiting the attack surface, and configuring fine-grained security policies in each service.
- Simplify automated testing, deployment, and monitoring of services.

Implementation We created separate containers for the following services:

- Nginx
- PostgreSQL
- Web application (Django)
- Elasticsearch
- Elasticsearch exporter
- Kibana
- Filebeat
- Logstash
- Prometheus
- Grafana

We used a Docker Compose file to define and manage the containers. Within the Docker Compose file, we specified the services, environment variables, volumes to persist and/or share information, and network for each container.

5.5.4 Accessibility Modules Implementation

5.5.4.1 Minor: Add support for expanding Browser Compatibility

Technology Overview The Expanding Browser Compatibility module ensures that our web application performs consistently across different web browsers, enhancing accessibility and user experience. By default, Bootstrap supports a range of modern browsers.

Key Objectives

- Support for an Additional Web Browser: Ensuring that the application runs smoothly on lesser-used or newer browsers in addition to the major ones (e.g., Chrome, Firefox, Safari, Edge).

- Visual Testing: Extensive testing to detect and resolve browser-specific issues, ensuring a seamless experience across platforms.
- Addressing Compatibility Issues: Handling any discrepancies in how browsers render the application, particularly around advanced features like SSR and JavaScript-heavy interactions.
- Ensuring Consistent User Experience: Providing a uniform experience in terms of design, layout, and functionality, regardless of the user's browser.

How we Implemented This Module

- We performed visual testing on the user interface and confirmed it was similar on both Safari and Google Chrome.
- We used tools like <https://caniuse.com/> to confirm the availability of features in both Safari and Google Chrome.

5.5.4.2 Minor: Server-Side Rendering (SSR) Integration

Server-Side Rendering (SSR) is a technique used in modern web applications where the initial content of a webpage is generated and rendered on the server before being delivered to the client's browser. When a user requests a page, the server processes the required backend logic, retrieves any necessary data from databases or APIs, and sends a fully formed HTML page to the client. Unlike traditional client-side rendering (CSR), where content is dynamically loaded and rendered in the browser using JavaScript, SSR ensures that the user sees the fully rendered content almost instantly upon page load.

How SSR Works

1. **Request:** The client's browser sends a request for a specific page to the server.
2. **Server Processing:** The server runs the backend code, executes business logic, and fetches the necessary data from databases or external APIs.
3. **HTML Generation:** The server uses this data to dynamically generate a fully rendered HTML page.
4. **Response:** The server sends the fully rendered HTML page back to the client's browser.
5. **Display:** The client's browser displays the HTML content immediately, without requiring additional JavaScript processing.

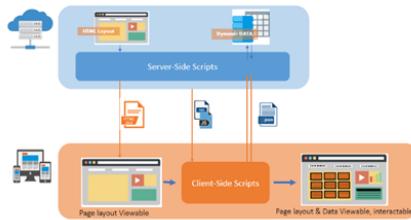


FIGURE 5.9: SSR Overview

Key Features of SSR

- **Pre-rendering Content:** The server generates the full HTML content before sending it to the client, improving perceived load times.
- **Improved SEO:** Search engines can easily crawl and index the fully rendered HTML content.
- **Enhanced Performance:** SSR reduces the client-side processing burden, particularly for content-heavy pages, improving performance on low-power devices.

How we Implemented This Module

- We used the templating functionality in Django to easily navigate through the pages.
- We utilized the Fetch API to make AJAX requests, enabling a single-page application that prevents default page reloads and retrieves raw server-rendered HTML content for seamless navigation across the website.

Chapter 6

Testing

Testing was a crucial part of the development process. Through various testing methods such as unit tests, integration tests, security tests, and performance tests, we were able to identify potential flaws in the system and rectify them before they could become issues during deployment.

We employed various test scenarios to ensure the reliability of the site.

6.1 Performance Tests

- Monitoring the time taken for each request per page.
- Monitoring resource usage of the website on both server and user sides.
- Monitoring resource usage of the game.
- Monitoring resource usage of all Docker containers.
- Testing with multiple active users.

6.2 Reliability Tests

- Ensuring the user experience with the website is intuitive.
- Ensuring the website works on multiple browsers.
- Ensuring systems can restart after a failure.
- Checking logs for errors during runtime.
- Conducting website and database tests through Postman and CURL.

6.3 Security Tests

- Ensuring users cannot access unauthorized pages.
- Ensuring the accuracy and functionality of login and signup validation processes.
- Ensuring users cannot bypass security with fake JWT tokens.
- Conducting Cross-Site Request Forgery (CSRF) tests for authentication and secured pages.
- Ensuring users cannot bypass login with false passwords or usernames.

6.4 Enacting Changes After Testing

From testing, we were able to identify and fix several issues. The following are some of the issues that were addressed:

- During development, we found that some old code was no longer functional due to changes in dependencies. Thus, we used fixed versions for Python packages.

Chapter 7

Evolution

As the project evolves, we aim to transition from a simple local game to a more secure, scalable platform. Each advancement will be designed to meet growing user needs, enhance system performance, and increase features offered to users. Some examples are given below:

7.1 Enhance Logging

7.1.1 Add Logs for Logging and Monitoring Services

To further enhance the logging capabilities of the Transcendence project, we could extend our log collection to include logs generated by the ELK stack itself, as well as Prometheus and Grafana. By capturing these logs, we would gain more insight into the health, performance, and behavior of our logging and monitoring systems, which we currently lack. With ELK stack logs, we can track issues such as node failures, index corruption, or cluster health problems in Elasticsearch, as well as any errors or misconfigurations within Logstash or Kibana. Prometheus logs would help us detect potential scraping errors, storage issues, or performance bottlenecks, while Grafana logs could be used to track dashboard rendering, plugin behavior, and user interactions.

7.1.2 Refine Logstash Filters

We can refine the filters in Logstash to further customize the information indexed into Elasticsearch, making it easier to visualize in Kibana. This will allow us to filter out unnecessary noise and focus on critical logs that provide meaningful insights, reducing storage overhead and making the logs more understandable and actionable. We could

also add metadata to our log entries, such as the application name, environment, or severity level, to facilitate categorization.

7.1.3 Automated Logging Alerts

Lastly, we could set up automated alerts based on certain log patterns and further optimize the ILM policy by creating custom policies for different types of logs.

7.2 Centralizing Secrets Management

Centralizing secrets management in Vault enhances both the security and operational efficiency of the system, reducing manual overhead while ensuring that sensitive data is tightly controlled and monitored. Using audit logs, we can monitor and track which users and services have interacted with sensitive data, ensuring compliance with security policies.

7.2.1 Auto Unsealing

To properly unseal the Vault automatically, we would need to configure a cloud service like AWS to facilitate this process.

7.2.2 Automating SSL Certificate Management

Instead of managing SSL/TLS certificates manually in each container, we can utilize Vault's PKI (Public Key Infrastructure) secrets engine to automate the generation, distribution, and renewal of certificates. This approach allows us to:

- Automate certificate rollovers. Vault can automatically generate new certificates before they expire, reducing the risk of expired certificates and manual renewal errors.
- Centralize certificate storage. Storing all certificates in Vault makes them easier to manage and allows us to control which services and users can access them.
- Dynamically issue certificates as needed. Vault can issue short-lived certificates on demand for services, further reducing the attack surface by limiting the lifespan of certificates.

7.2.3 Leverage Vault's Transit Engine

Rather than relying on Django's built-in password hashing, we could use Vault's Transit Secrets Engine for password encryption. This simplifies the application architecture, reducing complexity and minimizing the risk of security misconfigurations in Django.

7.2.4 Dynamic Secrets

We can use Vault for automatic key rotation, ensuring that encryption keys are periodically rotated without disrupting existing data. We can also generate temporary credentials for services such as PostgreSQL. Instead of relying on static credentials, Vault can issue short-lived access tokens that automatically expire after a certain period, reducing the risk of unauthorized access.

7.2.5 Customized Access Control

Currently, we only have one access policy defined, but we can improve security by implementing additional, fine-grained access policies. We can define more specific policies that restrict each service to viewing and writing only the secrets relevant to them. This approach enhances security by enforcing the principle of least privilege, ensuring that no service has access to more data than necessary. It also allows for more precise control over secret management, reducing the risk of unauthorized access or accidental exposure of sensitive information.

7.3 Take the Game Online

We could enable remote playing by taking the game online and including features like live chat to develop a community. Adding a ranking system would allow us to base matchmaking on skills, making it more challenging. Taking the game online would enable users to play from different devices, so we would have to address issues like lag and unexpected disconnections.

7.4 Allow for Bigger Tournaments

Expanding the number of users that can participate in a tournament will make the game more exciting and engaging. We could accommodate 8 to 16 users and implement

a system to notify players when their match is scheduled. By linking it to streaming services like YouTube, we could attract spectators and boost engagement.

7.5 Add Another Game

We could continue with our plan to add Tic Tac Toe as another game on our platform. This game would utilize the same user management, authentication, and matchmaking systems but diversify users based on their interests. We could later gather user feedback before adding additional games.

The evolution of this project is focused on expanding both its technical capabilities and user engagement. By improving security through Vault, enhancing monitoring with ELK, and taking the game online, the project can scale to handle more players and complex game mechanics. As tournaments grow larger and new games are added, the platform will become a comprehensive, competitive gaming environment with a global reach.

Conclusion

This project successfully met the primary objective of creating a full-stack web application featuring a multiplayer Pong game while integrating modern technologies for enhanced performance, security, and scalability. By leveraging Django, Bootstrap, PostgreSQL, and advanced logging and monitoring tools like ELK and Prometheus, we ensured that the application was built with both functionality and future growth in mind.

The journey was not without its challenges. Implementing two-factor authentication (2FA) and managing sensitive data with HashiCorp Vault presented initial hurdles but ultimately resulted in a much more secure application. This experience has underscored the importance of security in software design and the benefits of using industry-standard tools.

Looking ahead, there are several exciting opportunities to further evolve this project. Expanding the game's features to allow for larger online tournaments and introducing new games will make the platform more engaging. Additionally, enhancing Vault integration and further refining performance monitoring will help maintain security and responsiveness as the application scales.

This project has been an invaluable learning experience, combining theoretical knowledge with practical application and providing a strong foundation for future professional development.

References

Google. (n.d.). *SEO starter guide*. Available at:

<https://developers.google.com/search/docs/fundamentals/seo-starter-guide>.

YouTube. (n.d.). *Video tutorial*. Available at:

<https://youtu.be/5ws4RDEw91Q?si=g6rM1JP3SzfTS6B3>.

YouTube. (n.d.). *Another tutorial*. Available at:

<https://www.youtube.com/watch?v=KKu0n6OW0uk>.

YouTube. (n.d.). *Further explanation*. Available at:

<https://www.youtube.com/watch?v=nl0KXCa5pJk>.

Atlassian. (n.d.). *What is agile?*. Available at:

<https://www.atlassian.com/agile>.

Django. (n.d.). *Django framework - Python Social Auth documentation*. Available at:

<https://python-social-auth.readthedocs.io/en/latest/configuration/django.html>.

Elastic. (n.d.). *Elasticsearch Guide*. Available at:

<https://www.elastic.co/guide/en/elasticsearch/reference/8.14/index.html>.

freeCodeCamp. (2023). *Server side rendering in JavaScript – SSR vs CSR explained*.

Available at:

<https://www.freecodecamp.org/news/server-side-rendering-javascript/>.

GeeksforGeeks. (2024). *What is Single Page Application?*. Available at:

<https://www.geeksforgeeks.org/what-is-single-page-application/>.

Otto, M., & Mark, J.T. (n.d.). *Get started with Bootstrap*. Available at:

<https://getbootstrap.com/docs/5.3/getting-started/introduction/>.

Simple JWT. (n.d.). *Simple JWT documentation*. Available at:

<https://django-rest-framework-simplejwt.readthedocs.io/en/latest/>.

HashiCorp. (n.d.). *Vault: Tutorials — HashiCorp Developer*. Available at: <https://developer.hashicorp.com/vault/tutorials>.

Django Project. (n.d.). *Using the Django authentication system*. Available at: <https://docs.djangoproject.com/en/5.1/topics/auth/default/>.

Can I Use. (n.d.). ‘Position: sticky’: *Can I use... support tables for HTML5, CSS3, etc..* Available at: <https://caniuse.com/>.

Appendix A

Wireframes and User Journey

Wireframes serve as a visual representation of the foundational structure for the project's user interface (UI) and are integral to illustrating the user journey. They allow early visualization of layout and user interactions, providing clarity on how users will navigate the system, well before the detailed design and coding stages.

Wireframes emphasize:

- **Layout Structure:** Mapping out the arrangement of elements on each screen, including headers, navigation menus, content areas, and footers, ensuring a cohesive design.
- **Content Prioritization:** Organizing content based on importance to guide users seamlessly through their journey, ensuring an intuitive and efficient experience.
- **Interaction Points:** Highlighting interactive elements such as buttons, links, and form fields, offering a clear understanding of how users will engage with the system.

In this project, wireframes, combined with key screenshots from our website, visually narrate the user journey. They provided a blueprint that guided the development team in transforming design concepts into a functional interface, ensuring alignment with user needs and project goals.

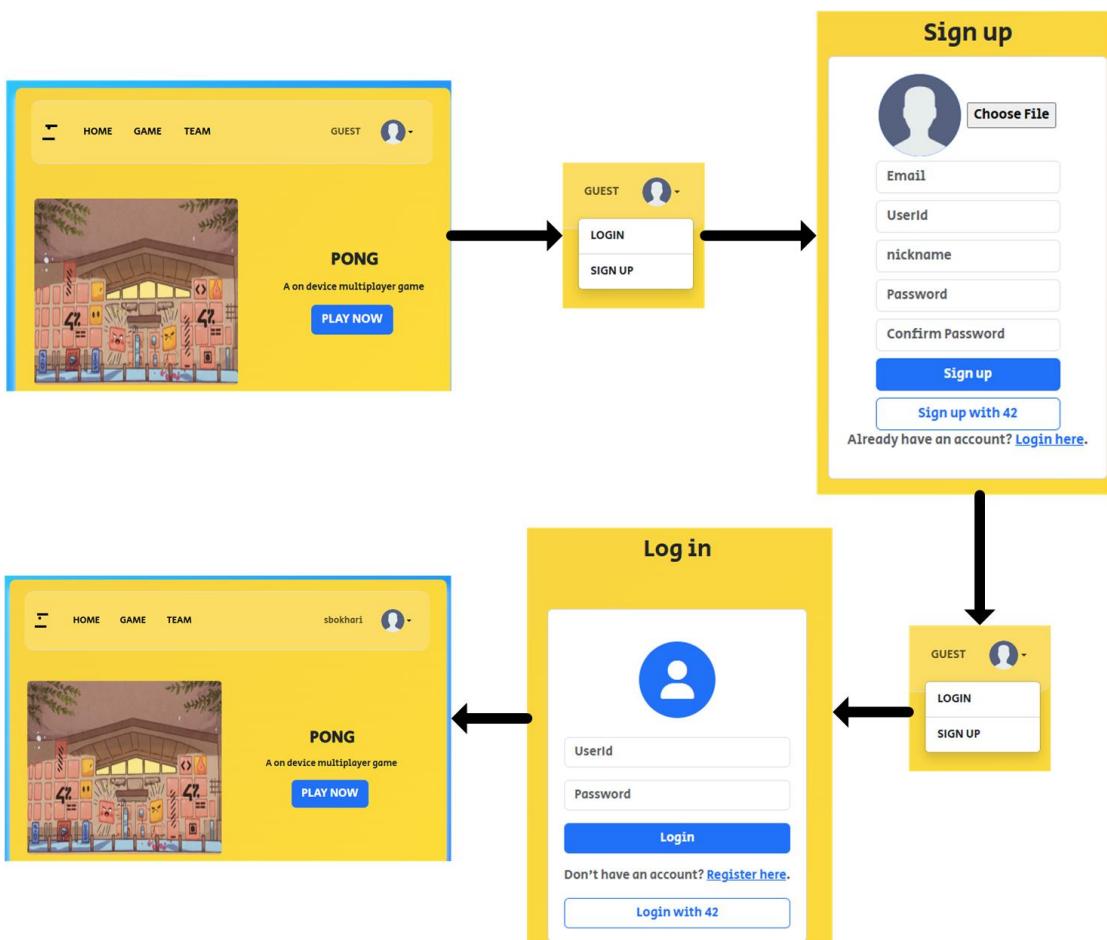


FIGURE A.1: Signup & Login

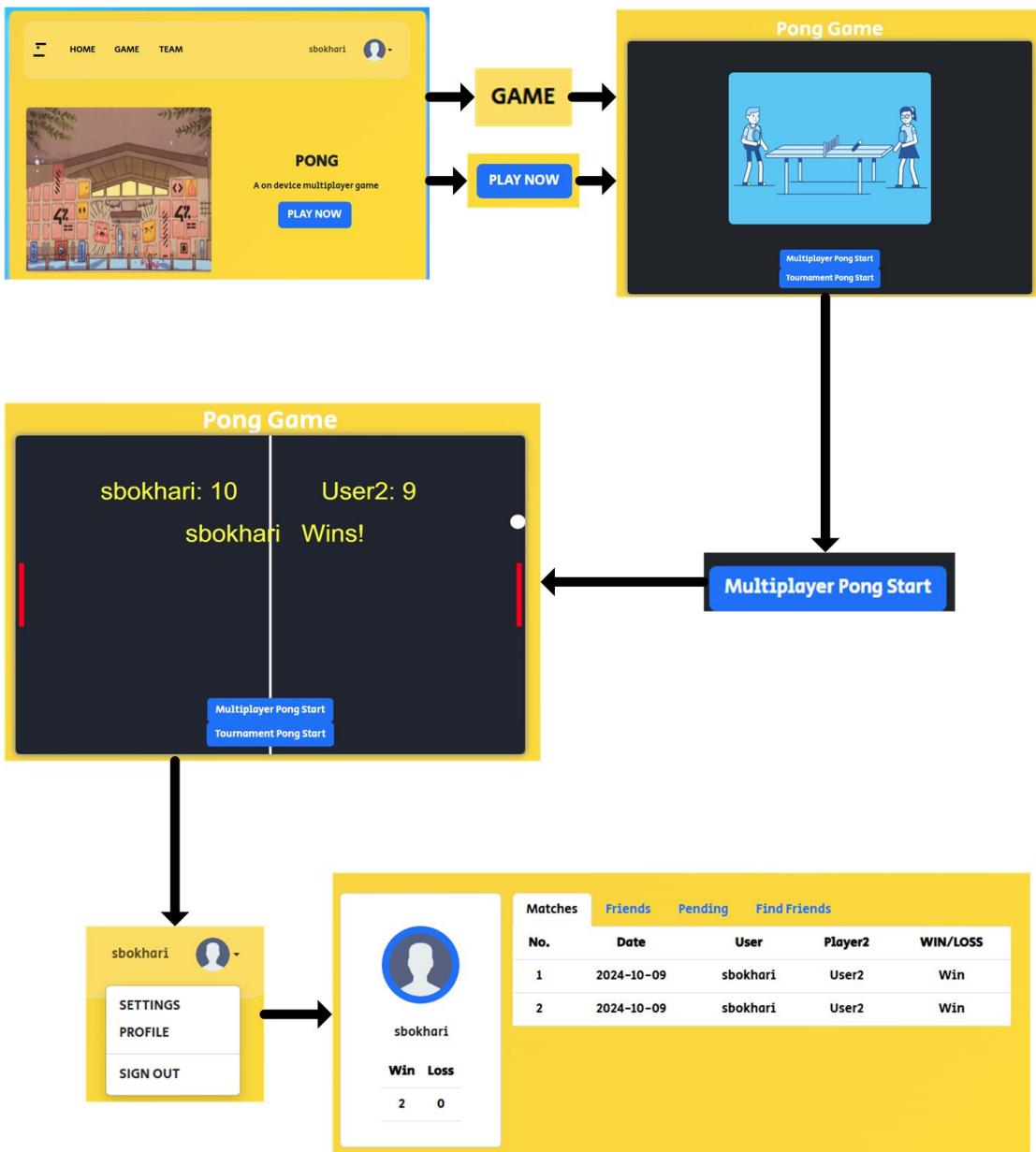


FIGURE A.2: Game & History

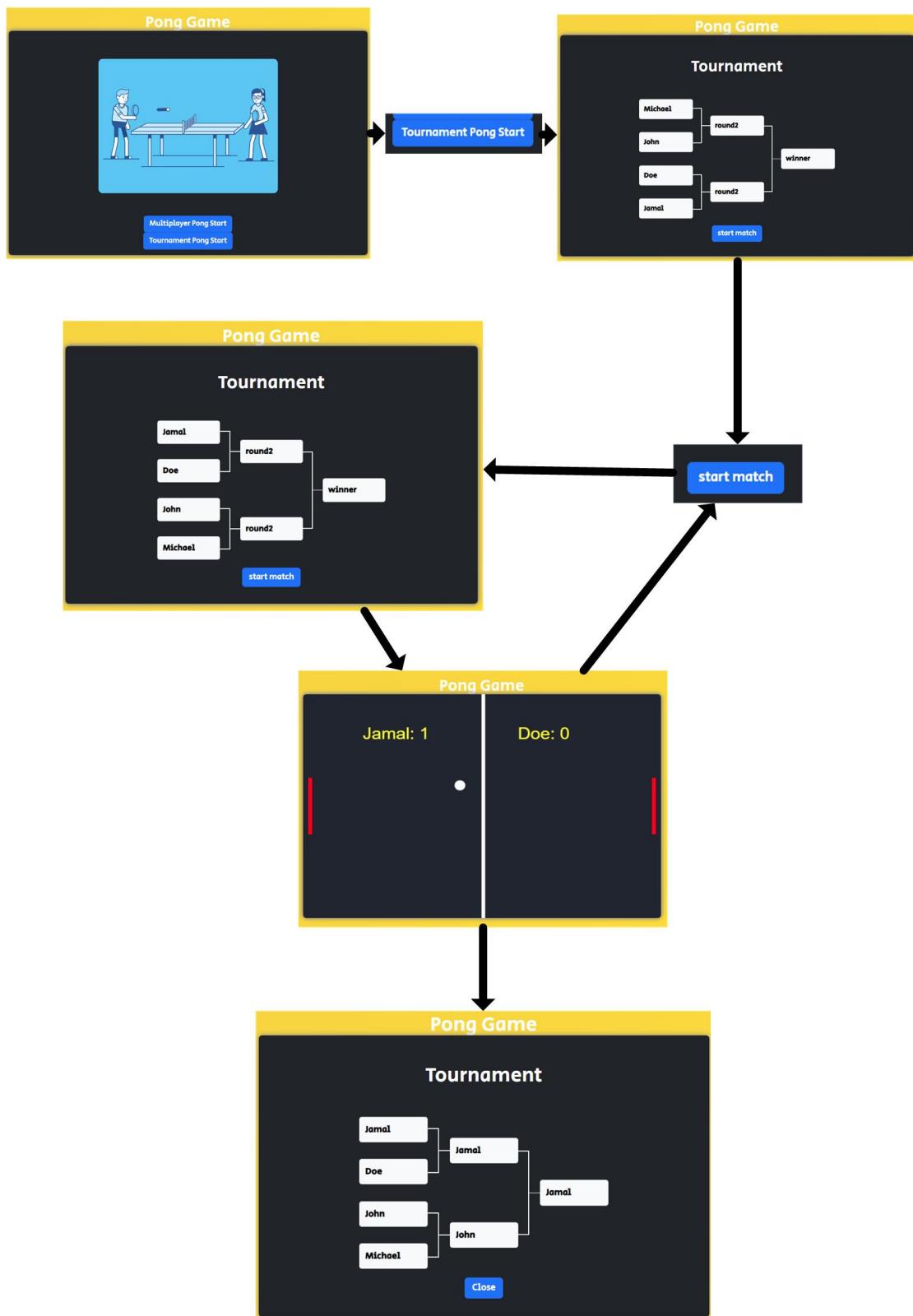


FIGURE A.3: Tournament



FIGURE A.4: Enabling 2FA

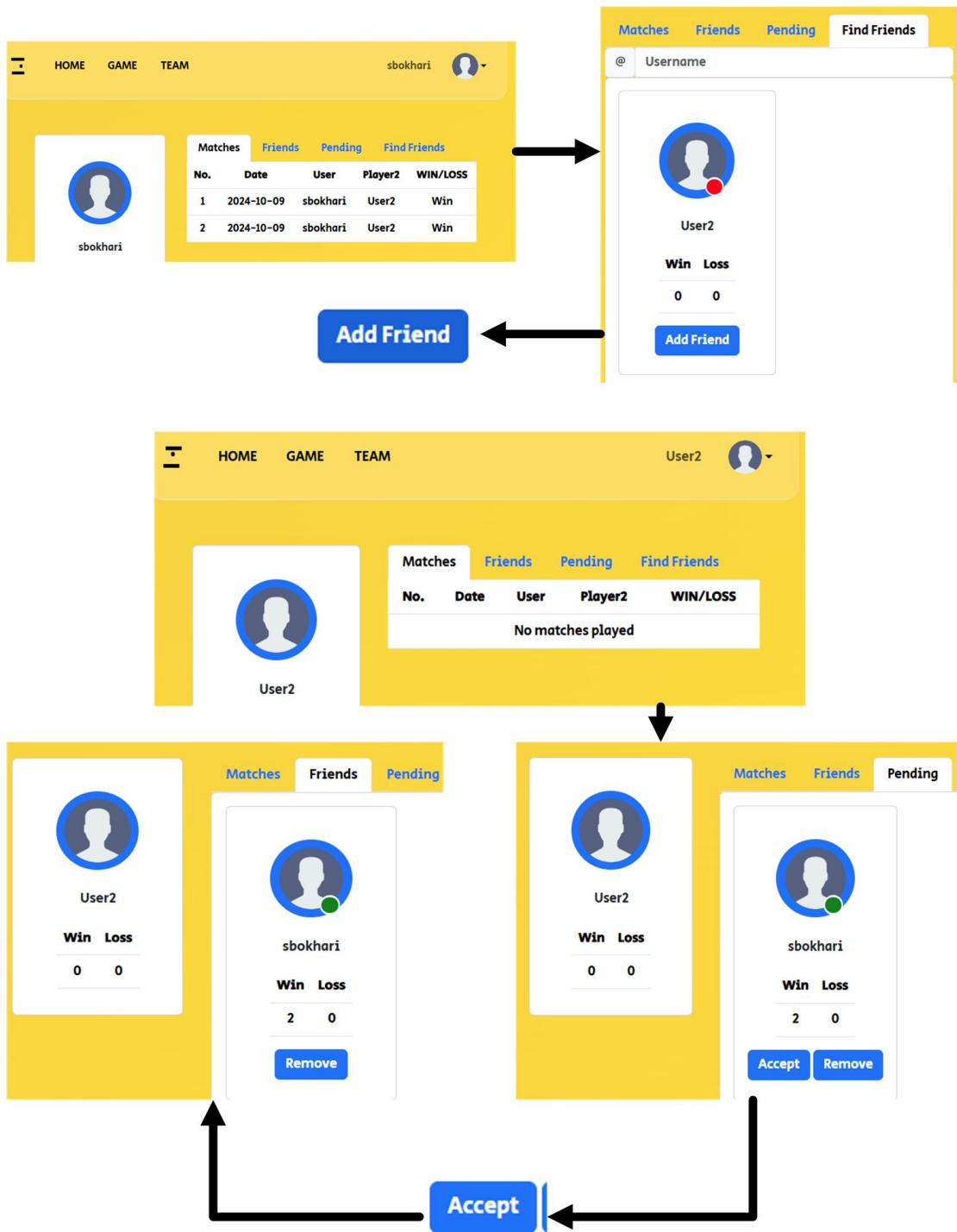


FIGURE A.5: Adding Friends

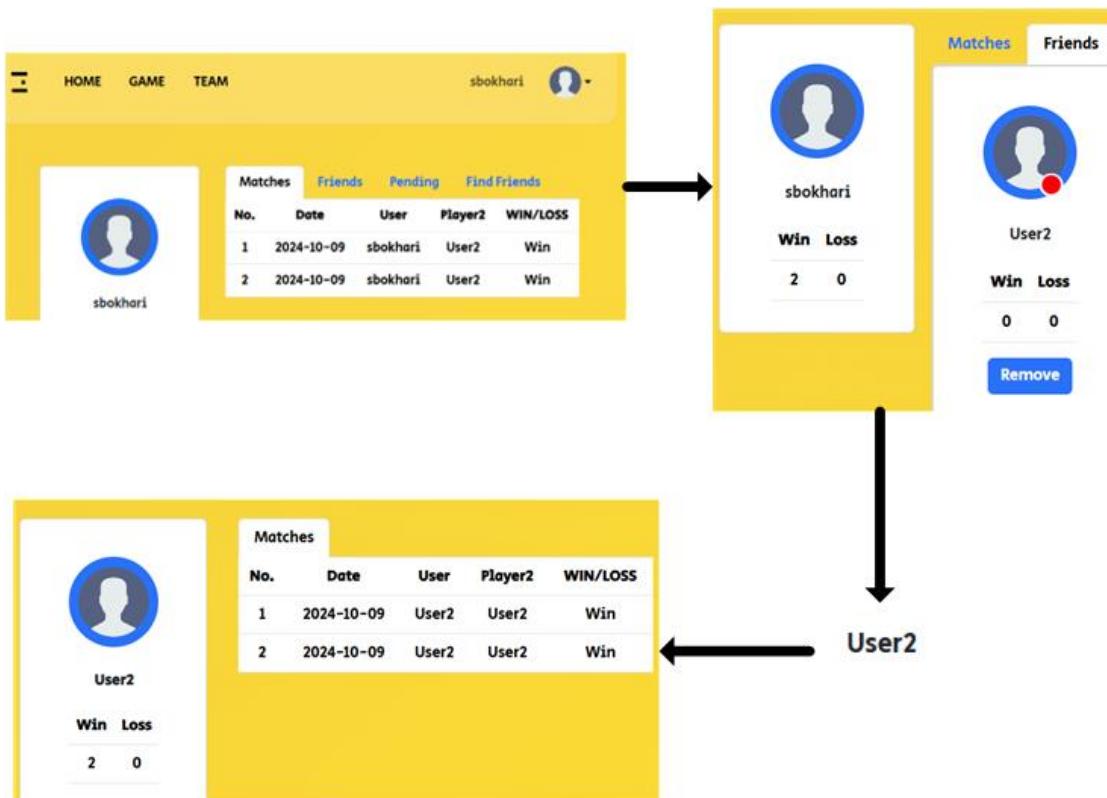


FIGURE A.6: View friend Profile