SOFTWARE REQUIREMENTS SPECIFICATION

for

Experis Academy Case: HvZ

Version 3.1
Winter 2022

By Greg Linklater

Noroff Accelerate AS

March 2022

Contents

1. Revision History

Name	Date	Reason For Changes	Version
Initial Version	02.04.2019	Created Document	1.0
Autumn 2019	05.10.2019	Specification Revision	2.0
Autumn 2019	31.10.2019	Revision for Växjö cohort	2.1
Winter 2020	03.03.2020	Revision	3.0
Winter 2022	04.03.2022	Update dates	3.1

2. Introduction

2.1. Purpose

This SRS (Software Requirements Specification) describes the software product to be created by one of the candidate groups in the *Experis Academy Remote* contingent of the *Fullstack Development* short course.

2.2. Document Conventions

For the purposes of this document the following definitions shall apply:

- The *course* refers to the *Fullstack Development* short course as currently offered at *Experis Academy Remote*; specifically the Winter 2022 intake of said course.
- A *candidate* refers to an individual currently enrolled in the course. The plural *candidates* refers to the candidates as they are arranged in groups of four individuals for this case.
- The *software* refers to the final software product to be created.
- *Mentors* refer to the industry experts giving their time to assist the candidates during the case period.

2.3. Intended Audience and Reading Suggestions

This document is intended for use by the mentors and the candidates. Mentors and candidates should read this document and familiarize themselves with the specifications of the software to be created.

2.4. Project Scope

The primary purpose of the software to be a capstone experience for the candidates; they are expected to utilize a selection of possible development options to produce a single software solution that demonstrates their capabilities as developers. The candidates must produce a software solution that is considered a final product. The software is to be produced over a period of three weeks.

2.5. Delivery

A deployed version of the software must completed by the morning of the 31^{st} of March or 1^{st} of April 2022. Nearing the completion of the software, candidates will be required to present their solutions twice. A mock presentation will take place on the 29^{th} of March or 30^{th} of March 2022, and the final presentation will take place on the 31^{st} of March or 1^{st} of April 2022.

2.6. References

Candidates should find the following documents on the Noroff Learning Management System[1] (*Moodle*) site:

- Group membership lists with mentors assigned.
- A copy of this document.

[1] `https://lms.noroff.no/`

3. Requirements Specifications

Humans vs. Zombies (HvZ) is a game of tag played at schools, camps, neighborhoods, libraries, and conventions around the world. The game simulates the exponential spread of a fictional zombie infection through a population. The game is played as follows:

- The game begins with one or more "Original Zombies" (OZ) or **patient zero**. The purpose of the OZ is to **infect** human players by tagging them. Once tagged, a human becomes a zombie for the remainder of the game.
- Human players are able to defend themselves against the zombie horde using Nerf weapons and *clean*, rolled-up socks which may be thrown to **stun** an unsuspecting zombie.
- Many variants of the game rules exist which introduce additional rules and activities into the game; for example, the Rhodes University variant introduces a web-based game map where markers appear showing the location of missions and supplies. The appearance of supplies forces humans to leave their **safe zones** and risk being turned.

While HvZ is a highly physical and active game, the administrators would have difficulty communicating with all of the players and keeping track of the game and individual player states without help. For this case, you are tasked with designing and implementing such a system that will enable players to manage their own state in accordance with the rules of the game and leave the administrators free to focus on improving the game itself.

3.1. Product Perspective

For this case, candidates are expected to design and implement a software solution for managing the state and communication of one or more concurrent games of HvZ. The main components of the system are as follows:

- A static, single-page, front-end using a modern web framework.
- A RESTful API service, through which the front end may interact with the database.
- A suitable database.

3.2. Front-end Requirements

3.2.1. FE-01: Landing Page / Game Master

A landing page with a list of currently available games.

- Each game listed should display the title, the current game state, number of registered players and relevant dates.
- The game list should be visible even to anonymous users, however the option to visit the game detail page (by clicking one of the options in the game list) should only be available to active, authenticated users.
- Users must be able to authenticate (or register) themselves. This need not be overly fancy, however standard security practices should be followed. Candidates could delegate authentication to a federated identity provider service, such as Google, however this is optional.

3.2.2. FE-02: Game Detail

Upon selection, a distinct view where details of the selected game are shown should be displayed to the user. The game detail screen should display a number of view fragments that are conditional on the player's state within that game. The fragments are described below:

- **Title Fragment**. The name, description and rules of the game should be displayed to any user. The name and description should vary between games but the rules should be constant.
- **Registration Fragment**. A simple button that allows a user to register as a player in this game.
- **Map Fragment**. An interactive map of the game area that may contain **gravestones** and **mission markers** from time to time. The visibility of individual markers may vary based on the player state.
- **Bite Code Fragment**. When tagged, human players are required to provide a unique, secret bite code to the zombie. The bite codes should be randomly generated and appropriate for manual text entry.
- **Bite Code Entry**. Zombies that collect the bite code of a human must log their kill in the system to turn the human player into a zombie. Optionally the killer may specify GPS coordinates and a text description of their kill to create a gravestone marker on the map.
- **Squad Registration Fragment**. The squad registration fragment should display a small form to create a new squad and a list of available squads to be joined. Each entry in the list should display thier name, respective total number of members, number of "deceased" members and a button to join the squad.

		Player State			
		[§] *Unregistered*	*Administrator*	[†] *No Squad*	*Squad Member*
Fragment	*Title*	✓	✓	✓	✓
	Registration	✓			
	Mission Markers		✓	✓[‡]	✓[‡]
	Squad Check-in Markers		✓		✓
	Gravestones	✓	✓	✓	✓
	Bite Code			✓(H)	✓(H)
	Bite Code Entry			✓(Z)	✓(Z)
	Squad List	✓	✓	✓	✓
	Squad Creation			✓	
	Squad Detail				✓[‡]
	Global Chat		✓	✓	✓
	Faction Chat		✓	✓[‡]	✓[‡]
	Squad Chat		✓		✓[‡]

[§] Authenticated but not registered for the current game.
[†] Player (faction agnostic) registered for the game but not for any squad.
[‡] Per faction (i.e. Human players are shown human features).

Table 3.1.: Game Detail Fragment Visibility Matrix

- **Squad Details Fragment**. This should display the names, relative ranks and state of each of the members of your squad. Additionally there should be buttons to leave a check in marker on your current position and to leave the squad.
- **Chat Fragment**. A simple tabbed, factional chat display with a text field for sending a new message and a button to send. Pressing the enter key while in the text field should have the same behaviour as pressing the send button.

The visibility of each of these fragments by game and player state is described in Table 3.1.

3.2.3. FE-03: Administrator View

Game administrators should have an appropriate interface to do the following:

- Create and edit a game.
- Edit any individual player's state.
- Create and edit mission markers.

3.2.4. FE-04: Desktop and Mobile Compatible

Using modern design concepts/frameworks, the system should be function on both desktop computers and mobile devices.

3.3. API Requirements

The API forms the core of this project.
The API must safely deal with all the possible errors that can occur during the processing of an order/payment.

3.3.1. API-01: No query parameters

The API must not use query parameters at all. End points should use either `Headers` or `Body` (application/json). The only exception to this is that query parameters may be used to provide input to a search operation.

3.3.2. API-02: Game

`GET /game`
Returns a list of games. Optionally accepts appropriate query parameters.
`GET /game/<game_id>`
Returns a specific game object.
`POST /game`
Creates a new game. Accepts appropriate parameters in the request body as `application/json`. *Admin only.*
`PUT /game/<game_id>`
Updates a game. Accepts appropriate parameters in the request body as `application/json`. *Admin only.*
`DELETE /game/<game_id>`
Deletes (cascading) a game. *Admin only.*
`GET /game/<game_id>/chat`
Returns a list of chat messages. Optionally accepts appropriate query parameters. The messages returned should take into account the current game state of the player, i.e. a human should recieve chat messages addressed to the "global" (cross-faction chat) and "human" chats but not the "zombie" chat.
`POST /game/<game_id>/chat`
Send a new chat message. Accepts appropriate parameters in the request body as `application/json`.

3.3.3. API-03: Game States

A game may only have one of the following states:

- **Registration**. The game has not begun. Users may register themselves as players of the game.
- **In Progress**. The game has begun. New self-registrations are not being accepted.
- **Complete**. The game is complete. No further write actions on the game object are permitted.

3.3.4. API-04: Player

`GET /game/<game_id>/player`

Get a list of players. Player details should be not expose sensitive information such as the player's `is_patient_zero` flag when being accessed by a non-administrator. Optionally accepts appropriate query parameters.

`GET /game/<game_id>/player/<player_id>`

Returns a specific player object. Player details should be not expose sensitive information such as the player's `is_patient_zero` flag when being accessed by a non-administrator.

`POST /game/<game_id>/player`

Registers a user for a game. Player objects must be unique per user, per game. Only administrators may specify appropriate parameters in the request body as `application/json` – otherwise all parameters assume default values.

`PUT /game/<game_id>/player/<player_id>`

Updates a player object. Accepts appropriate parameters in the request body as `application/json`. *Admin only.*

`DELETE /game/<game_id>/player/<player_id>`

Deletes (cascading) a player. *Admin only.*

3.3.5. API-05: Kill

`GET /game/<game_id>/kill`

Get a list of kills. Optionally accepts appropriate query parameters.

`GET /game/<game_id>/kill/<kill_id>`

Returns a specific kill object.

`POST /game/<game_id>/kill`

Creates a kill object by looking up the victim by the specified bite code. Accepts appropriate parameters in the request body as `application/json`. Returns `401 Bad Request` if the user is not an administrator and the specified bite code is invalid.

`PUT /game/<game_id>/kill/<kill_id>`

Updates a kill object. Accepts appropriate parameters in the request body as `application/json`. *Only the killer or an administrator may update a kill object.*

`DELETE /game/<game_id>/kill/<kill_id>`

Delete a kill. *Admin only.*

3.3.6. API-06: Squad

`GET /game/<game_id>/squad`

Get a list of squads. Optionally accepts appropriate query parameters.

`GET /game/<game_id>/squad/<squad_id>`

Returns a specific squad object.

`POST /game/<game_id>/squad`

Creates a squad object. Accepts appropriate parameters in the request body as `application/json`. Should also automatically create a corresponding squad member object that registers the player as the ranking member of the squad they just created.

`POST /game/<game_id>/squad/<squad_id>/join`

Creates a squad member object. Accepts appropriate parameters in the request body as `application/json`.

`PUT /game/<game_id>/squad/<squad_id>`

Updates a squad object. Accepts appropriate parameters in the request body as `application/json`. `Admin only.`

`DELETE /game/<game_id>/squad/<squad_id>`

Delete a squad. *Admin only.*

`GET /game/<game_id>/squad/<squad_id>/chat`

Returns a list of chat messages. Optionally accepts appropriate query parameters. The messages returned should take into account the current game state of the player, i.e. a human should recieve chat messages addressed to the "global" (cross-faction chat) and "human" chats but not the "zombie" chat.

`POST /game/<game_id>/squad/<squad_id>/chat`

Send a new chat message addressed to a particular squad. Accepts appropriate parameters in the request body as `application/json`. Only administrators and members of a squad who are still in the appropriate faction may send messages to the squad chat, i.e. a human who has died should not be able to continue sending messages to their human squad; in this event it returns `403 Forbidden`.

`GET /game/<game_id>/squad/<squad_id>/check-in`

Get a list of squad check-in markers. Optionally accepts appropriate query parameters. Only administrators and members of a squad who are still in the appropriate faction may see squad check-ins, i.e. a human who has died should not be able to access the check-ins of their human squad; in this event it returns `403 Forbidden`.

`POST /game/<game_id>/squad/<squad_id>/check-in`

Create a squad checkin. Accepts appropriate parameters in the request body as `application/json`. Only members of a squad who are still in the appropriate faction may check-in with their squad, i.e. a human who has died should not be able to access the check-ins of their human squad; in this event it returns `403 Forbidden`.

3.3.7. API-07: Mission

`GET /game/<game_id>/mission`

Get a list of missions. Optionally accepts appropriate query parameters. This should only return missions that are faction appropriate.

`GET /game/<game_id>/mission/<mission_id>`

Returns a specific mission object. Returns `403 Forbidden` if a human requests a zombie mission and vise versa.

`POST /game/<game_id>/mission`

Creates a mission object. Accepts appropriate parameters in the request body as `application/json`. *Admin only.*

`PUT /game/<game_id>/mission/<mission_id>`

Updates a mission object. Accepts appropriate parameters in the request body as `application/json`. `Admin only.`

`DELETE /game/<game_id>/mission/<mission_id>`

Delete a mission. *Admin only.*

3.4. Database Requirements

3.4.1. DB-01: Database Design

While an ERD is supplied in Appendix A, it is incomplete and is provided to the candidates as a suggestion. Candidates must alter, or design from scratch, and document a database schema for use by this solution.

4. Other Considerations & Requirements

4.1. Security Requirements

4.1.1. SEC-01: User Authentication

Users should be authenticated appropriately before being allowed to interact with the system. All API endpoints (unless otherwise marked) should require an appropriate bearer token to be supplied in the `Authorization` header of the request. 2FA should be enforced[1].

It is recommended to use an external authentication provider (such as Microsoft, Google, or Gitlab) and the *OpenID Connect Auth Code Flow* [2]. If desired, a self-hosted identity provider can be quickly deployed using Docker[3] and Keycloak[4].

Failed authentication attempts should prompt a `401 Unauthorized` response. Authentication related endpoints should also be subject to a rate limiting policy where, if authentication is attempted too many times (unsuccessfully), then requests from the corresponding address should be temporarily ignored. Candidates should decide on an appropriate threshold for rate limiting.

4.1.2. SEC-02: Input Sanitation

All endpoints that accept input from users must ensure that the provided data is appropriately sanitized or escaped so that it cannot be used in an XSS or injection attack.

4.1.3. SEC-03: Credential Storage

Care should be taken to ensure that administrative credentials (i.e. database credentials) are not hard coded into the application and are instead provided to the application using environment variables.

4.1.4. SEC-04: HTTPS

The final, public facing deployment of the application should enforce communication only over HTTPS.

[1] `https://authy.com/what-is-2fa/`
[2] `https://openid.net/specs/openid-connect-core-1_0.html`
[3] `https://hub.docker.com/r/jboss/keycloak/`
[4] `https://www.keycloak.org/`

4.2. Performance Requirements

4.2.1. PRF-01: Loading Time

The system is expected to be responsive at all times and not 'hang' or take excessive time to perform actions.

4.2.2. PRF-02: Error Messages

In the event of an error case, the user should be shown a meaningful error message describing what they have done wrong without being confusing. While error messages should be comprehensive, care should be taken not to expose sensitive information that could compromise the security of the application.

4.3. Documentation Requirements

4.3.1. DOC-01: User Manual

Candidates should submit a user manual containing instructions for how to perform each action within the system with accompanying screenshots.

4.3.2. DOC-02: API Documentation

Candidates should submit a detailed listing of each API endpoint they create with the following information:

1. Endpoint method.
2. Endpoint path.
3. Required and accepted headers.
4. Accepted parameters
5. Expected changes to the data.
6. Possible responses and their meanings.
7. Possible error cases with explanations.

4.3.3. DOC-03: README.md

Candidates should provide a `README` file with the following information:

1. A name for the project
2. A list of team members and project participants
3. Detailed installation instructions to run the service from scratch.

The previously mentioned API documentation ***MAY*** also be included in the `README` as opposed to being a separate document.

The user manual ***MAY NOT*** be included as part of the `README` and must be a separate document.

4.4. Other / Miscellaneous Requirements

4.4.1. MISC-01: Deliverables

The candidates are expected to deliver the following artifacts:

1. A `Git` repository containing all code and inline documentation.
2. Any and all project documentation, including plans, database schema and instructional material (such as the user manual).
3. A live, functional deployment of the completed product.
4. A presentation of their product.

4.4.2. MISC-02: Presentation

Candidates must produce a 30 minute presentation which introduces the team, and discuss their product in two parts:
Development (10 min). Candidates must discuss the choices made during development, the difficulties they faced and how they overcame them.
Feature Demonstration (10 min). Candidates must produce a brief overview of the functionality of their final product.
Questions (10 min). Sufficient time should remain for the audience to ask the candidates questions. Candidates should be prepared to defend decisions made throught the development process to the satisfaction of those present.
Candidates should create a 'walk-through' of the core functions of the system. This should include highlighting how the system behaves differently for different user types, and how each user type can achieve their individual goals within the system.

Appendices

A. ERD

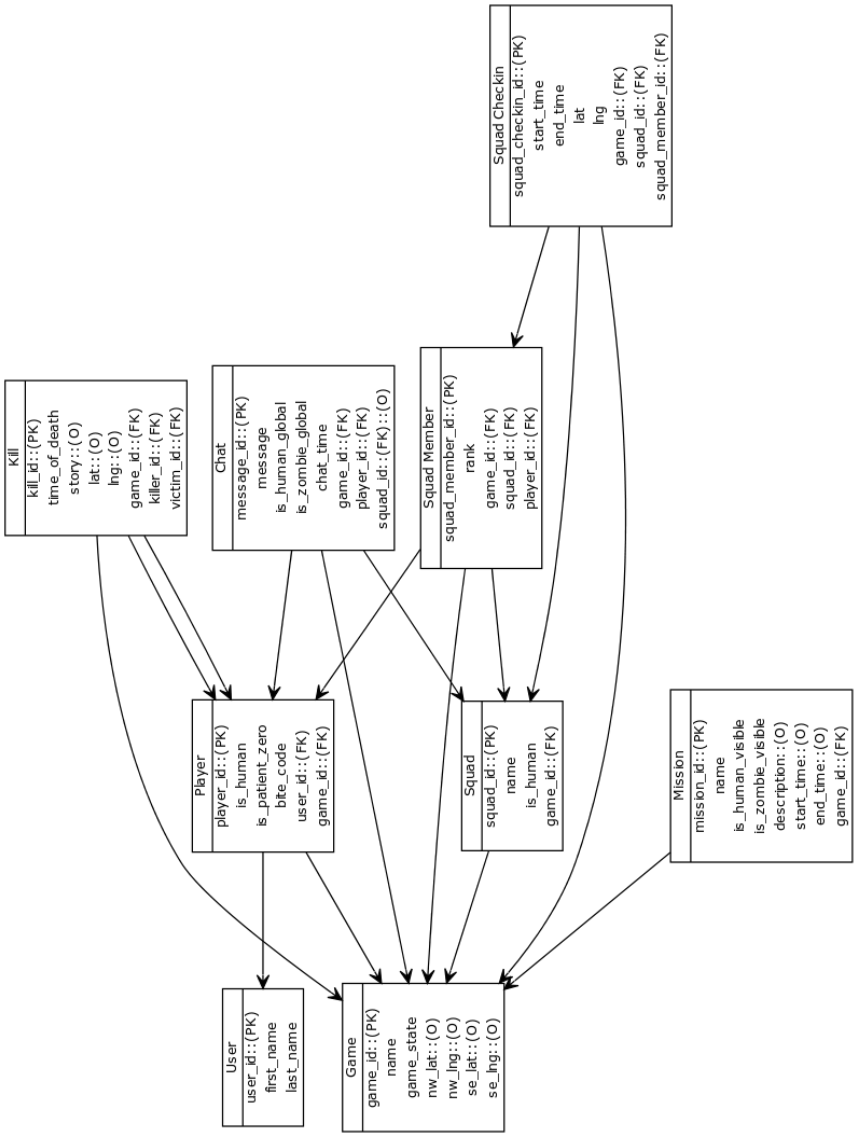

Figure A.1.: Proposed ERD for HvZ System