Liar's Poker

Francesco Buda
francesco.buda3@studio.unibo.it

Emanuele Sanchi
emanuele.sanchi@studio.unibo.it

Tommaso Severi
tommaso.severi2@studio.unibo.it

Febraury 2025

Up to ∼2000 characters briefly describing the project.

1 Concept

This project aims to develop a web-based game application for playing Liar's Poker, a fun and engaging card game centered around deception and bluffing. Inspired by traditional poker, Liar's Poker is played with a standard deck of cards and is suitable for two or more players. Our application will provide an online platform where users can enjoy the game with others in real-time, no matter where they are.

1.1 Game Rules

Each player starts with one card. A round starts with someone naming a poker combination. With the turn rotating clockwise, each next player has a choice:

- **Rise the stakes**

  name a higher poker combination than the one named by the previous player that they belive is present in the combined cards of all players.

- **Call bullshit**

  the player calls bullshit on the previous player's combination if the player think the previous player is bluffing or mistaken.

When someone calls bullshit, all players reveal their cards and the presence of the contested poker hand is verified. If the hand is present, the player who called bullshit loses the round; else, the player who named this poker hand loses.

The loser starts all subsequent rounds with one extra card, and names the first combination in the next round.
if someone loses when they have 5 cards, they are kicked from the game. The last person remaining in the game is the winner.

1.2 Questions and Answers

- **Question:** *How do users interact with the system?*

  Answer: Users can access the application through a web browser.

- **Question:** *What happens if one player disconnect during the game?*

  Answer: The game will pause until the player reconnects. If the player does not reconnect within a certain time frame, they will be kicked from the game.

- **Question:** *What happens if the server dies during the game?*

  Answer: The game goes on

- **Question:** *Does the system need to store player's data?*

  Answer: The system doesn't need to store data, as the player does not need to create an account.

1.3 Usage Scenarios

This section describes the typical usage scenario of a group of friends playing Liar's Poker on our platform.

1. **Lobby creation**

   One user creates a new lobby and shares the access code with friends.

2. **Joining the lobby**

   Other users join the lobby by entering the code. While joining, the players will be asked to choose a unique nickname to use during the game.

3. **Gameplay**

   Each player receive one card and than the game starts (The first player is chosen randomically). The game proceeds as described in the rules. when the player want to rise the stakes, he can choose from the possible combinations (non valid combinations will be disabled), if the combination needs some specific card to be declared, the player will be provided with a list of possible cards to choose from.

4. **Elimination**

   When a player loses, they are eliminated from the game and they can chose to spectate the game or leave the lobby.

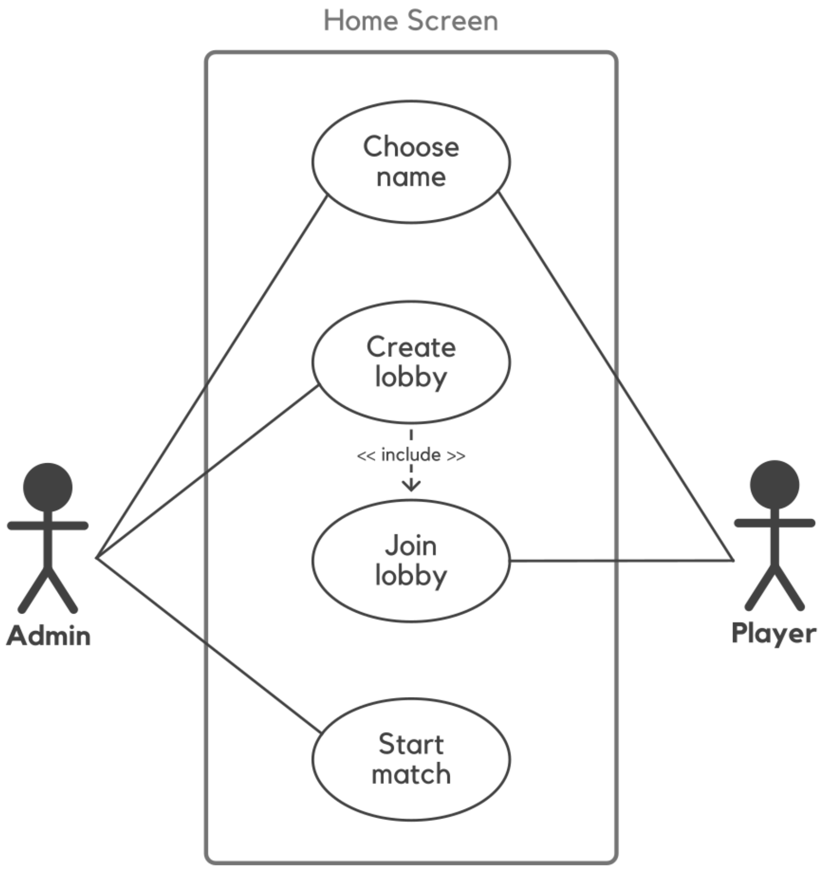


Figure 1: Use case diagram of the homescreen

5. **End of the game**

   When the game ends the winner is declared and players who are still in the lobby will be asked wether they want to play another game or leave the lobby.

Here you should explain:

- The type of product developed with that project, for example (non-exhaustive):
  - Application (with GUI, be it mobile, web, or desktop)
  - Command-line application (CLI could be used by humans or scripts)
  - Library
  - Web-service(s)
- Use case collection

- *where* are the users?
- *when* and *how frequently* do they interact with the system?
- *how* do they *interact* with the system? which *devices* are they using?
- does the system need to *store* user's **data**? *which*? *where*?
- most likely, there will be *multiple* **roles**

2 Requirements

- The requirements must explain **what** (not how) the software being produced should do.
  - you should not focus on the particular problems, but exclusively on what you want the application to do.
- Requirements must be clearly identified, and possibly numbered
- Requirements are divided into:
  - **Functional**: some functionality the software should provide to the user
  - **Non-functional**: requirements that do not directly concern behavioural aspects, such as consistency, availability, etc.
  - **Implementation**: constrain the entire phase of system realization, for instance by requiring the use of a specific programming language and/or a specific software tool
    * these constraints should be adequately justified by political / economic / administrative reasons...
    * ... otherwise, implementation choices should emerge *as a consequence of* design
- If there are domain-specific terms, these should be explained in a glossary
- Each requirement must have its own **acceptance criteria**
  - these will be important for the validation phase

3 Design

This chapter explains the strategies used to meet the requirements identified in the analysis. Ideally, the design should be the same, regardless of the technological choices made during the implementation phase.

> You can re-order the sections as you prefer, but all the sections must be present in the end

3.1 Architecture

The architecture of the system is a classic MVC where:

- Model are the classes that represents players and cards
- View is UI realized with Web components
- Controller is the class that manages the game logic

For the communication architecture and the distributed part, we opted for a client-server architecture using also a publish-subscribe pattern.
The idea is that the broker manages the communication between the clients and the server, and the server manages the game logic.
The server is the one that creates the game; then the clients can join the game publishing their nickname to the broker; every time a new player joins the game, it automatically subscribe to the game topics.
All the clients are subscribed to the same topic, so they can receive the messages from the server (eg. moves, game status, etc.).

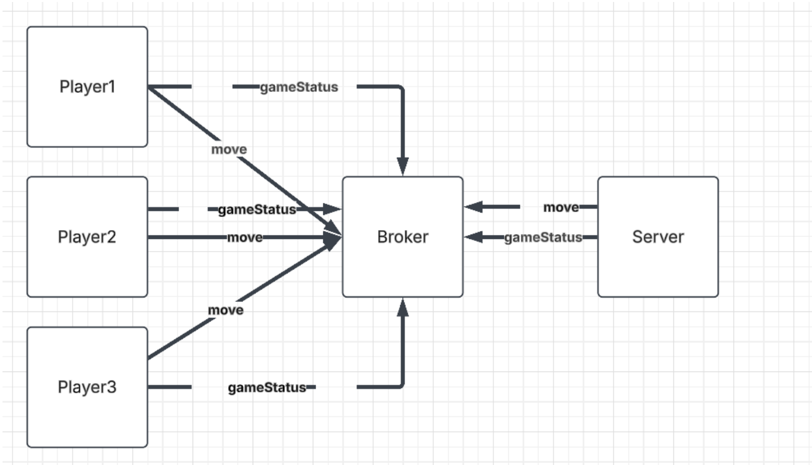
We choose this architecture because it is simple and it is suitable for our project. In fact, using a publish-subscribe architecture, clients have not to request the server for the game status, but they receive it automatically when the server publishes it (and so on for the other messages).

3.2 Infrastructure

- both players and server are components of the pub-sub model
- there is one server that manages the game logic so is the only one who can publish messages on some specific topics
- there is a broker that manages the communication between the clients and the server
- data aren't store permanently, so there is no need for a database
- there aren't authentication mechanisms so every one who knows the broker address can join the game

Components are distributed over all the network, so they can be on different machines and different networks. For semplicity, we used the same network for all the components and the same machine both for server and broker.
To find each other, the components use the broker address and the topics to which they have to subscribe or publish; so clients are only required to know the broker address to join the game.
Components are named using a nickname (unique) that every body choose when they join the game.

Figure 2: Architecture diagram



Dotted lines are for subscribe messages, while solid lines are for publish messages.

3.3 Modelling

The system is fundamentally composed by the classes shown in fig. 3.
Each component models the real world entities that are needed to play the game:

- Players join and create matches, having each one of them has a nickname and a list of cards.
- The lobbies are the places where the players can join for a match, waiting for there to be enough people.
- The games have a set of players and a deck of playing cards. Each game has a start, where the cards are shuffled and given to players, a playing time when players are given turns to play, and an end when a winner is found.

At the core of the entire system are messages, these, given the simple nature of the game, are quite simple: containing a single string that represents the move the player wants to make.
The real difference and context of the message is given by the topic to which it is published: ff a name is passed to the "player" topic it will be interpreted as the player's nickname, if it is passed to the "game" topic it will be interpreted as the player's move.
Most notably these are the main topics:

- **player**

  The topic where the players publish their nickname to join the game.

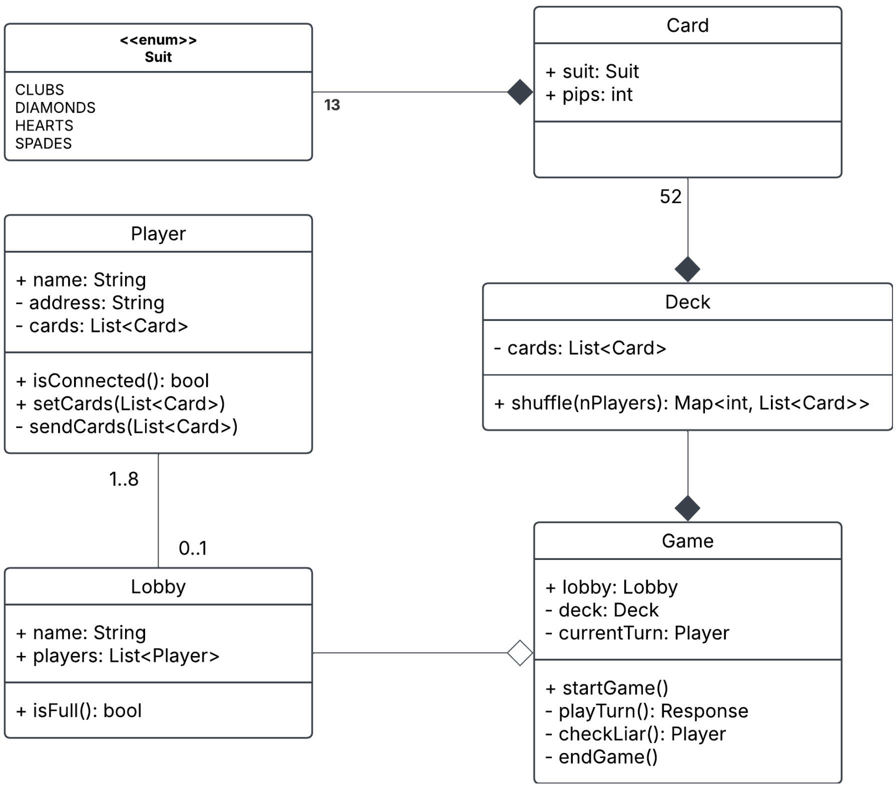


Figure 3: Class diagram of the system

- **game**

  The topic where the players publish their moves and the server the game status. The name of the topic varies depending on the lobby the player is in.

- **lobby**

  The topic where the players create and join lobbies.

3.4 Interaction

Components communicate over the network using the MQTT protocol so using a Message Broker pattern. Using this pattern, publishers can send messages without knowing who is going to receive them, and subscribers can receive messages without knowing who is going to send them. The broker is the one who manages the communication between the clients and the server, so the clients don't have to know the server address, but only the broker address; on the other hand, the server has to know the broker address to publish messages, but it doesn't have to know the clients' addresses.
This kind of architecture is highly decoupled, so the components can be easily replaced or modified without affecting the others.
The following subsections will explain the interaction between the components in more detail.

3.4.1 Connection

3.4.2 Game Move

3.4.3 End Game

3.5 Behaviour

Even if for the game to be played are only needed players and a centralized logic, more actors are needed to keep the game going and to manage the communication between them.
The following are all the components actually needed and their behaviours:

- **The client** fig. 4

  The client first asks the player for their name and the tries to connect him to the broker. If the connection is successful, the client can choose to join a lobby or create a new one.
  When the game starts, the client waits for the server to publish the game status and then it can start playing. The client can only send messages to the server when it is his turn, either rising the stakes or calling bullsh*t. If they player lost the match can only wait for it to finish. The player leaves the lobby only when a winner is found or loses the game.
  At any point during the connection the player can leave or be disconnected, if the player wants to rejoin the system will try to put him back in a lobby if they where in one.

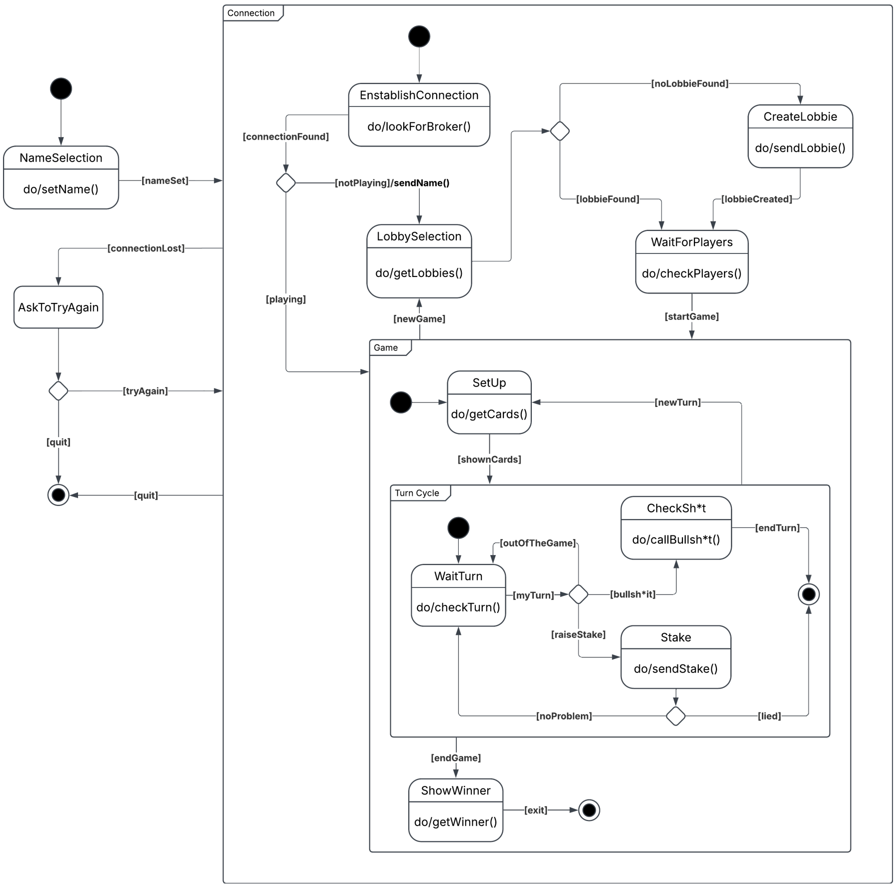


Figure 4: State diagram of the client

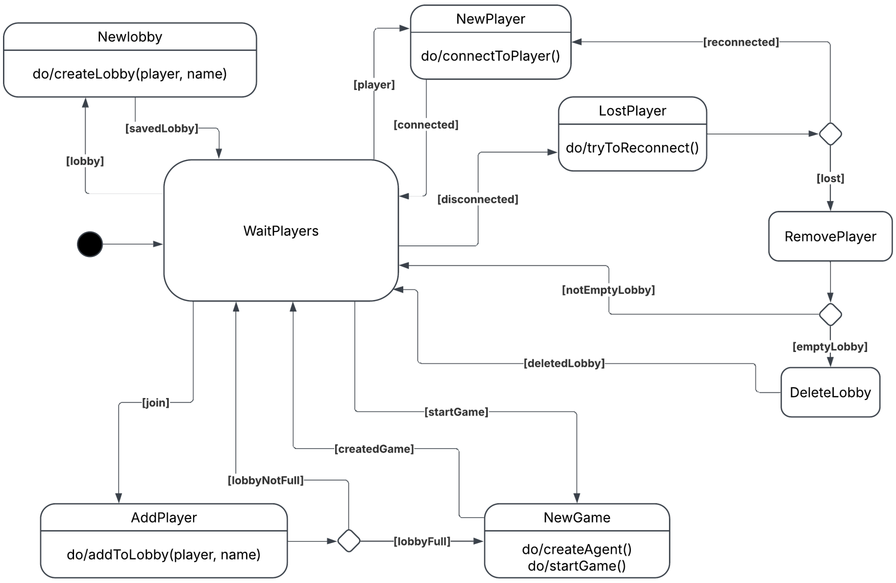


Figure 5: State diagram of the server

- **The server** fig. 5

  The server is mainly in charge of connecting new players and creating lobbies, delegating the game logic to an agent.
  At any point in time a new player can ask to connect or disconnect himself, causing the need to check wether the player was alone, deleting it if left empty. The players connected can also ask to create or join a new lobby and, in case it's full, it will launch a new game istance.

- **The broker** fig. 6

  The broker is the central core to all communications between all the components. Always active and using the MQTT standard.

- **The game agent** fig. 7

  The game agent is an istance of the server that takes on the role of managing the single matches. It is in charge of managing the game state, the players' moves and the game logic.
  It also preoccupies itself to reconnect the players and resume the game where it left off in case of disconnection. Only checking each connection every time it's the player's turn to save some resources.

- **The backup** fig. 8

  The backup has to keep a record of every message sent by the broker, keeping a copy of server state.
  This way if the server or the broker fails it can launch a new istance of whichever failed and resume the service.

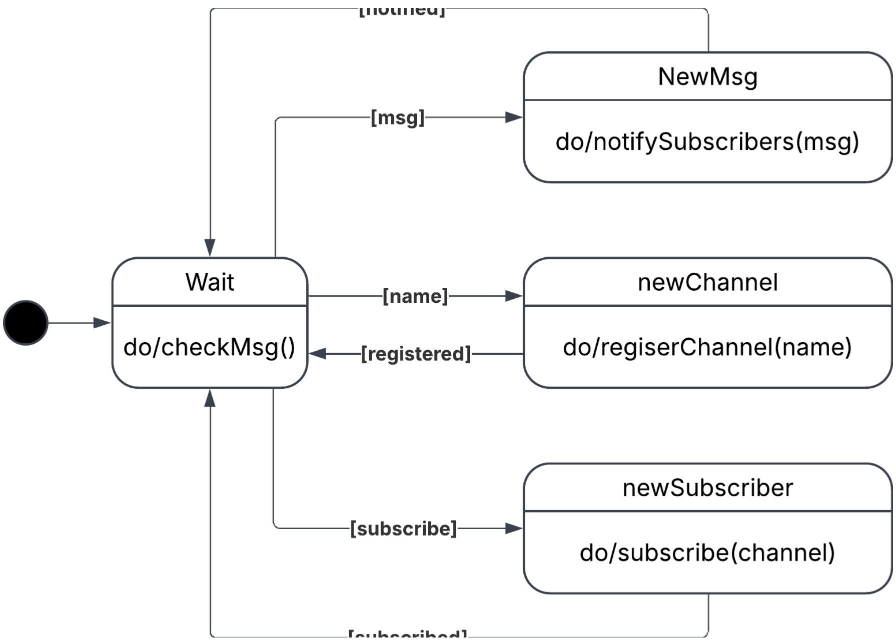


Figure 6: State diagram of the broker

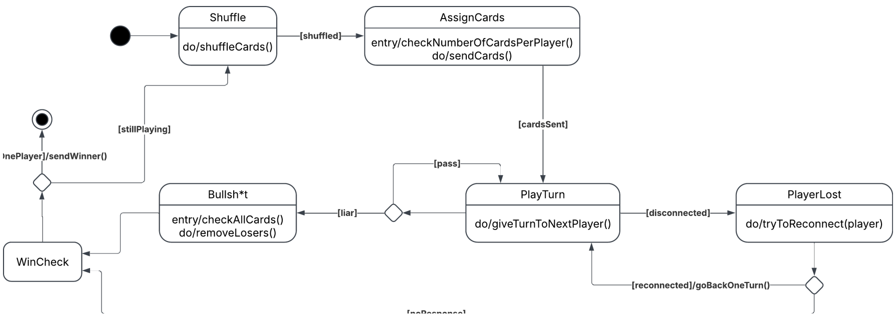


Figure 7: State diagram of the game agent

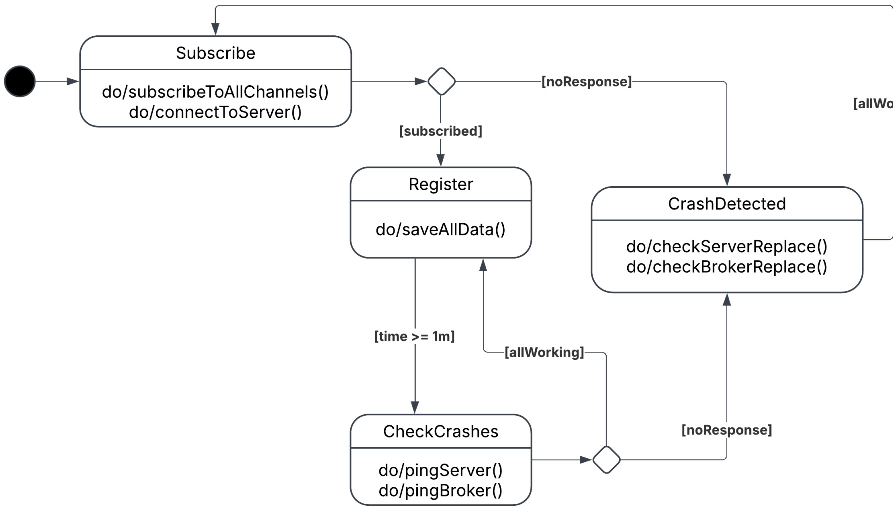


Figure 8: State diagram of the backup

3.6 Data and Consistency Issues

Since we don't need to store data permanently because this is a card game, we don't deal with queries and databases.
Some datas are shared between components, for example the game status, the players' nicknames, and other crucial information to perform the game.

3.7 Fault-Tolerance

Different types of fault tolerance are implemented in the systems, everything is set up to keep the game running even if some components fail.
The mechanics differ in their scope and purpose:

- **Server-Side**

  The server and the broker are at the core of the game, if they fail the game is lost. To prevent this from happening, they are costantly monitored (about once per minute) by a backup server. The whole porpouse of the backup is to keep a copy of the game state and to be able to resume the game where it left off, in case of failure it launches a new istance of whichever failed, setting it up with the saved data.

- **Network-Side**

  To make sure that no accidental disconnection can ruin the game, the game agent checks the connection of each player every time it's their turn. If a player is disconnected, the game agent will wait for them to reconnect, if they don't in a certain time frame they will be kicked from the game.

Furthermore, the client always has the opportunity to try to reconnect, picking where it left off, if anything goes wrong.

3.8 Availability

At this time we don't really have an availability mechanism.
The only type of caching we have is the broker that keeps the last message sent in a topic and the backup server that keeps a copy of the game state in case of failure.

3.9 Security

There aren't any security mechanisms in our project.

---

4 Implementation

- which **network protocols** to use?
  - e.g. UDP, TCP, HTTP, WebSockets, gRPC, XMPP, AMQP, MQTT, etc.
- how should *in-transit data* be **represented**?
  - e.g. JSON, XML, YAML, Protocol Buffers, etc.
- how should *databases* be **queried**?
  - e.g. SQL, NoSQL, etc.
- how should components be *authenticated*?
  - e.g. OAuth, JWT, etc.
- how should components be *authorized*?
  - e.g. RBAC, ABAC, etc.

4.1 Technological details

- any particular *framework* / *technology* being exploited goes here

5 Validation

5.1 Automatic Testing

- how were individual components ***unit*-test**ed?
- how was communication, interaction, and/or integration among components tested?

- how to ***end-to-end*-test** the system?
  - e.g. production vs. test environment
- for each test specify:
  - rationale of individual tests
  - how were the test automated
  - how to run them
  - which requirement they are testing, if any

  recall that *deployment* **automation** is commonly used to *test* the system in *production-like* environment

  recall to test corner cases (crashes, errors, etc.)

5.2 Acceptance test

- did you perform any *manual* testing?
  - what did you test?
  - why wasn't it automatic?

6 Release

- how where components organized into *inter-dependant modules* or just a single monolith?
  - provide a *dependency graph* if possible
- were modules distributed as a *single archive* or *multiple ones*?
  - why?
- how were archive versioned?
- were archive *released* onto some archive repository (e.g. Maven, PyPI, npm, etc.)?
  - how to *install* them?

7 Deployment

- should one install your software from scratch, how to do it?
  - provide instructions
  - provide expected outcomes

8 User Guide

- how to use your software?
  - provide instructions
  - provide expected outcomes
  - provide screenshots if possible

9 Self-evaluation

- An individual section is required for each member of the group
- Each member must self-evaluate their work, listing the strengths and weaknesses of the product
- Each member must describe their role within the group as objectively as possible.

It should be noted that each student is only responsible for their own section

References

[1] D. Adams. *The Hitchhiker's Guide to the Galaxy.* San Val, 1995.
[1]