# Projektplan

**Phase 1: Stabilisieren & Definieren (Grundlage schaffen)**

1. **Gegenlesen/Prüfen/Inline-Doku/Logs/TODOs:**
   * **Warum zuerst?** Bevor du neue Funktionalität hinzufügst oder Tests schreibst, sollte der *aktuelle* Code so sauber und verständlich wie möglich sein. Das Beheben kleinerer TODOs, das Übersetzen von Logs und das Hinzufügen von Docstrings (RestructuredText ist super!) schafft eine solide Basis und vermeidet, dass du Fehler oder Unklarheiten in neue Bereiche "mitziehst".
   * **Fokus:** Gehe jede Klasse und Methode durch. Sind die Namen klar? Funktionieren die Abläufe logisch? Sind die deutschen Log-Einträge präzise? Sind alle deine Kommentare noch aktuell? Füge Docstrings hinzu, die erklären, *was* eine Funktion/Klasse tut, welche Parameter sie erwartet und was sie zurückgibt.
2. **Nachrichten / Kommunikation zw. Server und Client definieren:**
   * **Warum jetzt?** Du kannst die Spiellogik in der GameEngine und den Test-Client (oder später die WebApp) nicht sinnvoll implementieren, wenn du nicht genau weißt, *welche* Daten in *welchem Format* (JSON-Struktur) ausgetauscht werden.
   * **Fokus:** Lege fest, wie z.B. eine "Spielkarte spielen"-Aktion vom Client aussieht ({'action': 'play\_cards', 'payload': {'cards': ['H10', 'H1']}}?), wie ein "Spielzustand-Update" vom Server aufgebaut ist ({'type': 'public\_state\_update', 'payload': {...PublicState...}}), wie Fehler gemeldet werden etc. Dokumentiere diese Strukturen.

**Phase 2: Kernlogik & Basis-Tests**

1. **Abgleich mit deprecated Code / Implementierung PublicState/PrivateState/GameEngine:**
   * **Warum jetzt?** Mit den definierten Nachrichtenstrukturen kannst du nun die Platzhalter in der GameEngine mit Leben füllen. Implementiere die Kernmechaniken: Karten austeilen, handle\_player\_message für die definierten Aktionen (Karten spielen, passen, Tichu ansagen?), Runden-/Stichlogik, Punkte zählen (zumindest grundlegend). Fülle PublicState und PrivateState mit den notwendigen Attributen.
   * **Fokus:** Ziel ist ein minimal funktionsfähiger Spielablauf innerhalb der Engine, der auf die definierten Nachrichten reagiert.
2. **Test-Client schreiben:**
   * **Warum jetzt?** Sobald die Engine auf grundlegende Nachrichten reagieren kann, ist ein Test-Client *Gold wert*. Du kannst damit viel schneller und gezielter testen als mit einer echten WebApp oder nur mit Unit-Tests. Du kannst damit das Verbinden, Trennen, Senden von Spielzügen etc. automatisieren und die Serverantworten prüfen.
   * **Fokus:** Ein einfaches Python-Skript (mit aiohttp als Client) das eine WebSocket-Verbindung aufbaut, sich authentifiziert (Parameter sendet), die definierten Nachrichten senden und empfangene Nachrichten ausgeben oder rudimentär prüfen kann.

**Phase 3: Umfassende Tests & Dokumentation**

1. **Unit-Tests schreiben:**
   * **Warum jetzt?** Mit dem Test-Client kannst du die Gesamt-Flows testen. Jetzt ist ein guter Zeitpunkt, detailliertere Unit-Tests (z.B. mit pytest) für spezifische, kritische oder komplexe Teile zu schreiben:
     + Einzelne Funktionen der GameEngine (z.B. Validierung von Kartenkombinationen).
     + Die Timer-Logik und Randfälle in der GameFactory.
     + Die Zustandsübergänge in PublicState/PrivateState.
     + **Deine genannten Fälle sind sehr gut!** Ergänzend:
       - Fall 7b: Connect, Tisch beitreten (wo schon 1 Mensch, 2 KIs sitzen).
       - Fall 8: Ungültige Aktion senden (z.B. Karten spielen, die man nicht hat; Aktion senden, wenn man nicht dran ist).
       - Fall 9: Schnelles Verbinden/Trennen/Wiederverbinden (Stresstest für Timer/Reconnect).
       - Fall 10: Fehler bei der JSON-Serialisierung/Deserialisierung.
   * **Fokus:** Teste einzelne Komponenten isoliert und decke Randfälle ab.
2. **Dokumentation für Entwickler schreiben:**
   * **Warum jetzt?** Der Code ist stabilisiert, die Kernlogik implementiert und getestet. Jetzt kannst du die übergeordnete Dokumentation schreiben, die die Architektur, Klassenaufgaben, den Datenfluss und wichtige Designentscheidungen erklärt. Das Glossar ist hier auch sehr sinnvoll.
   * **Fokus:** Eine README.md oder separate Dokumentationsdateien, die neuen Entwicklern (oder deinem zukünftigen Ich) den Einstieg erleichtern.

**Phase 4: Frontend**

1. **WebApp für den Spieler schreiben:**
   * **Warum zuletzt?** Die WebApp benötigt ein funktionierendes und stabiles Backend mit einer klar definierten API (den Nachrichten). Jetzt kannst du dich auf die Client-Seite konzentrieren. Die Portierung von Godot nach HTML/JS/CSS ist ein eigenes Projekt.
   * **Fokus:** UI-Design, Client-seitige Logik, Kommunikation mit dem WebSocket-Server basierend auf den definierten Nachrichten.

**Zusammenfassend die empfohlene Reihenfolge:**

1. Code prüfen/dokumentieren/übersetzen/TODOs bearbeiten
2. Nachrichten/Kommunikation definieren
3. Kernlogik GameEngine/States implementieren
4. Test-Client schreiben (und parallel zu 3 nutzen)
5. Unit-Tests schreiben (detaillierte Fälle)
6. Entwickler-Dokumentation schreiben
7. WebApp schreiben

## Angepasster Plan

Das ist ein sehr sinnvoller Ansatz, um sicherzustellen, dass die grundlegende Infrastruktur (Verbindung, Start, Ende, Reconnect, KI-Übernahme) robust funktioniert.

**Angepasster Plan:**

**Phase 1: Stabilisierung & Grundfunktionalität (Aktuell)**

1. **Code-Überprüfung & Inline-Doku (Fortsetzung):**
   * Gehe die Klassen GameEngine2, GameFactory, Client, Player, Agent, PublicState, PrivateState, Arena2, server.py nochmal durch.
   * Stelle sicher, dass alle Docstrings und Kommentare aktuell und verständlich sind (deutsch).
   * Übersetze verbleibende englische Log-Einträge.
   * Adressiere einfache TODOs, die keine neue Logik erfordern.
   * **WICHTIG:** Stelle sicher, dass GameEngine2 und Arena2 den Wrapper (\_run\_episode\_sync\_wrapper, asyncio.run) korrekt verwenden, wie im vorletzten Beispiel gezeigt.
2. **Nachrichten für Basis-Interaktionen definieren/finalisieren:**
   * C -> S: Beitrittsanfrage (via URL Params).
   * S -> C: joined\_table.
   * C -> S: {"action": "start\_game", "payload": {}} (vom Host-Client).
   * S -> C: public\_state\_update (zeigt Spieler, Phase "lobby", "playing", "game\_over").
   * S -> C: private\_state\_update (zeigt initial leere Hand).
   * S -> C: {"type": "event", "payload": {"event\_name": "game\_started"}}
   * S -> C: {"type": "event", "payload": {"event\_name": "game\_over", "final\_scores": ...}}
   * S -> C: error Nachrichten (Tisch voll, nicht Host etc.).
   * C -> S: leave\_table (optional, für sauberes Verlassen).
3. **Grundlegende Engine-Logik implementieren (Platzhalter für Runden):**
   * **GameEngine2.start\_game:** Startet \_run\_game\_loop.
   * **GameEngine2.\_run\_game\_loop:**
     + Setzt current\_phase auf "playing".
     + Sendet game\_started Event.
     + **PLATZHALTER:** Anstatt \_run\_round aufzurufen, simuliert es einfach ein sofortiges Spielende nach einer kurzen Pause oder nach einer festen Anzahl "Runden".
     + # Beispiel Platzhalter in \_run\_game\_loop
     + self.public\_state.current\_phase = "playing"
     + await self.\_broadcast\_public\_state()
     + await self.players[0].notify("event", {"event\_name": "game\_started"}) # Beispiel Event
     + logger.info(f"[{self.table\_name}] Spiel gestartet (Rundenlogik Platzhalter).")
     + await asyncio.sleep(5) # Simuliere Spielzeit
     + logger.info(f"[{self.table\_name}] Spiel künstlich beendet (Platzhalter).")
     + # Setze fiktive Endergebnisse
     + self.public\_state.total\_scores = [1050, 800] # Beispiel
     + self.public\_state.is\_game\_over = True

# Restliche Logik im Loop erkennt is\_game\_over und beendet.

content\_copydownload

Use code [with caution](https://support.google.com/legal/answer/13505487).Python

* + **GameEngine2.handle\_player\_message:** Implementiert die Reaktion auf start\_game. Andere Aktionen können vorerst ignoriert oder mit Fehler beantwortet werden.
  + **Reconnect/KI-Übernahme:** Die Logik in GameFactory und GameEngine2 (wie zuletzt besprochen) für Timer-Start, replace\_player\_with\_agent und Timer-Abbruch bei Reconnect muss implementiert und funktionsfähig sein.

**Phase 2: Tests & Dokumentation für den Rahmen**

1. **Test-Client anpassen/erstellen:**
   * Der Client muss sich verbinden können (ws://...?tableName=X...).
   * Er muss die playerId speichern und bei erneutem Verbinden mitsenden.
   * Er muss eine start\_game-Nachricht senden können (wenn er Host ist).
   * Er muss joined\_table, public\_state\_update, game\_started, game\_over, error empfangen und anzeigen können.
   * Er sollte Verbindungsabbrüche simulieren können.
2. **Unit-/Integrations-Tests schreiben (Fokus auf Rahmen):**
   * **Verbindung/Beitritt:**
     + Client verbindet sich, kommt an Tisch.
     + Zweiter Client verbindet sich.
     + Tisch voll -> Fehler.
     + Reconnect erfolgreich.
   * **Spielstart/-ende:**
     + Host startet Spiel -> game\_started Event, Phase wechselt.
     + Spiel endet (künstlich) -> game\_over Event, Scores.
     + Nicht-Host versucht zu starten -> Fehler.
     + Start mit weniger als 4 Spielern -> Fehler.
   * **Disconnect/Reconnect/KI-Übernahme:**
     + Client disconnected -> Timer startet (prüfen durch Logging/internen Status?).
     + Client reconnected *vor* Timeout -> Timer wird abgebrochen.
     + Client reconnected *nach* Timeout -> Beitritt wird abgelehnt (oder neuer Slot gesucht?).
     + Client disconnected, Timeout -> replace\_player\_with\_agent wird aufgerufen, Spieler wird zu Agent im public\_state.
     + Letzter Client disconnected, Timeout -> KI ersetzt, is\_empty\_of\_humans wird True, Tisch wird von Factory entfernt (remove\_game).
3. **Entwickler-Dokumentation (Basis):**
   * Aktualisiere die Docstrings (haben wir schon gemacht).
   * Erstelle eine README.md oder eine separate Doku-Seite, die beschreibt:
     + Die aktuelle Architektur (Server, Factory, Engine, Player, States).
     + Den Verbindungsablauf.
     + Den Spielstart/-ende-Ablauf (mit Platzhalter-Runden).
     + Die Reconnect/KI-Logik.
     + Die definierten WebSocket-Nachrichten (Basis).
     + Wie man den Server startet und einen Test-Client verbindet.

**Phase 3: Implementierung der Rundenlogik (später)**

1. Implementierung von \_run\_round, \_play\_stich etc.
2. Detaillierte Tests für Spielzüge, Regeln, Punkte.
3. Erweiterung der Entwickler-Doku.
4. WebApp.