

---

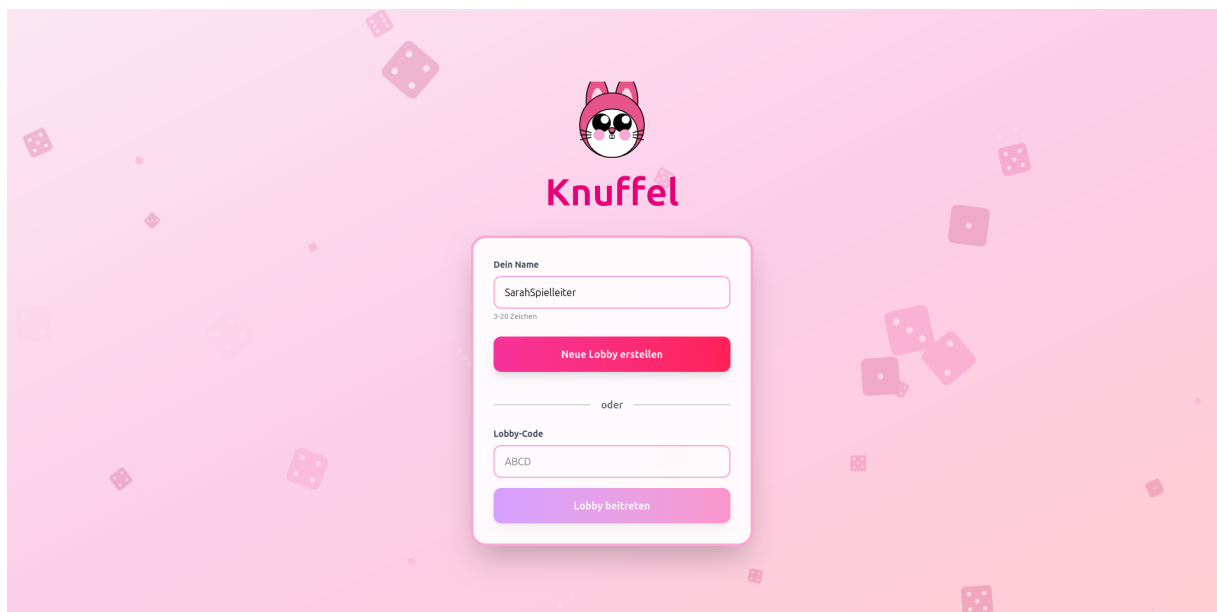
# Knuffel

Wie Kniffel, aber flauschiger

Wir haben ein echtzeitfähiges Multiplayer Online Kniffelspiel entwickelt.

## Demo

### Startseite



The image shows the start page of the 'Knuffel' game. It features a pink background with floating dice. At the top center is a pink cat icon with the word 'Knuffel' below it. In the center is a white login/lobby creation form with a pink border. The form has two sections: 'Dein Name' with a text input field containing 'SarahSpielleiter' and a '3-20 Zeichen' hint, and a red 'Neue Lobby erstellen' button; and 'Lobby-Code' with a text input field containing 'ABCD' and a pink 'Lobby beitreten' button. Between these sections is the word 'oder' flanked by horizontal lines.

Knuffel

Dein Name

SarahSpielleiter

3-20 Zeichen

Neue Lobby erstellen

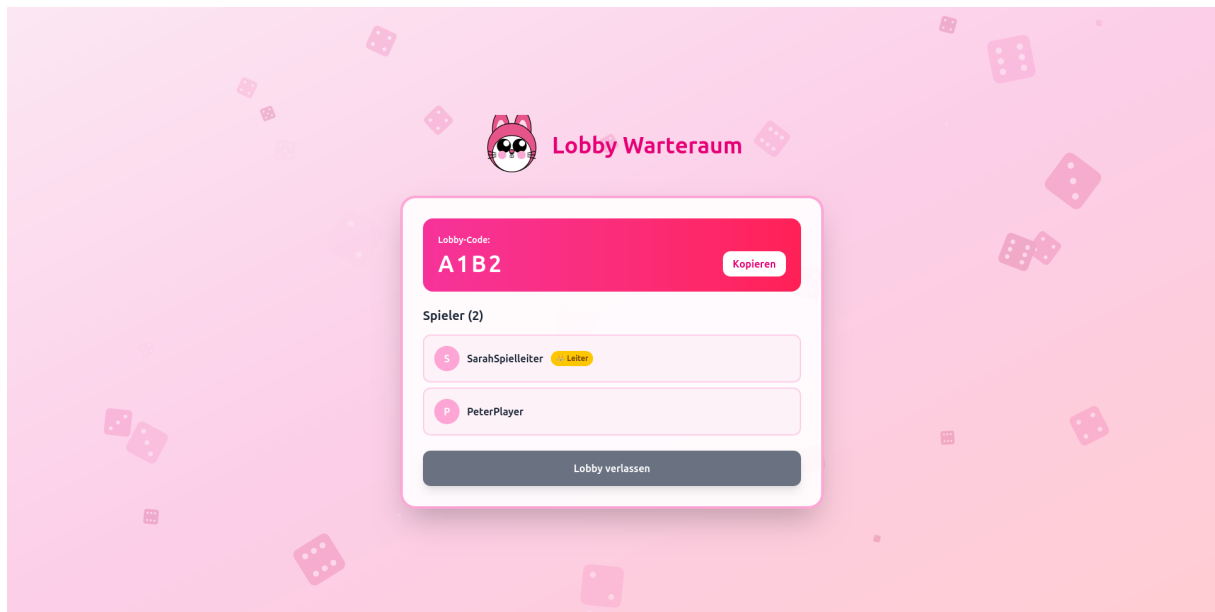
oder

Lobby-Code

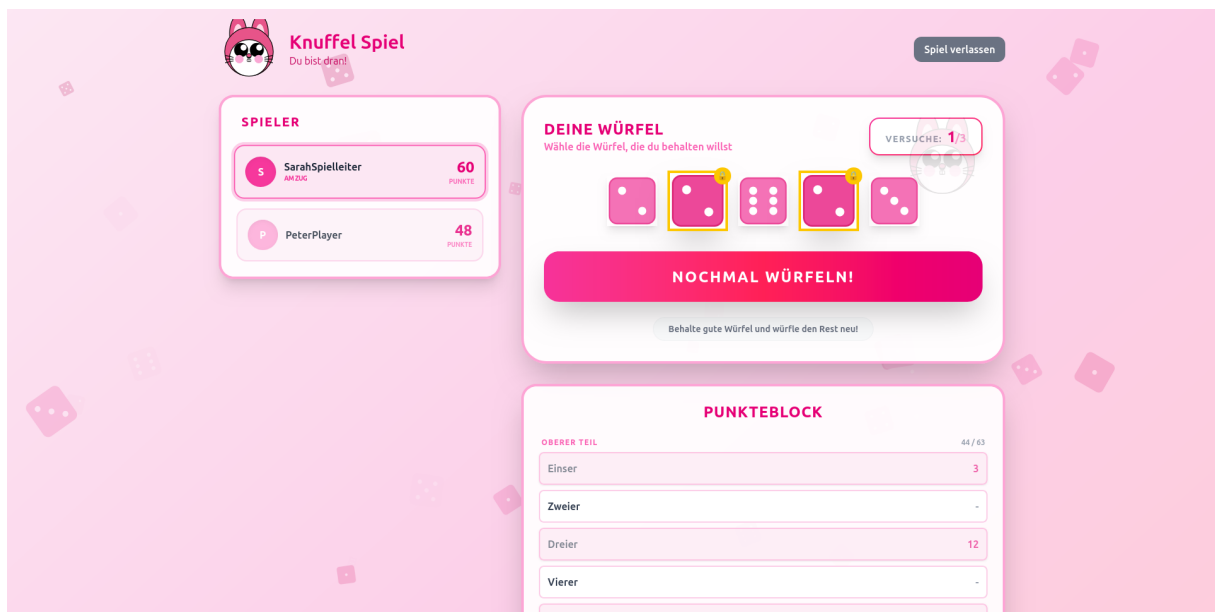
ABCD

Lobby beitreten

## Lobbyseite

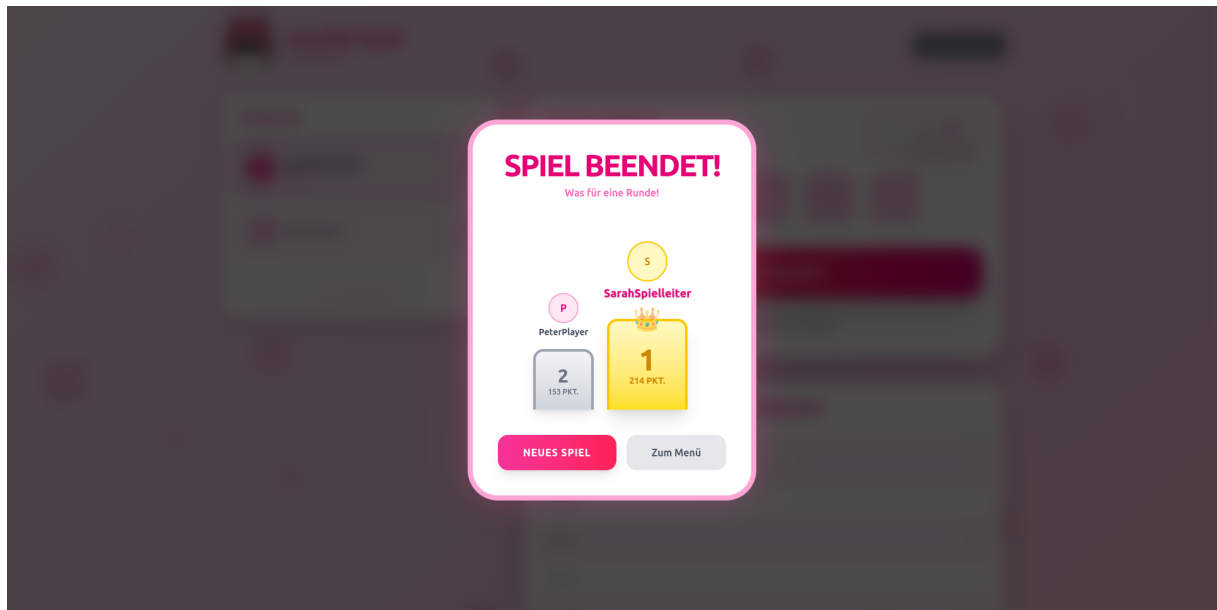


## Spielseite



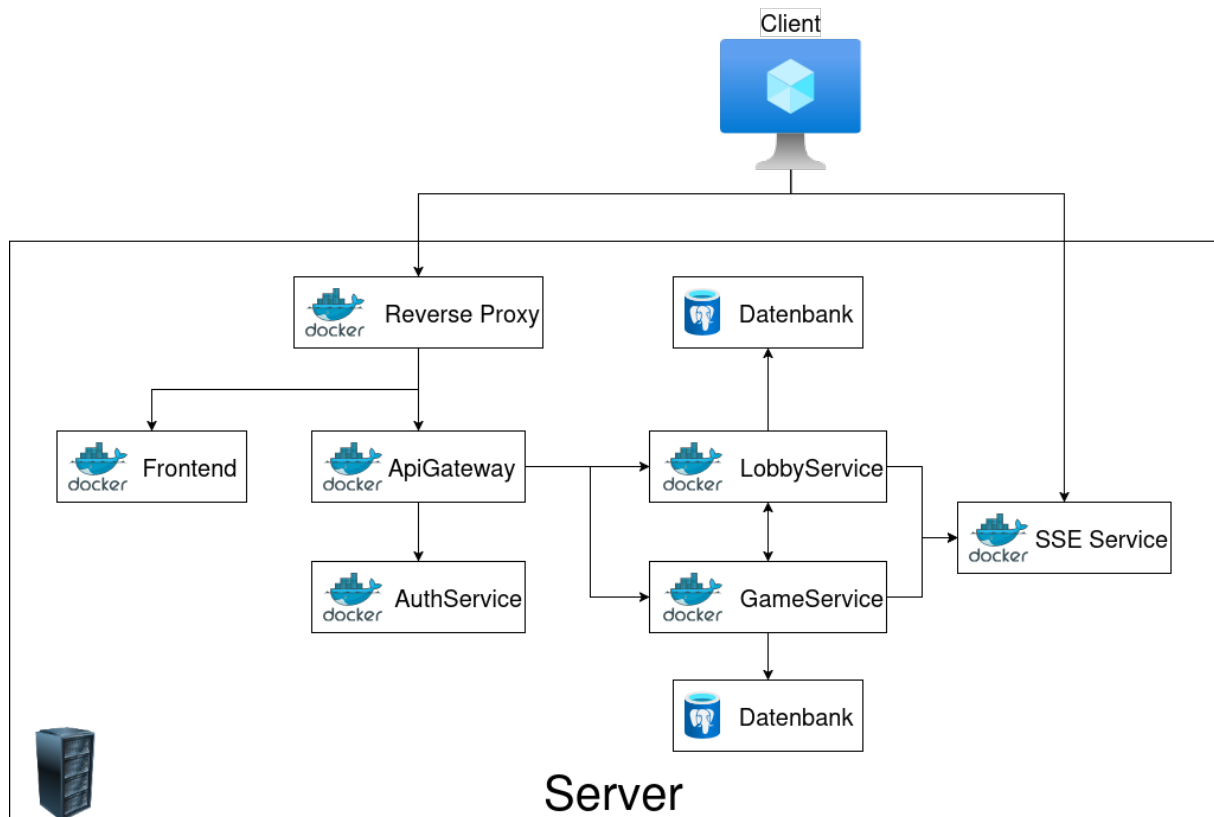
---

## Ergebnisseite



---

## Systemarchitektur



Die Systemarchitektur besteht aus einem Server und unterschiedlichen Diensten, die in Form von Docker-Containern bereitgestellt werden. Durch die Containerisierung sind alle Dienste isoliert und kommunizieren ausschließlich über das virtuelle Docker-Netzwerk miteinander.

Im Schaubild sind folgende Dienste dargestellt:

- **Reverse Proxy:** Der Reverse Proxy leitet die Anfragen der Clients an den entsprechenden Dienst (Frontend / Backend Services) weiter. Zusätzlich sorgt er für die TLS-Verschlüsselung der Kommunikation. Er erlaubt den Zugriff auf die Dienste über eine einzige öffentliche Domain.
- **Frontend:** Das Frontend ist eine Single Page Application (SPA), die in React entwickelt wurde. Es stellt die Benutzeroberfläche des Kniffelspiels dar und kommuniziert mit den Backend Services über REST-APIs und Server-Sent Events (SSE).
- **Api Gateway:** Das API Gateway fungiert als zentraler Einstiegspunkt für alle Backend-Services. Es leitet die Anfragen der Clients an den entsprechenden Microservice weiter, validiert die JWT-Tokens zur Authorisierung und fügt den User-Header zur Authentifizierung hinzu.
- **Auth Service:** Der Auth Service ist für die Authentifizierung und Autorisierung der Benutzer zuständig. Stellt JWT-Tokens aus und verifiziert diese für das API Gateway.

- 
- **Lobby Service:** Der Lobby Service verwaltet die Spiel-Lobbys, in denen sich die Spieler vor dem Spielstart treffen. Er ermöglicht das Erstellen, Beitreten und Verlassen von Lobbys und hält die Spielerinformationen.
  - **Game Service:** Der Game Service ist für die Spiellogik und den Spielablauf verantwortlich. Er verwaltet die Spielzustände, Würfelwürfe und Punktevergabe.
  - **SSE Service:** Der SSE Service stellt eine Verbindung zu den Clients her, um Echtzeit-Updates über Server-Sent Events zu senden. Er empfängt Ereignisse von den anderen Diensten und leitet sie an die entsprechenden Clients weiter.
  - **Datenbanken:** Lobby und Game Service besitzen jeweils eine eigene Datenbank zur Speicherung der relevanten Daten.

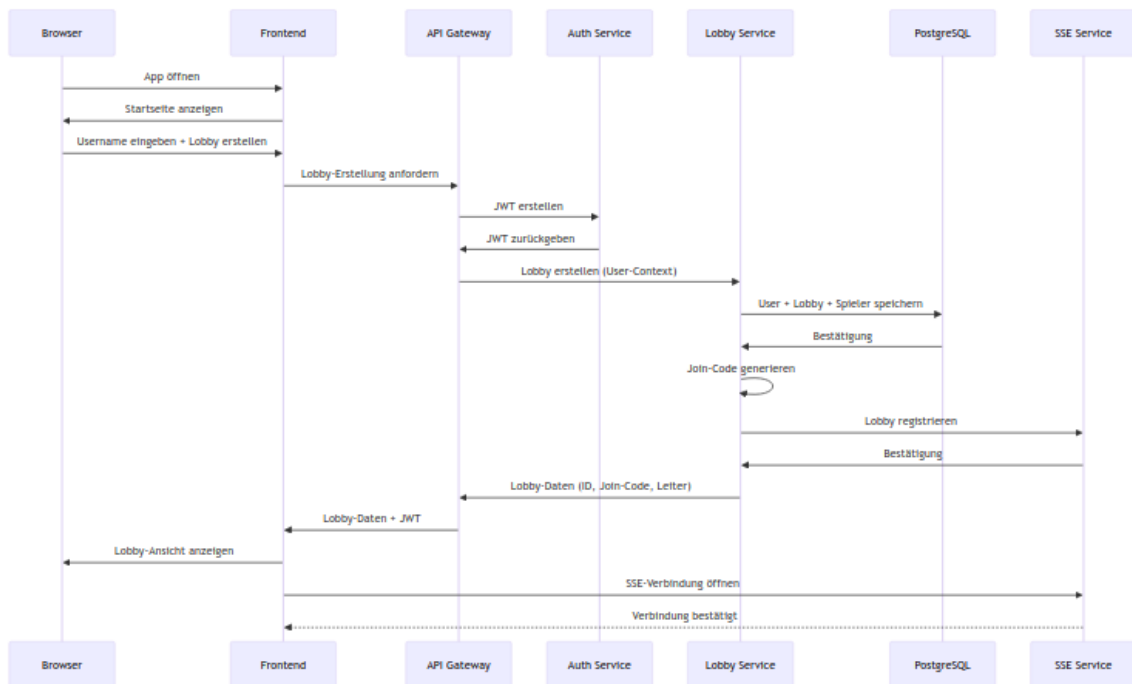
## Userflows

### Flow 1: User-Onboarding & Lobby erstellen

#### Szenario

Ein neuer User öffnet die App, gibt einen Username ein und erstellt eine neue Lobby.

## Sequenzdiagramm



## Wichtige Datenflüsse

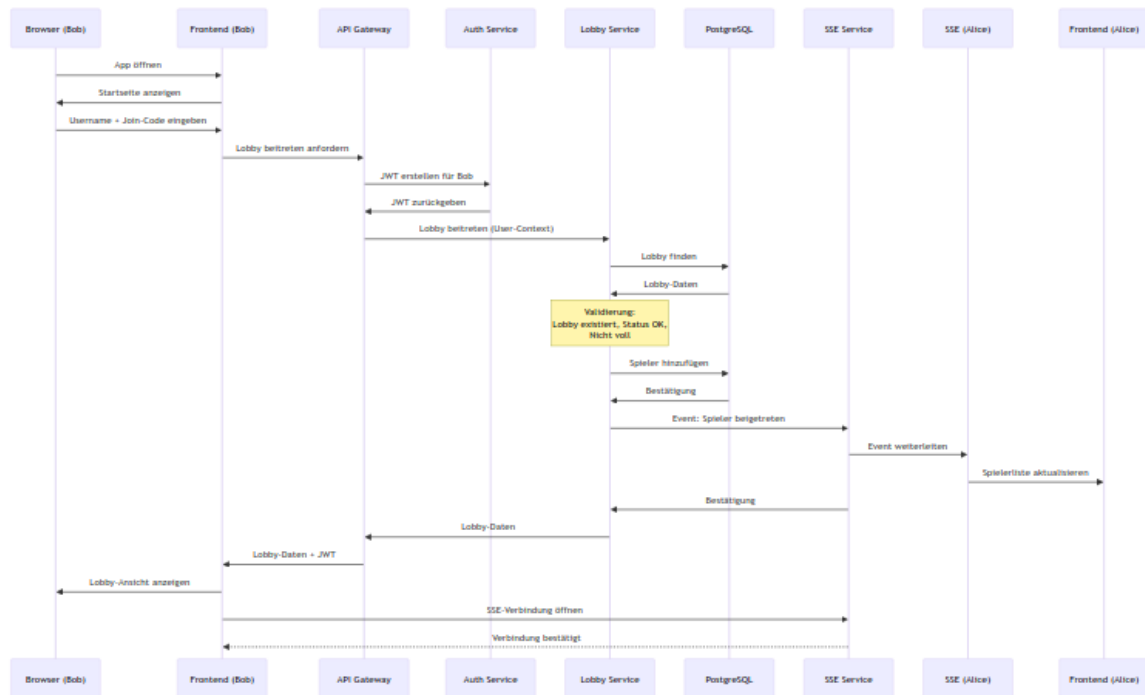
- Username → JWT-Erstellung
- Lobby-Daten → Persistierung
- Join-Code-Generierung
- SSE-Verbindung für Live-Updates

## Flow 2: Lobby beitreten

### Szenario

Ein zweiter User (Bob) möchte der Lobby von Alice beitreten. Er gibt seinen Username und den Join-Code ein.

## Sequenzdiagramm



## Wichtige Datenflüsse

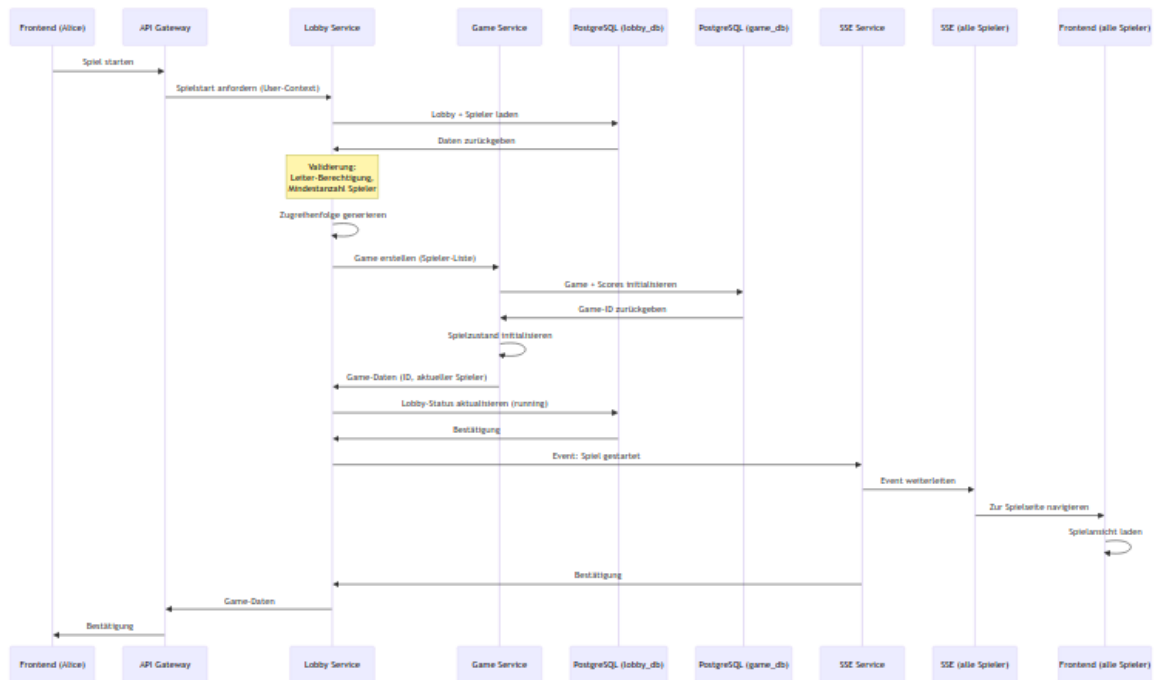
- Join-Code → Lobby-Identifikation
- Spieler-Hinzufügen → Persistierung
- Event → Alle Lobby-Mitglieder
- Neue SSE-Verbindung

## Flow 3: Spiel starten

### Szenario

Alice (Lobby-Leiter) klickt auf “Spiel starten”. Das System erstellt ein Game, legt die Zugreihenfolge fest und leitet alle Spieler zur Spielseite weiter.

## Sequenzdiagramm



## Wichtige Datenflüsse

- Leiter-Berechtigung → Validierung
- Zufällige Zugreihenfolge → Generierung
- Game-Initialisierung → Persistierung
- Event → Alle Spieler gleichzeitig
- Lobby-Status-Änderung (waiting → running)

## Flow 4: Spielzug komplett

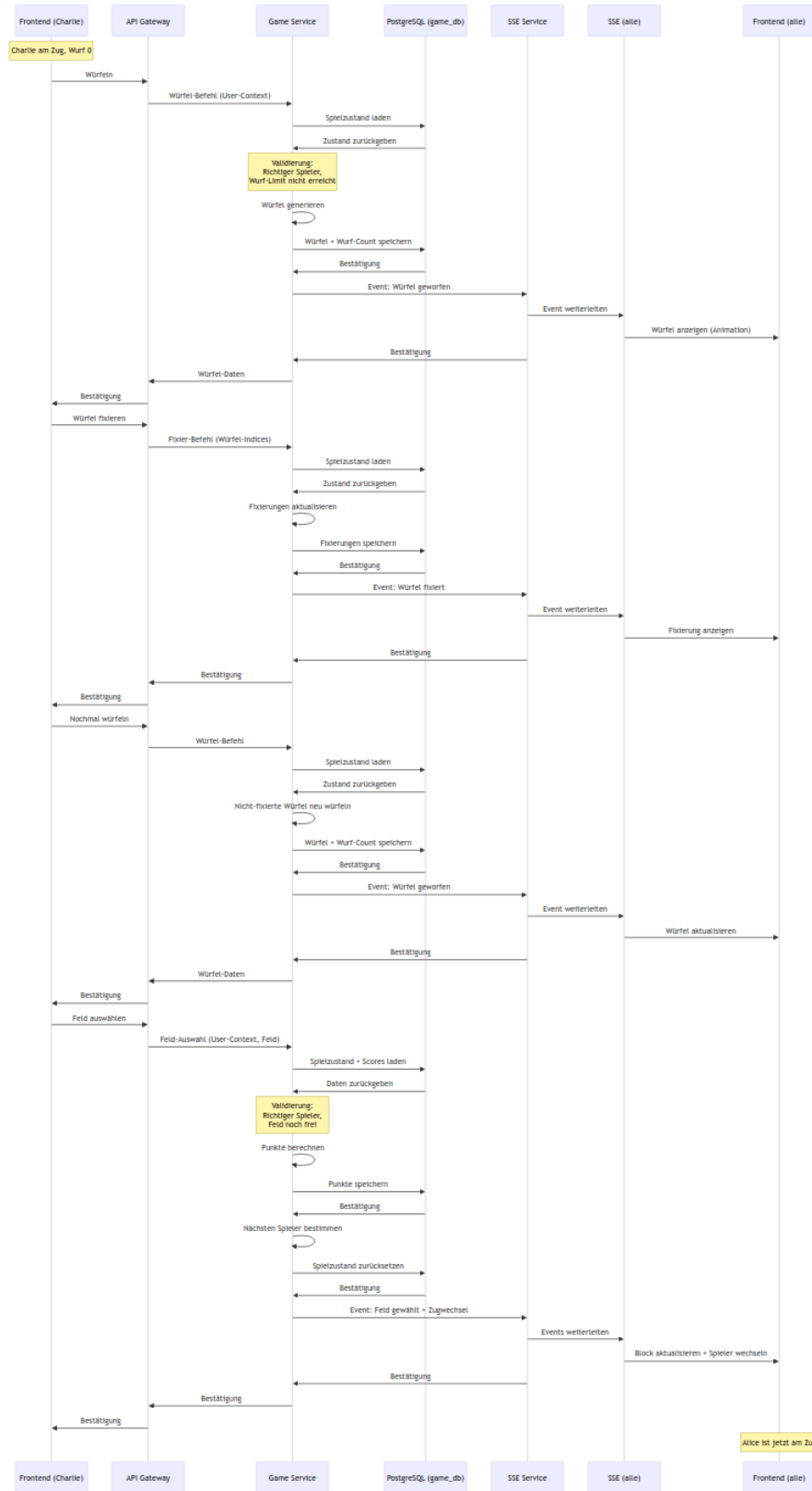
### Szenario

Charlie ist am Zug. Er würfelt dreimal, fixiert Würfel zwischen den Würfeln, wählt dann ein Feld aus. Danach ist Alice an der Reihe.



---

## Sequenzdiagramm



---

## Wichtige Datenflüsse

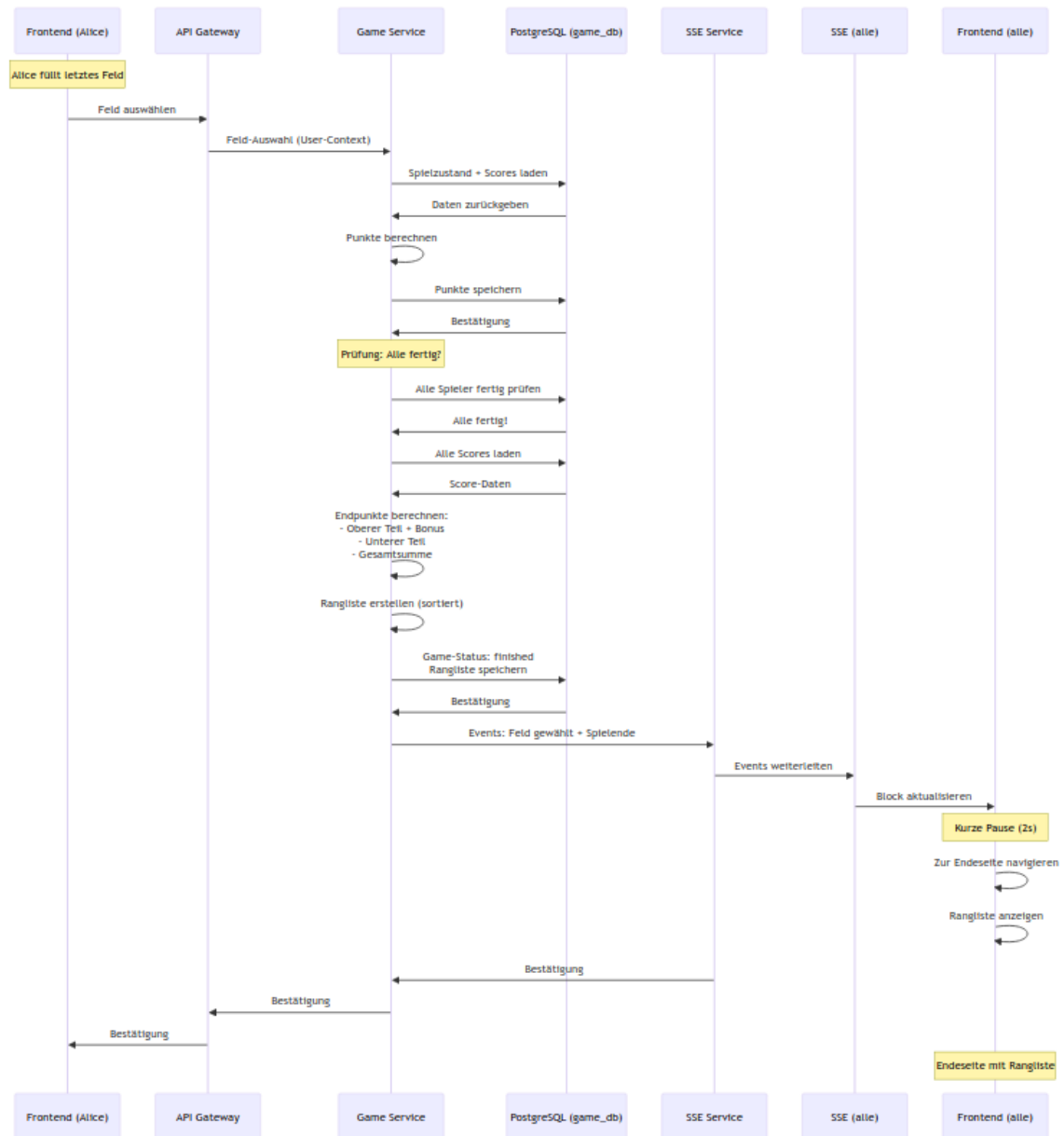
- Würfeln: Zufallswerte → Persistierung → Broadcasting
  - Fixieren: Würfel-Auswahl → Persistierung → Broadcasting
  - Punkteberechnung: Würfel + Feld → Punkte
  - Zugwechsel: Spieler-Index → Nächster Spieler
  - Timer-Reset bei jeder Interaktion
- 

## Flow 5: Spielende erkennen & Rangliste

### Szenario

Alice füllt ihr letztes (13.) Feld aus. Der Game Service erkennt, dass alle Spieler fertig sind, berechnet die Endpunkte und erstellt eine Rangliste.

## Sequenzdiagramm



## Wichtige Datenflüsse

- Letzte Feldwahl → Punkteberechnung
- Prüfung: Alle Spieler fertig?
- Endpunkte-Berechnung (Bonus-Logik)
- Rangliste erstellen (sortiert)

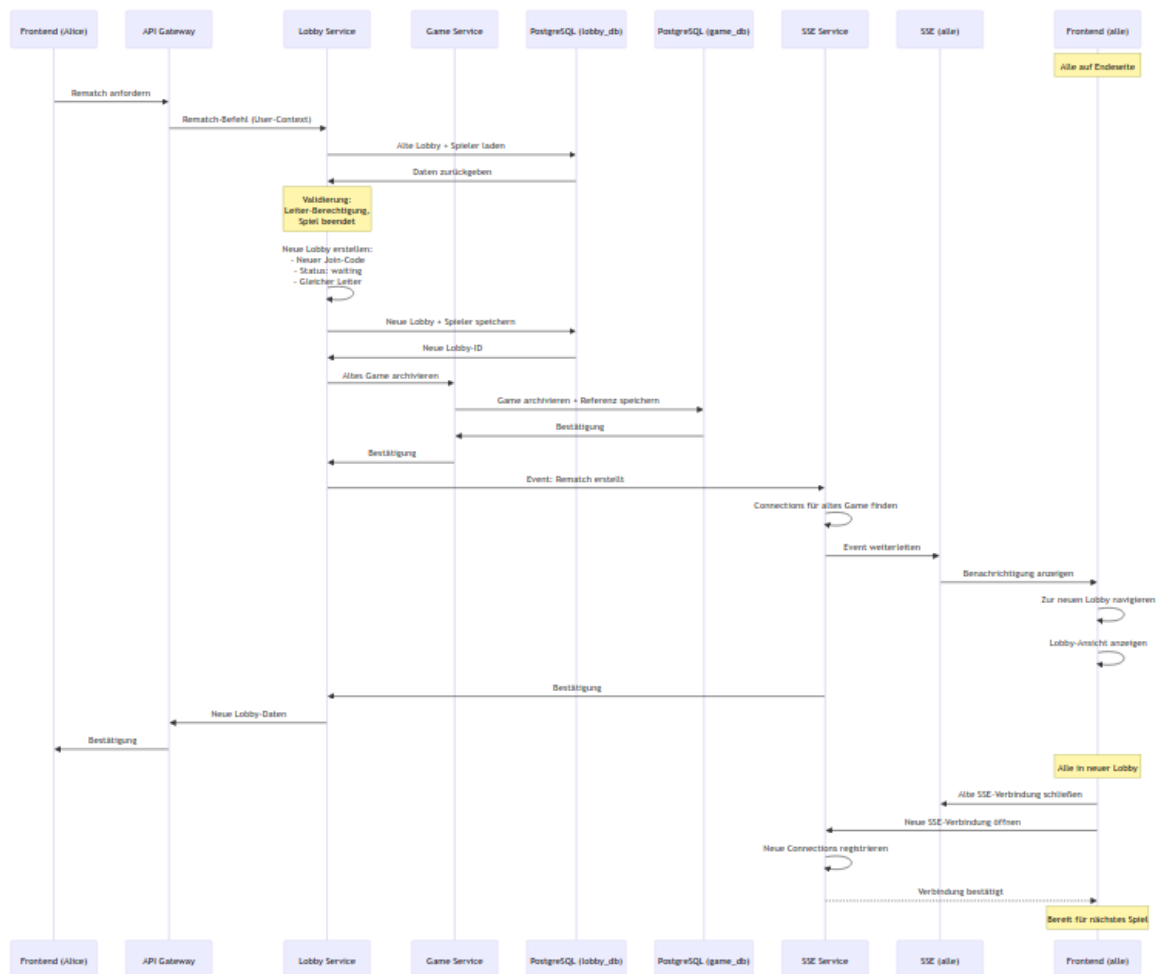
- Game-Status: running → finished
- Navigation aller Spieler zur Endeseite

## Flow 6: Nochmal spielen / Neue Runde

### Szenario

Die Spieler sind auf der Endeseite. Alice (Lobby-Leiter) klickt auf “Nochmal spielen”. Eine neue Lobby wird mit denselben Spielern erstellt.

### Sequenzdiagramm



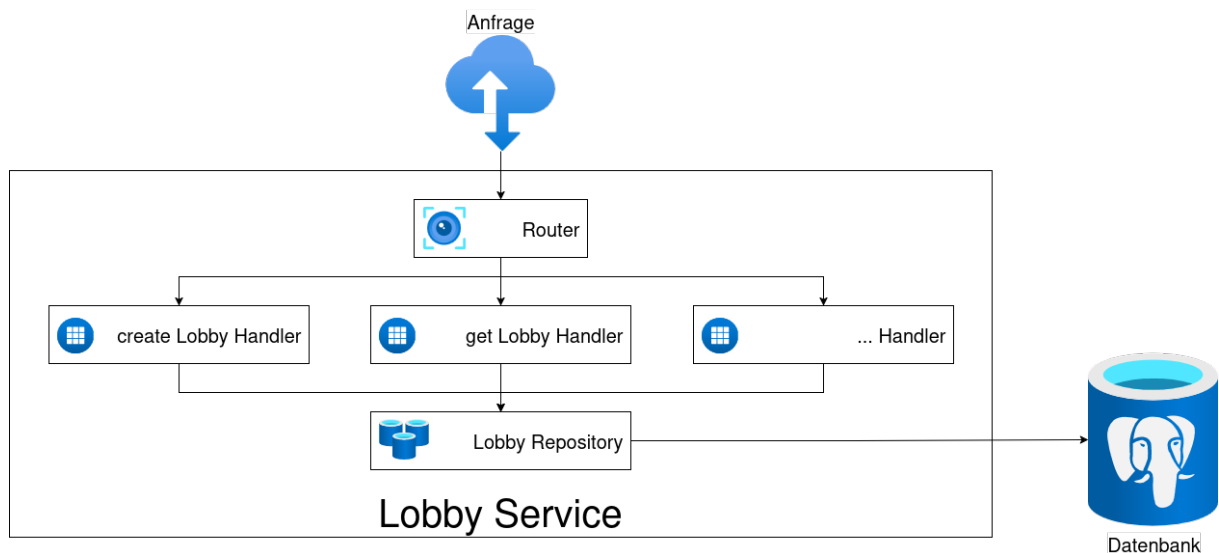
---

## Wichtige Datenflüsse

- Rematch-Berechtigung → Validierung (nur original Leiter)
- Neue Lobby-Erstellung (neuer Join-Code)
- Spieler-Übernahme (automatisch)
- Altes Game → Archivierung
- Event → Alle Spieler
- SSE-Verbindungen wechseln (alt → neu)

## Architektursichten

### 3-Schichtenarchitektur



Die Kernservices (Lobby Service und Game Service) sind nach dem 3-Schichtenarchitektur-Muster aufgebaut.

Die Services enthalten einen Router, der die eingehenden HTTP-Anfragen entgegennimmt und an die entsprechenden Handler weiterleitet. Der Router stellt somit die Präsentationsschicht dar.

Die Handler implementieren die Anwendungslogik.

Die Datenzugriffsschicht wird durch Repositorys und Models realisiert, die den Zugriff auf die Datenbanken kapseln und feste Interfaces definieren.

---

## Codebeispiel: Router

```
1 // New constructs the HTTP router with repository and join code
   ↳ generator dependencies
2 func New(repo repository.Repository, codeGen *joincode.Generator)
   ↳ http.Handler {
3     r := chi.NewRouter()
4     // replace chi default logger with structured slog based
       ↳ middleware
5     l := logger.Default()
6     r.Use(logger.ChiMiddleware(l))
7
8     // Healthcheck
9     healthcheck.Mount(r)
10
11    // Internal endpoints (no auth required)
12    r.Route("/internal", func(r chi.Router) {
13        r.Route("/lobbies", func(r chi.Router) {
14            r.Put("/{lobby_id}/players/{player_id}/active",
   ↳ handlers.UpdatePlayerActiveStatusHandler(repo))
15            r.Get("/{lobby_id}",
   ↳ handlers.GetLobbyInternalHandler(repo))
16        })
17    })
18
19    // Lobby endpoints grouped under auth middleware
20    r.Route("/lobbies", func(r chi.Router) {
21        // Authentication middleware (reads X-User-ID / X-Username
       ↳ and injects user into context)
22        r.Use(auth.AuthMiddleware)
23
24        // Create lobby (any authenticated user)
25        r.Post("/", handlers.CreateLobbyHandler(repo, codeGen))
26
27        // Join lobby (any authenticated user)
28        r.Post("/join", handlers.JoinLobbyHandler(repo))
29
30        // Get lobby details - require membership
```

---

```

31         r.With(handlers.RequireLobbyMember(repo)).Get("/{lobby_id}",
↪ handlers.GetLobbyHandler(repo))
32
33         // Kick player - require leadership
34
↪ r.With(handlers.RequireLobbyLeader(repo)).Post("/{lobby_id}/kick",
↪ handlers.KickPlayerHandler(repo))
35
36         // Other lobby routes can use RequireLobbyMember or
↪ RequireLobbyLeader as appropriate
37         // e.g.
↪ r.With(handlers.RequireLobbyLeader(repo)).Post("/{lobby_id}/start",
↪ handlers.StartLobbyHandler(db))
38     })
39
40     return r
41 }

```

### Codebeispiel: Handler

```

1 // GetLobbyInternalHandler returns an http.HandlerFunc that retrieves
↪ lobby details for internal use
2 // Path parameter: lobby_id (UUID)
3 // Returns lobby details with all players, no authentication or
↪ authorization required
4 func GetLobbyInternalHandler(repo repository.Repository)
↪ http.HandlerFunc {
5     return func(w http.ResponseWriter, r *http.Request) {
6         log := log-
↪ ger.Logger(r.Context()).WithGroup("handler").With(slog.String("action",
↪ "get_lobby_internal"))
7
8         // Extract lobby_id from URL path parameter
9         lobbyIDStr := chi.URLParam(r, "lobby_id")
10        if lobbyIDStr == "" {
11            log.Warn("missing lobby_id parameter")
12            httpx.WriteBadRequest(w, "Missing lobby_id parameter",
↪ nil, log)

```



---

```

13         return
14     }
15
16     lobbyID, err := uuid.Parse(lobbyIDStr)
17     if err != nil {
18         log.Warn("invalid lobby_id format",
19 ↪ slog.String("lobby_id", lobbyIDStr), slog.String("error",
20 ↪ err.Error()))
21         httpx.WriteBadRequest(w, "Invalid lobby ID format",
22 ↪ map[string]interface{}{"detail": err.Error()}, log)
23         return
24     }
25
26     // Query lobby via repository
27     response, err := repo.GetLobbyDetail(r.Context(), lobbyID)
28     if err == sql.ErrNoRows {
29         log.Info("lobby not found", slog.String("lobby_id",
30 ↪ lobbyID.String()))
31         httpx.WriteNotFound(w, "Lobby not found", log)
32         return
33     }
34     if err != nil {
35         log.Error("failed to query lobby", slog.String("error",
36 ↪ err.Error()), slog.String("lobby_id", lobbyID.String()))
37         httpx.WriteInternalServerError(w, "Database error", nil, log)
38         return
39     }
40
41     log.Info("lobby details retrieved",
42         slog.String("lobby_id", lobbyID.String()),
43         slog.Int("player_count", len(response.Players)))
44
45     httpx.WriteJSON(w, http.StatusOK, response, log)
46 }
47
48 }
```

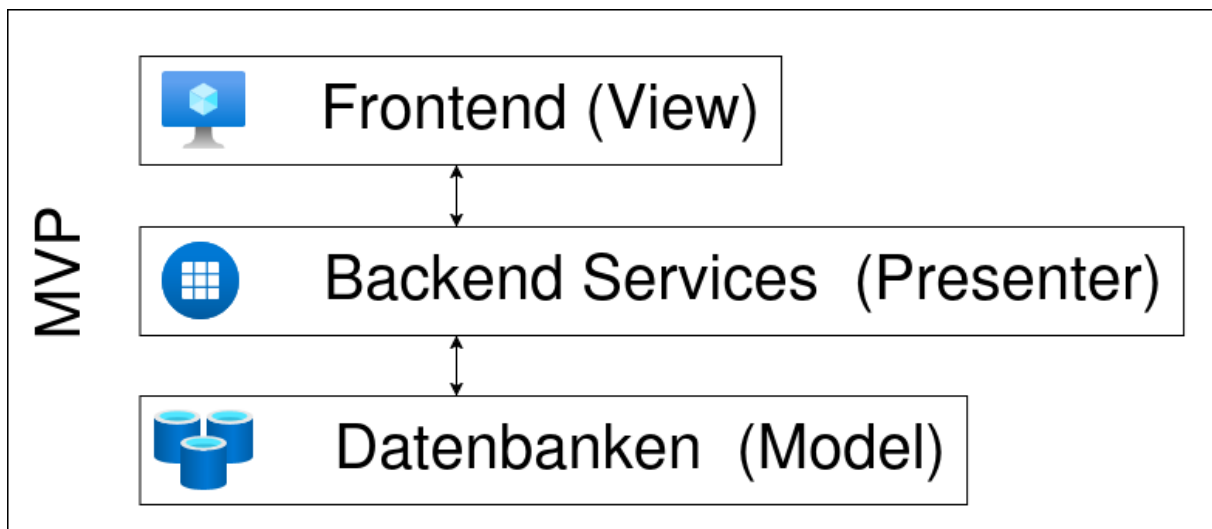
---

---

### Codebeispiel: Repository

```
1 func (r *PostgresRepository) CreateLobbyTx(tx *sql.Tx, joinCode
  ↳ string, leaderID uuid.UUID) (uuid.UUID, error) {
2     var lobbyID uuid.UUID
3     if err := tx.QueryRow(`
4         INSERT INTO lobbies (join_code, leader_id, status)
5         VALUES ($1, $2, $3)
6         RETURNING id
7     `, joinCode, leaderID, models.LobbyStatusWaiting).Scan(&lobbyID);
  ↳ err != nil {
8         return uuid.Nil, err
9     }
10    return lobbyID, nil
11 }
```

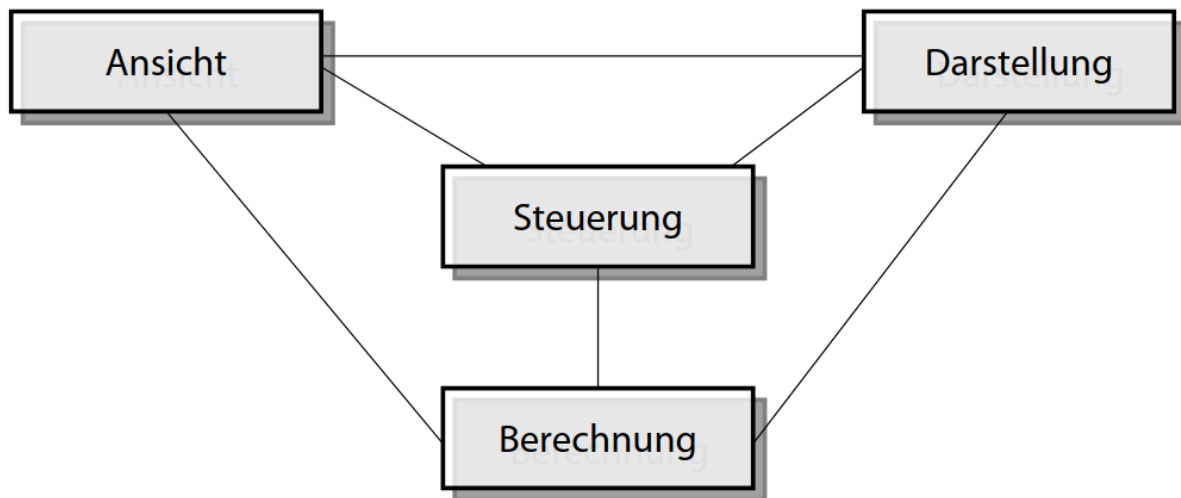
### Model View Presenter (MVP)



Die Anwendung lässt sich auch in das Model-View-Presenter (MVP) Muster einordnen. - **View**: Das Frontend (React SPA) stellt die View dar. Es ist verantwortlich für die Darstellung der Benutzeroberfläche und die Interaktion mit dem Benutzer. - **Presenter**: Das API Gateway fungiert als Presenter. Es vermittelt zwischen der View (Frontend) und dem Model (Backend Services). Es verarbeitet die Anfragen der View, ruft die entsprechenden Backend Services auf und formatiert die Antworten für die View. - **Model**: Die Backend Services repräsentieren das Model. Sie enthalten die Geschäftslogik und den Datenzugriff.

---

## Standardmodell für Webanwendungen



- **Ansicht:** Frontend (React SPA)
- **Darstellung:** API Gateway
- **Steuerung:** Backend Services
- **Berechnung:** Backend Services

## Serviceorientierte Architektur (SOA)

Die Systemarchitektur ist in Form von Microservices gestaltet, die jeweils eine spezifische Domäne abdecken (z.B. Lobby Management, Spiel-Logik, Authentifizierung). Jeder Dienst ist unabhängig deploybar und skalierbar, was eine hohe Flexibilität und Wartbarkeit ermöglicht. Die Dienste kommunizieren über klar definierte Schnittstellen (REST-APIs, SSE), was die lose Kopplung zwischen den Komponenten fördert.

- **Interaction Services:** Frontend (React SPA), SSE Service – stellen die Schnittstelle zum Benutzer dar und ermöglichen Echtzeitkommunikation
- **Processing Services:** API Gateway, Auth Service, Lobby Service, Game Service – implementieren die Geschäftslogik und verarbeiten Anfragen
- **Information Services:** Lobby Service, Game Service – verwalten und persistieren domänenspezifische Daten (Lobby-Zustände, Spielstände)
- **Access Services:** PostgreSQL Datenbanken, Auth Service (Token-Verwaltung) – kontrollieren den Zugriff auf Datenspeicher und Autorisierung