Projektdokumentation: Himues Darts Scoreboard V1.2

# Projektübersicht

Das Projekt dient als Scoreboard für Autodarts. Es wurde entwickelt, um den aktuellen Spielstand auf einem TV-Browser darzustellen und somit allen Mitspielern einen aktuellen Überblick zu geben, ohne aufs kleine Tablet blicken zu müssen. Außerdem sollen wirklich nur die Spielrelevanten Daten angezeigt werden (warum müssen Spieler das Bild der Dartscheibe oder viele – für sie unnütze – Buttons sehen?).

Das Projekt unterstützt die Spielmodi X01, Cricket, Tactics, Bermuda, Shanghai, Gotcha, Around the Clock, Round the World, Count Up, Random Checkout und Segment Training. Zusätzlich wird die Bull-off-Phase vor X01-Spielen als eigener, temporärer Modus visualisiert.

Es besteht aus zwei Hauptkomponenten:

* **Backend:** Ein modularer Python-Dienst, der sich mit den Autodarts-Servern verbindet, Live-Spieldaten empfängt und diese für die unterstützten Spielmodi verarbeitet. Er reichert die Daten bei Bedarf mit Informationen aus einer eigenen MariaDB-Datenbank an , um langfristige Spielerstatistiken zu erstellen.
* **Frontend:** Eine Web-Anwendung, die als Brücke (Bridge) dient. Sie empfängt aufbereitete Events vom Backend und stellt sie als voll funktionsfähiges, dynamisches Scoreboard in einem Webbrowser dar, das sich automatisch an den jeweiligen Spielmodus anpasst.

Das Kern-Feature des Projekts ist die modulare Architektur und ein robustes, erweiterbares System zur Verwaltung von Spieler-Statistiken (Average, MPR, Hit-Rate, etc.), inklusive eines In-Memory-Caches zur Reduzierung von Datenbankabfragen. Die Nutzung der Datenbank ist optional und kann in der Konfiguration deaktiviert werden.

Beide Dienste sind technologisch auf gevent vereinheitlicht und für den Betrieb mit Gunicorn optimiert.

Die komplette Trennung zwischen Backend und Frontend ist eine bewusste Entscheidung. Dadurch ist es möglich das Frontend gegen ein anderes auszutauschen ohne Änderungen am Backend vornehmen zu müssen.

Ein Beispiel für ein alternatives Frontend findet sich im Kapitel [Beispielanwendung: Das Standalone CMD-Dashboard](#_Beispielanwendung:_Das_Standalone). Das zugehörige Programm ist installiert und kann im Ordner frontend\_cmd mit dem Befehl start-prod.py gestartet werden.

# Installationshinweis

Es schadet zwar nicht erst die Dokumentation zu lesen, aber um direkt anzufangen kann das Installations-Script mit

python3 install.py

aufgerufen werden.

Es sollte **NICHT als root** ausgeführt werden. Da für den Betrieb keine besonderen Rechte erforderlich sind ist ein unprivilegierter User mit sudo-Rechten ausreichend.

sudo wird benötigt um:

* das python3-venv Paket zu installieren (falls es nicht vorhanden ist)
* das libmariadb-dev Paket zu installieren (falls es nicht vorhanden ist)
* die Pakete 'build-essential' und 'python3-dev' zu installieren (falls sie nicht vorhanden ist)
* den Linger-Modus zu aktivieren um die Dienste später mit „systemctl –user“ im User-Kontext installieren, starten und stoppen zu können

Das Installations-Script sorgt dafür:

* dass die Python venv-Umgebung eingerichtet wird,
* die Pakete aus requirements.txt
* sowie einige notwendige weitere bei Bedarf installiert werden
* auf Wunsch die Verbindung zur maridb hergestellt und die Datenbank mit den Tabellen angelegt wird
* Die Boarddaten abgefragt und gespeichert werden
* Die Autodarts-Keycloack-Dateien abgefragt und gespeichert werden (wenn diese nicht zur Verfügung stehen, kann Himues Darts Scoreboard aktuell leider nicht verwendet werden)
* und das Backend und Frontend als systemd-Dienste eingerichtet werden.

Danach wir dein Reboot empfohlen und nach kurzer Zeit sollte das Frontend unter https://<IP>:6002 zur Verfügung stehen.

Vorlagen zur Erreichbarkeit über einen NGINX-Proxy sind im Kapitel [NGINX-Konfiguration:](#_NGINX-Konfiguration:) zu finden.

# Architektur

Die Systemarchitektur ist auf eine klare Trennung von Logik und Darstellung ausgelegt, dem Prinzip eines "Smart Backend" und eines "Dumb Frontend".

Ein Bild, das Text, Screenshot, Software, Design enthält.

KI-generierte Inhalte können fehlerhaft sein.

***(Mermaid-Sytax. Kann z.B. mit*** [***https://app.diagrams.net/***](https://app.diagrams.net/) ***dargestellt werden)***

sequenceDiagram

participant Browser as Browser-Client (JS)

participant Bridge as Frontend-Bridge (Python)

participant Backend as Backend / Hub (Python)

participant Autodarts as Autodarts Server

participant DB as MariaDB

Autodarts-->>Backend: 1. Authentifizierung & WebSocket-Verbindung

Backend-->>DB: (Optional) 2. Lade Spieler-Statistiken (Cache)

Note over Autodarts, Backend: Ein Match beginnt...

Autodarts->>Backend: 3. 'start'-Event

Backend->>Autodarts: 4. "Kickstart" (leerer PATCH-Request)

Autodarts->>Backend: 5. Vollständiges 'game-update' (Live-Daten)

loop Jeder Wurf

Autodarts->>Backend: 6. 'game-update' (Live-Daten)

Backend->>Backend: 7. Daten verarbeiten &<br/>mit Statistiken anreichern

Backend->>DB: (Optional) Statistiken am Leg-Ende speichern

Backend->>Bridge: 8. Sendet aufbereitetes 'GameEvent'

Bridge->>Browser: 9. Leitet Event als 'status\_update' weiter

Browser->>Browser: 10. Aktualisiert die UI basierend auf den Event-Daten

end

## "Cheat Sheet" für Schlüsselkonzepte

|  |  |
| --- | --- |
| Konzept | Kurzbeschreibung |
| Spieler-Reihenfolge | Backend "latcht" eine feste display\_order, um die rotierende Server-Reihenfolge für eine stabile UI zu kompensieren. |
| Match-Start | Ein leerer PATCH-Request ("Kickstart") an die Autodarts-API erzwingt die Sendung vollständiger Startdaten. |
| Event-Struktur | Alle Backend-Frontend-Nachrichten folgen einer einheitlichen GameEvent-Struktur für maximale Vorhersagbarkeit. |
| Kompatibilität | Serverseitige Logik wählt zwischen Canvas-Animation (fireworks.js) und Video-Fallback für ältere Browser. |
| Statistik-Engine | Ein modulares System speichert Leg-Daten in spielmodus-spezifischen Tabellen und berechnet daraus langfristige Statistiken. |
| Datenverarbeitung | Eine universelle Funktion (create\_universal\_game\_event) erstellt ein generisches Event, das von Spiel-Modulen "veredelt" wird. |

## Backend (app.py)

Das Backend fungiert als zentrale Steuerungseinheit ("Dirigent"). Es ist in vier logische Hauptpakete unterteilt: core, autodarts, spiellogik und statistik.

* **Verbindung:** Authentifiziert sich bei Autodarts und baut eine permanente WebSocket-Verbindung auf, um Live-Events zu abonnieren.
* **Datenverarbeitung:** Eingehende Live-Daten werden von einem zentralen Dispatcher (websocket\_handlers.py) an das zuständige Spielmodul (z.B. process\_match\_x01.py) weitergeleitet.
* **Datenbank-Interaktion:** Kommuniziert mit einer MariaDB-Datenbank, um Averages zu lesen, zu speichern und zu berechnen.
* **Broadcasting:** Sendet standardisierte GameEvent-Objekte an die Frontend-Brücke.
* **Debugging:** Stellt zwei Webseiten zur Live-Analyse zur Verfügung:
  + **/debug:** Zeigt alle Events/Daten, die vom Backend an das Frontend gesendet werden.
  + **/debugad:** Zeigt alle Rohdaten, die vom Autodarts-Server empfangen werden.
* **Stabilität:**
  + Verwendet eine benutzerdefinierte Gunicorn-Worker-Klasse, um harmlose SSL-Verbindungsfehler auf der Konsole zu unterdrücken.
  + Die Logik zur **Stabilisierung der Spielerreihenfolge** wurde von X01 auf alle relevanten Spielmodi ausgeweitet, um eine konsistente Anzeige zu gewährleisten.
  + Die **Datenverarbeitung** wurde weiter zentralisiert. Eine universelle Hilfsfunktion (create\_universal\_game\_event in match\_handler.py) erstellt bei jedem Wurf ein generisches, vollständiges Event-Objekt. Die spielspezifischen Module (z.B. process\_match\_x01.py) sind nun primär dafür zuständig, dieses generische Event bei Bedarf mit der Logik und den Daten ihres jeweiligen Spielmodus zu "veredeln" oder zu modifizieren.
  + Wenn während eines laufenden Spiels das Backend nicht funktionieren sollte (Absturz, manueller, Neustart, etc.) fragt bei einem Neustart das Backend die Autodarts-API nach allen potenziell aktiven Spielen. Es identifiziert dann intelligent das neueste, noch nicht beendete Spiel, um sicherzustellen, dass immer der korrekte Spielstand wiederhergestellt wird und keine "Geister-Spiele" geladen werden (es ist vorgekommen, dass die Autodarts-API mehr als ein Spiel geliefert hat, die zum Teil schon vpr Stunden gestartet wurden aber nicht wirklich aktiv waren).

## Frontend-Brücke (app.py)

Diese Komponente dient als einfache, aber wichtige Brücke zwischen dem Backend und den Web-Browsern. Sie wurde auf gevent umgestellt, um technologisch konsistent mit dem Backend zu sein.

* **Empfang:** Verbindet sich mit dem WebSocket des Backends und lauscht auf dessen Events.
* **Weiterleitung:** Leitet alle empfangenen Events unter dem einheitlichen Namen status\_update an alle verbundenen Browser weiter.
* **Zustandsspeicherung:** Hält eine Kopie des letzten status\_update-Events, um neu verbindenden Browsern sofort den aktuellen Spielstand senden zu können.
* **Verbindungs-Management:** Der Frontend-Server besitzt eine robuste, eigene Logik, um die Verbindung zum Backend hartnäckig wiederherzustellen. Er merkt sich den Verbindungsstatus (verbunden/nicht verbunden) und informiert die Browser-Clients aktiv über den aktuellen Zustand mit den Events backend\_connected und backend\_disconnected. Neue Browser erhalten sofort den korrekten Status, was Timing-Probleme beim Seitenaufbau verhindert.

### Scoreboard.js und scoreboard\_helpers.js

Ursprünglich gab es nur die scoreboard.js

Um die Lesbarkeit und Wartbarkeit des Frontends zu verbessern, wurde die Kern-JavaScript-Datei scoreboard.js aufgeteilt. Sie folgt nun einer klareren Trennung von Verantwortlichkeiten:

* **scoreboard.js (Der Controller):** Diese Datei ist nun der reine **Controller**. Ihre Hauptaufgaben sind:
  + Die Verwaltung der globalen State-Variable appState.
  + Der Aufbau und die Verwaltung der Socket.IO-Verbindung zum Frontend-Server.
  + Das Empfangen von status\_update-Events und das Anstoßen des Update-Prozesses.
* **scoreboard\_helpers.js (Die View-Logik):** Diese neue Datei bündelt die gesamte Logik, die sich direkt auf die **Darstellung und UI-Interaktion** bezieht. Dazu gehören:
  + Das Setup des Startbildschirms (\_setupInitialScreen).
  + Die Verarbeitung und Anzeige von Benachrichtigungen und Overlays (\_processGameNotifications und untergeordnete \_display...-Funktionen).
  + Die Verwaltung der Zustandsvariablen, die nur die UI betreffen, wie isDisplayActive und die fireworksInstance.
  + Das Routing an die korrekte, spielmodus-spezifische View (\_routeToGameViewUpdater).

Durch diese Aufteilung ist der Code modularer und Zuständigkeiten sind klarer voneinander abgegrenzt.

## Frontend-Anzeige (Single-Page-App)

* **Zentrales Layout:** Die scoreboard.html enthält den globalen Rahmen sowie den oberen "Fokus-Bereich", der für alle Spielmodi geteilt wird. Spielspezifische Anzeigeelemente (Tabellen, Karten) sind in \_view\_\*.html-Partials ausgelagert.

scoreboard.html enthält den globalen Rahmen. Der obere **Fokus-Bereich** wurde als **3-spaltige Struktur** definiert, um neben dem Haupt-Score/Ziel auch Platz für dynamische Grafiken zu schaffen.

* **ViewModel-Pattern:** scoreboard.js agiert als Controller, der auf status\_update-Events lauscht und die \_routeToGameViewUpdater-Funktion aufruft. \_routeToGameViewUpdater ruft die zuständige "intelligente" update...View-Funktion auf. Diese bereitet die Daten in einem FocusAreaViewModel-Objekt vor und übergibt es an eine "dumme" Rendering-Funktion, die nur noch den DOM aktualisiert.
* **Event-Handling:** scoreboard.js agiert als Controller, der auf status\_update-Events lauscht und die \_routeToGameViewUpdater-Funktion aufruft.
* **Dynamisches Rendering:** Die \_routeToGameViewUpdater-Funktion ist ein "Router", der basierend auf dem Spielmodus die zuständige update...View-Funktion aufruft. Diese wiederum nutzt <template>-Tags, um die Anzeige effizient und fehlerfrei mit den neuen Daten zu aktualisieren.
* **Dynamische Grafiken:** Für den Modus "Segment Training" wurde eine dynamische SVG-Grafik implementiert, die das Zielsegment in korrekter Farbe und Rotation anzeigt und das zu treffende Feld durch eine pulsierende Animation hervorhebt.
* **Dynamischer Anzeigemodus für X01/Gotcha**: Für die Spielmodi X01 und Gotcha kann die Anzeige im unteren Bereich zwischen einer detaillierten Tabelle (table) und einer kompakteren Kartenansicht (cards) umgeschaltet werden. Die Standardeinstellung wird im Frontend über die Variable SHOW\_PLAYER\_CARD in der config\_frontend.py festgelegt.
  + True = Kartenansicht für den Player
  + Fasle = Tabellenansicht

Um die Ansicht flexibel und ohne Backend-Änderung wechseln zu können (insbesondere auf TV-Browsern, wo der Zugriff auf Konfigurationsdateien umständlich ist), kann die Anzeige direkt über einen URL-Parameter gesteuert werden. Durch Anhängen von ?xc an die URL des Scoreboards wird die Kartenansicht erzwungen, unabhängig von der Backend-Konfiguration.

* /?xc = Kartenansicht
* /?xt = Tabellenansicht  
    
  **Beispiel:** http://<deine-ip>:6002/?xc

## Datenbank (MariaDB)

Die Datenbank sorgt für die Persistenz der Spielerdaten über einzelne X01-Matches hinaus.

* **players Tabelle:** Speichert die Stammdaten der Spieler, inklusive des Namens, des berechneten Gesamt-Averages und eines Flags (is\_registered), um zwischen Gästen und registrierten Nutzern zu unterscheiden.
* **games\_history Tabelle:** Protokolliert die detaillierten Statistiken (Punkte, Darts, Average) von jedem einzelnen gespielten Leg zur späteren Neuberechnung der Gast-Averages.
* **Der Gesamt-**Average wird bei Gästen vom Backend selbst berechnet. Beim Board-Owner und bei registrierten Benutzern wird der vom Autodarts-Server gelieferte Gesamt-Average in der Datenbank gespeichert.

## Kompatibilitätsmodus für Animationen (Feuerwerk)

Um bei einem Match-Gewinn eine visuelle Rückmeldung zu geben, wurde eine Feuerwerk-Animation implementiert. Da moderne JavaScript-Animationen (basierend auf HTML5 Canvas) nicht von allen Endgeräten, insbesondere von älteren Smart-TV-Browsern, unterstützt werden, wurde ein flexibler Kompatibilitätsmodus geschaffen.

Das System kann zwischen zwei Anzeigemethoden umschalten:

1. **JavaScript-Animation (fireworks.js):** Eine performante und optisch ansprechende Canvas-basierte Animation. Dies ist die Standardmethode für moderne Browser.
2. **Video-Fallback (fireworks.mp4):** Eine einfache Video-Datei, die maximale Kompatibilität gewährleistet und auch auf leistungsschwächeren Geräten oder Browsern mit eingeschränkter JavaScript-Unterstützung zuverlässig funktioniert.

**Logik und Steuerung**

Die Entscheidung, welche Methode verwendet wird, trifft der **Python-Server (himues\_scoreboard.py)** beim initialen Laden der Seite. Dies stellt sicher, dass die ressourcenintensive fireworks.js-Bibliothek nur dann an den Browser gesendet und geladen wird, wenn sie auch tatsächlich unterstützt wird, um potenzielle Browser-Abstürze zu verhindern.

Die Logik basiert auf zwei Konfigurationsvariablen in der Frontend/config.py:

* **SHOW\_ONLY\_FIREWORK\_VIDEO (Boolean):**
* Wenn dieser Wert auf True gesetzt wird, wird **immer und für jeden Browser** das Video-Fallback verwendet. Dies dient als globaler Override.
* **BROWSER\_NAMES\_TO\_SHOW\_ONLY\_VIDEO (Liste von Strings):**
* Diese Liste enthält Schlüsselwörter (z.B. "SamsungBrowser"). Der Server prüft beim Seitenaufruf den User-Agent-String des anfragenden Browsers.
* Wird eines der Schlüsselwörter aus der Liste im User Agent gefunden, schaltet das System für diesen Client automatisch auf das Video-Fallback um.

Der Server übergibt seine Entscheidung als boolesche Variable (use\_video\_fireworks) an das scoreboard.html-Template. Dort wird mittels einer Jinja2-Bedingung ({% if not use\_video\_fireworks %}) entschieden, ob der <script>-Tag für fireworks.js in das HTML eingefügt wird oder nicht.

Das Frontend-JavaScript (scoreboard.js) liest diese Variable ebenfalls aus und startet bei einem Match-Gewinn die entsprechend korrekte Animation.

## Grafische Ziel-Anzeige (Segmente & Ringe)

Für Spielmodi, bei denen ein spezifisches Segment oder ein Ring das Ziel ist, wurde eine dynamische, visuelle Unterstützung im Fokus-Bereich implementiert. Diese zeigt dem Spieler grafisch das aktuell zu treffende Ziel an und hebt den geforderten Bereich durch eine Animation hervor.

### Technische Umsetzung

Die gesamte Logik zur Erzeugung der Grafiken ist in der Datei Frontend/static/js/utils\_frontend.js zentralisiert. Die Grafiken werden dynamisch als SVG-Elemente direkt im Browser erzeugt, was eine hohe Flexibilität gewährleistet. Es gibt zwei Hauptfunktionen:

1. **drawTargetSegment(targetSegment, mode)**

* Diese Funktion ist für die Darstellung eines einzelnen, spezifischen Dart-Segments (z.B. "20", "19", "Bull") zuständig.
* Sie berechnet die exakten Pfade für die vier Teile eines Segments (Inner-Single, Triple, Outer-Single, Double) basierend auf Standard-Dartboard-Geometrie.
* Die korrekte Farbgebung (rot/grün für Wertungsfelder, schwarz/weiß für Single-Felder) wird anhand einer vordefinierten Liste ermittelt.
* Die Grafik wird mittels einer SVG-Transformation (transform="rotate(...)") in die korrekte Position auf der "Scheibe" gedreht.
* Basierend auf dem mode-Parameter (Single, Double, Triple, Full) wird eine gelbe, pulsierende blinking-overlay über die entsprechenden Teile des Segments gelegt.

1. **drawTargetRings(mode)**

* Diese Spezialfunktion wird für den Spielmodus "Bermuda" verwendet, wenn das Ziel ein allgemeiner Ring ist (z.B. "beliebiges Double").
* Sie erzeugt zwei konzentrische, dicke Ringe, die durch eine stroke-dasharray-Technik in die 20 abwechselnd roten und grünen Segmente unterteilt werden.
* Je nach mode (Double oder Triple) wird ein passender gelber, pulsierender Highlight-Ring über den entsprechenden farbigen Ring gelegt.

### Anpassbarkeit und Styling

Ein zentrales Merkmal ist die vollständige Konfigurierbarkeit des Erscheinungsbildes über CSS. Alle relevanten Stil- und Größenangaben sind als **CSS Custom Properties (Variablen)** in der Datei Frontend/static/css/style.css unter der Regel #segment-training-graphic-container gebündelt.

Dies umfasst unter anderem:

* --segment-draw-size: Die interne Zeichengröße für die Segmente.
* --double-ring-radius, --triple-ring-radius: Die Radien für die Bermuda-Ringe.
* --segment-color-\*, --bull-color-\*: Alle Farbdefinitionen.

Durch die Anpassung dieser Variablen kann das Aussehen der Grafiken verändert werden, ohne dass Änderungen am JavaScript-Code notwendig sind.

**Einbindung in die Spielmodi**

Die jeweilige view\_\*.js-Datei eines Spielmodus (z.B. view\_shanghai.js) ist dafür verantwortlich, die korrekte Zeichenfunktion (drawTargetSegment oder drawTargetRings) mit den Zieldaten aus dem vom Server empfangenen Event aufzurufen.

# Langzeit-Statistiken und Datenbank

Ein Kernmerkmal des Backends ist die Fähigkeit, detaillierte Spielerstatistiken über einzelne Matches hinaus zu speichern und daraus langfristige Leistungsdaten zu berechnen. Die Nutzung der Datenbank ist über eine Einstellung in config.py optional.

## Architektur und Konzept

Das System ist darauf ausgelegt, für jeden Spielmodus, der eine aussagekräftige Kennzahl liefert, eine eigene Statistik zu führen.

* **Getrennte Tabellen:** Für jeden Spielmodus (X01, Cricket, Tactics, ATC, etc.) werden separate Tabellenpaare in der MariaDB-Datenbank angelegt (players\_\* und games\_history\_\*). Dies sorgt für eine saubere Datentrennung und Übersichtlichkeit.
* **Spieler-Typen:** Das System unterscheidet zwischen registrierten Spielern/Board-Ownern und Gästen. Bei **X01** wird für registrierte Spieler der offizielle Average vom Autodarts-Server übernommen. In **allen anderen Modi** sowie für Gäste in X01 berechnet das Backend die Statistik selbst.
* **Berechnung:** Die Berechnung für Gäste basiert immer auf den Rohdaten der **letzten 100 gespielten Legs** des jeweiligen Spielmodus, die in den games\_history\_\*-Tabellen gespeichert sind.

## Besonderheiten bei Spieler-Typen

Das System unterscheidet zwischen drei verschiedenen Spieler-Typen, um die Statistiken korrekt zu verwalten:

* **Board-Owner & Registrierte Spieler:** Bei Spielern, die bei Autodarts registriert sind oder als Board-Owner erkannt werden, wird der langfristige X01-Average direkt vom Autodarts-Server übernommen und in der Datenbank gespeichert. Dies stellt sicher, dass immer der offizielle, über alle Spiele hinweg berechnete Wert verwendet wird. Für Cricket und Tactics gibt es seitens Autodarts keine solche servergespeicherte Statistik; hier werden auch registrierte Spieler wie Gäste behandelt.
* **Gäste:** Bei nicht registrierten Spielern ("Gästen") berechnet das Backend den Gesamt-Average (für X01) bzw. den Gesamt-MPR (für Cricket/Tactics) selbst. Dazu werden die in den games\_history\_\*-Tabellen gespeicherten Daten der letzten 100 gespielten Legs herangezogen. Dies ermöglicht eine genaue und faire Leistungsbewertung auch für spontane Mitspieler.
* **Bei allen anderen Spielen außer X01:** Für einige Spiele (Cricket/Tactics **,** Around the World, Segment Training, Count Up liefert der Autodarts Server Statistiken für das aktuelle Leg/Match. Diese sind unabhängig davon, ob es sich um registrierte Spieler oder Gäste handelt.  
  Das Backend speichert die Werte für alle Spieler und berechnet langfristige Statistiken.

## Berechnungslogik der Statistiken

| Spielmodus | Statistik | Formel / Berechnung |
| --- | --- | --- |
| **X01** | Average | (SUM(leg\_points) / SUM(leg\_darts)) \* 3 |
| **Cricket / Tactics** | MPR | (SUM(leg\_marks) \* 3) / SUM(leg\_darts) |
| **ATC / Segment Training** | Hit Rate | AVG(leg\_hit\_rate) \* 100 (als %-Wert) |
| **Count Up** | PPR | (SUM(leg\_points) / SUM(leg\_darts)) \* 3 |

## 3.3 Tabellenstruktur (SQL)

SQL

-- Tabellen für X01

CREATE TABLE `players\_x01` ( ... );

CREATE TABLE `games\_history\_x01` ( ... );

-- Tabellen für Cricket

CREATE TABLE `players\_cricket` ( ... );

CREATE TABLE `games\_history\_cricket` ( ... );

-- Tabellen für Tactics

CREATE TABLE `players\_tactics` ( ... );

CREATE TABLE `games\_history\_tactics` ( ... );

-- Tabellen für ATC

CREATE TABLE `players\_atc` ( ... );

CREATE TABLE `games\_history\_atc` ( ... );

-- Tabellen für Count Up

CREATE TABLE `players\_countup` ( ... );

CREATE TABLE `games\_history\_countup` ( ... );

-- Tabellen für Segment Training

CREATE TABLE `players\_segment\_training` ( ... );

CREATE TABLE `games\_history\_segment\_training` ( ... );

*(Hinweis: Die vollständigen CREATE TABLE-Befehle befinden sich im Kapitel "Deployment und Betrieb".)*

# Dateistruktur des Projektes

## Backend

📁 Backend/

├── 📄 app\_backend.py

├── 📄 config.py

├── 📄 gunicorn.conf.py

├── 📄 .env

├── 📄 start-dev.sh

├── 📄 start-prod.sh

│

├── 📁 crt/

│ ├── 📄 dummy.crt

│ └── 📄 dummy.key

│

├── 📁 modules/

│ ├── 📄 \_\_init.py\_\_

│ ├── 📁 autodarts/

│ │ ├── 📄 autodarts\_keycloak\_client.py

│ │ ├── 📄 autodarts\_api\_client.py

│ │ ├── 📄 local\_board\_client.py

│ │ └── 📄 websocket\_handlers.py

│ ├── 📁 core/

│ │ ├── 📄 \_\_init\_\_.py

│ │ ├── 📄 app\_setup.py

│ │ ├── 📄 config\_loader.py

│ │ ├── 📄 constants.py

│ │ ├── 📄 database\_handler.py

│ │ ├── 📄 event\_structure.py

│ │ ├── 📄 shared\_state.py

│ │ ├── 📄 utils\_backend.py

│ │ ├── 📄 security\_module.py

│ │ └── 📄 webserver\_handler.py

│ ├── 📁 spiellogik/

│ │ ├── 📄 match\_handler.py

│ │ ├──📄 process\_match\_atc.py

│ │ ├──📄 process\_match\_bermuda.py

│ │ ├──📄 process\_match\_bob27.py

│ │ ├──📄 process\_match\_bull\_off.py

│ │ ├──📄 process\_match\_countup.py

│ │ ├──📄 process\_match\_cricket.py

│ │ ├──📄 process\_match\_gotcha.py

│ │ ├──📄 process\_match\_random\_checkout.py

│ │ ├──📄 process\_match\_rtw.py

│ │ ├──📄 process\_match\_segment\_training.py

│ │ ├──📄 process\_match\_shanghai.py

│ │ └──📄 process\_match\_x01.py

└── 📁 templates/

├── 📄 index.html

├── 📄 api\_index.html

├── 📄 debug\_state.html

├── 📄 debugadall.html

├── 📄 debugad.html

└── 📄 debug.html

## Frontend

📁 Frontend/

├── 📄 app\_frontend.py

├── 📄 config\_frontend.py

├── 📄 config\_gunicorn\_frontend.py

├── 📄 start-dev.sh

├── 📄 start-prod.sh

│

├── 📁 docs/

└── 📄 database\_schema.sql

├── 📁 modules/

│ └── 📁 core/

│ ├── 📄 app\_setup\_frontend.py

│ ├── 📄 config\_loader\_frontend.py

│ ├── 📄 shared\_state\_frontend.py

│ └── 📄 utils\_frontend.py

│

├── 📁 static/

│ ├── 📁 css/

│ │ └── 📄 style.css

│ ├── 📁 images/

│ │ ├── 📄 favicon.ico

│ │ ├── 📄 owner.png

│ │ ├── 📄 registered.png

│ │ └── 📄 Pfeil.svg

│ ├── 📁 js/

│ │ ├── 📄 fireworks.js

│ │ ├── 📄 scoreboard.js

│ │ ├── 📄 scoreboard\_helpers.js

│ │ ├── 📄 utils\_frontend.js

│ │ ├── 📄 show\_segment.js

│ │ └── 📁 spielmodi/

│ │ ├── 📄 view\_atc.js

│ │ ├── 📄 view\_bermuda.js

│ │ ├── 📄 view\_bobs27.js

│ │ ├── 📄 view\_bull\_off.js

│ │ ├── 📄 view\_countup.js

│ │ ├── 📄 view\_cricket.js

│ │ ├── 📄 view\_gotcha.js

│ │ ├── 📄 view\_random\_checkout.js

│ │ ├── 📄 view\_rtw.js

│ │ ├── 📄 view\_segment\_training.js

│ │ ├── 📄 view\_shanghai.js

│ │ └── 📄 view\_x01.js

│ └── 📁 videos/

│ └── 📄 fireworks.mp4│

└── 📁 templates/

├── 📄 scoreboard.html

└── 📁 partials/

├── 📄 \_view\_atc.html

├── 📄 \_view\_bermuda.html

├── 📄 \_view\_bob27.html

├── 📄 \_view\_bull\_off.html

├── 📄 \_view\_countup.html

├── 📄 \_view\_cricket.html

├── 📄 \_view\_gotcha.html

├── 📄 \_view\_random\_checkout.html

├── 📄 \_view\_rtw.html

├── 📄 \_view\_segment\_training.html

├── 📄 \_view\_shanghai.html

└── 📄 \_view\_x01.html

## Oberhalb von frontend/backend

📁 ./

├── 📄 setup.py

├── 📄 requirements.txt

├── 📄 release.py

├── 📄 README.md

└── 📄 Projektdokumentation Himues Darts Scoreboard V1.1.docx

# Struktur der Backend-Frontend-Nachrichten

Dank der GameEvent-Klasse haben alle Events, die vom Backend gesendet werden, eine einheitliche und vorhersagbare Struktur. Es gibt zwei Haupt-Event-Typen: match-started und game-update.

**Allgemeine Struktur:**

{

"event": "game-update",

"game\_state": "throw",

"current\_player\_index": 0,

"match": {

"game\_mode": "X01",

"legs\_to\_win": 2,

"sets\_to\_win": 0,

"max\_rounds": 50,

"start\_score": 121,

"in\_mode": "Straight",

"out\_mode": "Straight",

"scoring\_mode": null,

"targets": [],

"order": null,

"hits\_per\_target": 0,

"ends\_after\_type": "",

"ends\_after\_value": 0

},

"turn": {

"current\_round": 1,

"current\_leg": 1,

"current\_set": 1,

"target": null,

"throws": [],

"busted": false

},

"players": [

{

"name": "hallo",

"player\_type": "guest",

"display\_order": 0,

"score": 121,

"legs\_won": 0,

"sets\_won": 0,

"leg\_average": 0.0,

"match\_average": 0.0,

"overall\_average": 30.78,

"overall\_mpr": 2.87,

"overall\_hit\_rate":0.654,

"overall\_ppr": 60.11,

"mpr": 0.0,

"hits": {},

"darts\_thrown\_leg": 0,

"leg\_hit\_rate": 0.0,

"match\_hit\_rate": 0.0,

"current\_target": null

}

],

"winner\_info": {},

"checkout\_guide": []

}

Das Frontend kann sich darauf verlassen, dass alle Felder immer vorhanden sind, auch wenn sie leer sind (z.B. targets: [] bei X01).

# Analyse der Autodarts-Rohdaten

Das Backend empfängt zwei primäre Arten von Daten vom Autodarts-Server.

## Match-Start Daten (REST API)

Diese Daten werden einmalig bei Match-Start abgerufen und enthalten die statischen Grundinformationen des Spiels.

**Wichtige Felder:**

* **id**: Die eindeutige ID des Matches.
* **variant**: Die Spielvariante (z.B. "X01", "Cricket").
* **settings**: Ein Objekt, das die Spielregeln definiert.
* **players**: Eine Liste von Spieler-Objekten. Jedes Spieler-Objekt enthält:
  + name: Der Spielername.
  + user: Ein Unter-Objekt, das nur bei registrierten Spielern vorhanden ist und deren **langfristigen Gesamt-Average** (average) enthält.

**Beispiel-Datensatz (stark gekürzt):**

{  
 "id": "0198c660-d983-7157-a061-d4cffaa3f8d0",  
 "variant": "Cricket",  
 "settings": {  
 "gameMode": "Cricket",  
 "legs": 2  
 },  
 "players": [  
 {  
 "name": "christian",  
 "index": 0,  
 "user": null  
 },  
 {  
 "name": "corinna",  
 "index": 1,  
 "user": null  
 }  
 ]  
}

## Game-State Daten (WebSocket)

Nach jedem Wurf sendet der Server ein komplettes Update, das den gesamten aktuellen Spielzustand widerspiegelt.

**Wichtige Felder:**

* **player**: Der Index des Spielers, der aktuell am Zug ist
* **gameScores**: Liste der aktuellen Punktestände.
* **turns[0].throws**: Eine Liste der Darts in der aktuellen Aufnahme.
* **state.checkoutGuide**: (Nur X01) Eine Liste mit Checkout-Vorschlägen.
* **state.segments**: (Nur Cricket) Objekt mit der Anzahl der Treffer pro Spieler.
* **stats**: Eine Liste mit Live-Statistiken (average, mpr).
  + **legStats**: Statistiken für das **aktuelle Leg** (enthält average für X01 und mpr für Cricket).
  + **matchStats**: Kumulierte Statistiken für das **gesamte Match**.
* **gameFinished**: Wird true, wenn ein Leg beendet ist (Trigger für DB-Speicherung).
* **winner**: Enthält den Index des Match-Gewinners.
* **Beispiel-Datensatz (stark gekürzt):**

{  
 "id": "0198c7db-c74d-7ab1-a9ab-adee83dbeca0",  
 "player": 1,  
 "gameScores": [ 74, 82 ],  
 "turns": [  
 {  
 "throws": [  
 { "segment": { "name": "T20" } }  
 ]  
 }  
 ],  
 "state": {  
 "segments": {  
 "15": [ 3, 3 ],  
 "16": [ 3, 5 ],  
 "...": "..."  
 }  
 },  
 "stats": [  
 {  
 "legStats": { "mpr": 3.125 },  
 "matchStats": { "mpr": 3.125 }  
 },  
 {  
 "legStats": { "mpr": 3.166 },  
 "matchStats": { "mpr": 3.166 }  
 }  
 ],  
 "gameFinished": false,  
 "winner": -1  
}

## Klarstellung zur Spieler-Reihenfolge (wichtige Design-Entscheidung)

Die Analyse des Autodarts-Servers hat ein konsistentes, aber für die Anzeige herausforderndes System für die Spieler-Indizes offenbart:

* **Stabiler Index (index):** Jeder Spieler erhält zu Match-Beginn einen festen Index (z.B. 0, 1, 2). Dieser Index wird für match-übergreifende Daten wie den finalen **Match-Gewinner (winner)** verwendet.
* **Rotierende Reihenfolge:** Zu Beginn jedes neuen Legs **rotiert** der Server die Reihenfolge der Spieler in der players-Liste sowie in allen parallelen Daten-Arrays (stats, segments, etc.). Der Spieler, der das Leg beginnt, steht immer an Position 0.
* **Leg-Gewinner (gameWinner):** Der Index des Leg-Gewinners bezieht sich immer auf die Position in der **aktuell rotierten** players-Liste.

### Die Rolle des Backends: Anreicherung statt Sortierung

Um eine fehlerfreie Zuordnung der Daten (wie die des Match-Gewinners) zu ermöglichen und dem Frontend gleichzeitig volle Kontrolle über die Darstellung zu geben, wendet das Backend eine bewusste Strategie an:

1. **Backend sortiert nicht um:** Das Backend behält die vom Server gesendete, rotierende Reihenfolge in der players-Liste bei und reicht sie unverändert weiter.
2. **Anreicherung mit stabilem Anker:** Das Backend "merkt" sich die ursprüngliche Reihenfolge vom Spielstart. Bei jedem game-update fügt es jedem Spielerobjekt ein zusätzliches Feld hinzu: display\_order. Dieses Feld enthält für jeden Spieler seine ursprüngliche Startposition (z.B. 0, 1, 2).

### Die Rolle des Frontends: Kontrolle über die Darstellung

Durch diesen zusätzlichen display\_order-Anker hat das Frontend nun die Wahl:

* Es kann die Spielerliste so anzeigen, wie sie vom Server kommt, was zu einer Neusortierung in jedem Leg führen würde.
* Oder es kann – wie es aktuell als Standard implementiert ist – die empfangene Spielerliste **vor jeder Anzeige** anhand des display\_order-Feldes sortieren. Das sorgt für eine stabile Anzeige der Spielerliste über das ganze Match hinweg.

### Was diese Trennung bewirkt:

Das Backend stellt die notwendigen Daten bereit, um die komplexe Server-Logik abzubilden, während das Frontend die alleinige Entscheidungshoheit über die visuelle Präsentation behält und eine für den Zuschauer konsistente und ruhige Anzeige über das gesamte Match hinweg erzeugen kann oder die vom Autodarts-Server vorgegebene Spielerreihenfolge übernimmt.

## Wichtige Unterscheidung: Leg-Gewinner vs. Match-Gewinner

Die Analyse der Server-Events hat zwei unterschiedliche Mechanismen zur Bestimmung eines Gewinners offenbart, deren korrekte Interpretation entscheidend ist:

* gameWinner: **Leg-Gewinner.** Dieser Index bezieht sich **immer** auf die Position in der *aktuell rotierten* players-Liste.
* winner: **Match-Gewinner.** Dieser Index bezieht sich **immer** auf den *ursprünglichen, festen* index eines Spielers.

Diese Unterscheidung ist der Hauptgrund für die komplexe Logik zur Stabilisierung der Spielerreihenfolge im Backend.

## Verhalten des Scoreboards (bewusstes Design):

Um eine für den Zuschauer absolut ruhige und konsistente Anzeige zu gewährleisten, wird als Standard eine zu Beginn des Matches (ggf. nach dem Ausbullen) festgelegte Anzeigereihenfolge für die gesamte Dauer des Spiels beibehalten.

Dies wird durch eine intelligente "Latching"-Logik im Backend erreicht:

1. **Einmaliges "Festschreiben":** Zu Beginn einer Spielphase (z.B. dem ersten Leg des X01-Matches nach einem Bull-off) merkt sich das Backend die vom Autodarts-Server gesendete Spielerreihenfolge und speichert diese als feste Anzeigereihenfolge (display\_order).
2. **Übermittlung der Anweisung:** Bei jedem game-update-Event sendet das Backend die players-Liste in der vom Autodarts-Server rotierten Reihenfolge. Zusätzlich wird jedem Spieler-Objekt seine "festgeschriebene" display\_order-Nummer mitgegeben.
3. **Sortierung im Frontend:** Das Frontend-JavaScript ist dafür verantwortlich, die empfangene players-Liste anhand der display\_order-Nummer zu sortieren, **bevor** die Tabelle oder die Spielerkarten auf dem Bildschirm gezeichnet werden.

Durch diesen Mechanismus bleibt die visuelle Darstellung für den Zuschauer über alle Legs hinweg stabil, während das Backend intern weiterhin mit den rotierten Live-Daten des Servers arbeiten kann.

Diese feste Reihenfolge kann jedoch überschrieben werden. Entweder durch die URL-Parameter

* /?rs = „Reihenfolge sortiert“ (also der oben beschriebene Standard)
* /?rn = „Reihenfolge nativ“ (also die vom Autodarts-Server zu jedem Leg-Beginn neu sortierte Reihenfolge)

Ebenfalls ist es durch setzten des Parameters FORCE\_STABLE\_SORTING in config\_frontend.py vor dem Start des Frontends möglich.

* True = „Reihenfolge sortiert“ (also der oben beschriebene Standard)
* False = Reihenfolge nativ“ (also die vom Autodarts-Server zu jedem Leg-Beginn neu sortierte Reihenfolge)

## Weiteres

Das Backend verarbeitet die komplexen Daten vom Autodarts-Server und übersetzt sie in die einfache GameEvent-Struktur. Die wichtigsten Erkenntnisse aus der Analyse der Rohdaten sind:

* **variant vs. gameMode**: Das Backend ermittelt den korrekten Spielmodus jetzt zuverlässig über die Regel: game\_mode = settings.get('gameMode', data.get('variant')). Dies löst den Sonderfall Cricket/Tactics und wird in match.game\_mode gespeichert.
* **Bull-off**: Die Bull-off-Phase vor einem X01-Match wird jetzt als eigenständiger, temporärer game\_mode: "Bull-off" erkannt und verarbeitet, inklusive der Erkennung eines Unentschiedens.
* **Datenquelle**: Die Treffer bei Cricket (state.segments) und die Live-Statistiken (stats) werden bei jedem Wurf vom Server bereitgestellt, was eine zustandslose Verarbeitung im Backend ermöglicht. Der Gesamt-Average (overall\_average) für registrierte Spieler wird aus dem user- oder host-Objekt beim Match-Start und bei Leg-Ende ausgelesen.

# Analyse der Autodarts-Server Events für X01-Matches

Diese Dokumentation legt die Funktionsweise der vom Autodarts-Server gesendeten WebSocket-Events dar, mit besonderem Fokus auf die Reihenfolge und Indizierung der Spielerdaten über mehrere Legs hinweg. Die Analyse basiert auf den Log-Dateien eines Matches mit vier Spielern (christian, corinna, mona, marc).

## Grundlegende Objekte und ihre Eigenschaften

Die Events von Autodarts verwenden mehrere Listen und Indizes, deren Zusammenspiel entscheidend für die korrekte Datenverarbeitung ist.

**Der "Feste Index" (index)**

* Bei Match-Start (match-started Event) wird jedem Spieler ein **fester, unveränderlicher Index** für die gesamte Dauer des Matches zugewiesen. Dieser Index ist die primäre Referenz für alle statistischen Daten.
* *Beispiel aus den Logs:*  
  "players": [  
   { "name": "christian", "index": 0 },  
   { "name": "corinna", "index": 1 },  
   { "name": "mona", "index": 2 },  
   { "name": "marc", "index": 3 }  
  ]

### Das players-Array (Die Wurf-Reihenfolge)

* Die Reihenfolge der Spielerobjekte in diesem Array repräsentiert die **aktuelle Wurf-Reihenfolge** des laufenden Legs.
* In **Leg 1** entspricht die Reihenfolge im players-Array der Reihenfolge des Festen Index.
* Zu Beginn von **Leg 2** (und jedem folgenden Leg) **rotiert** die Reihenfolge in diesem Array. Die Spieler rücken eine Position auf, und der erste Spieler rückt an das Ende.

### Der player-Index (Aktueller Werfer)

* Das player-Feld in einem match-data-Event ist ein Integer (0, 1, 2, ...).
* Dieser Index bezieht sich **immer** auf die Position in der **aktuell rotierten players-Liste**.

### Die Daten-Arrays (gameScores, stats, scores)

* Diese drei Arrays sind immer parallel zur aktuell rotierten players-Liste sortiert. Das bedeutet: stats[0] gehört zum Spieler an Position 0 der players-Liste in demselben Event.

### Die Gewinner-Indizes (gameWinner, winner)

Hier gibt es zwei unterschiedliche Regeln:

* **gameWinner** (Leg-Gewinner): Dieser Index bezieht sich immer auf die Position (0, 1, 2...) des Spielers in der aktuell rotierten players-Liste des Events.
* **winner** (Match-Gewinner): Dieser Index bezieht sich immer auf den ursprünglichen, festen Index des Spielers, der zu Match-Beginn vergeben wurde.

## Das Zusammenspiel im Match-Verlauf

Anhand der Log-Daten lässt sich dieses Verhalten eindeutig nachweisen.

**Match-Start & Leg 1**

Beim match-started-Event wird die feste Reihenfolge festgelegt:

Reihenfolge:  
"leg": 1,  
"players": [  
 { "name": "christian", "index": 0 }  
 { "name": "corinna", "index": 1 },  
 { "name": "mona", "index": 2 },  
 { "name": "marc", "index": 3 },

]  
Während des gesamten ersten Legs ist die Wurf-Reihenfolge identisch mit dieser festen Reihenfolge. Wenn Christian (player: 0) wirft, wird gameScores[0] aktualisiert.

**Leg 2**

* **Die Rotation findet statt:** Das erste Event für Leg 2 zeigt die neue, rotierte Wurf-Reihenfolge:  
  "leg": 2,  
  "players": [  
   { "name": "corinna", "index": 0 },  
   { "name": "mona", "index": 1 },  
   { "name": "marc", "index": 2 },  
   { "name": "christian", "index": 3 }  
  ]
* **Der erste Wurf:** Corinna ist an der Reihe. Das Event zeigt player: 0, was korrekt auf ihre neue Position in der rotierten Liste verweist.
* **Die Daten-Aktualisierung:** Corinna wirft 41 Punkte und ihr Score sinkt auf 80. Wenn wir das gameScores-Array im entsprechenden Event betrachten, sehen wir, dass der Wert ebenfalls an **Position 0** aktualisiert wurde, was ebenfalls ihrem aktuellen Index in der rotierten Liste entspricht. Dies beweist, dass die Daten-Arrays immer die feste, ursprüngliche Indexierung beibehalten.

## Ableitung für die korrekte Logik

Aus dieser korrigierten Analyse ergibt sich die finale Architektur deines Backends:

1. **Initialisierung:** Bei der Verarbeitung des allerersten Events eines Matches wird ein zentrales Objekt g.player\_name\_to\_index\_map erstellt. Es speichert für jeden Spieler seinen stabilen Index (stable\_index) (also den Index, den er bei Beginn des Matches hatte) und eine anfangs leere Anzeigereihenfolge (display\_order: None).
2. **"Latching" der Anzeigereihenfolge:** Die universelle Funktion create\_universal\_game\_event prüft bei jedem Aufruf, ob die display\_order für einen Spieler None ist. Wenn ja, wird die aktuelle Position des Spielers aus der vom Server rotierten players-Liste als neue, feste Anzeigereihenfolge "festgeschrieben" (gelatcht).  
   Da dies beim Start des Matches geschieht, entspricht display\_order also immer dem Index der Startreihenfolge des Matches.
3. **Bull-off als "Reset":** Die process\_match\_bull\_off-Funktion setzt nach Ende des Ausbullens die display\_order für alle Spieler wieder auf None. Dies zwingt die Logik dazu, beim Start des eigentlichen Hauptspiels (z.B. X01) eine neue Anzeigereihenfolge basierend auf der vom Server (hoffentlich nach dem Bull-off-Sieger sortierten) players-Liste festzuschreiben.
4. **Datenanreicherung:** Module wie process\_match\_cricket.py nutzen den Schleifen-Index (enumerate), um die parallelen, rotierten Listen (players und stats) korrekt zu durchlaufen und die Daten dem richtigen Spieler zuzuordnen.
5. **Finale Sortierung im Frontend:** Das Backend sendet die players-Liste unsortiert (in der vom Server rotierten Reihenfolge), fügt aber jedem Spieler seine display\_order-Nummer hinzu. Das Frontend-JavaScript (utils\_frontend.js) ist dafür verantwortlich, die Liste anhand dieser Nummer zu sortieren, um eine für den Zuschauer über das gesamte Match stabile Ansicht zu erzeugen.

# Besonderheiten: Match-Initialisierung und Zufalls-Modi

Die Initialisierung eines Matches, insbesondere bei Spielmodi mit zufälligen Startwerten (z.B. Random Checkout), erfordert eine spezielle Logik, da der Autodarts-Server nicht alle benötigten Informationen sofort bereitstellt.

## Das "Kickstart"-Verfahren beim Match-Start

Die Analyse hat gezeigt, dass der Autodarts-Server direkt nach der Erstellung eines Matches (insbesondere bei Zufalls-Modi) oft nur einen unvollständigen Datensatz liefert. Um zuverlässig einen vollständigen und korrekten Startzustand zu erhalten, wurde ein "Kickstart"-Verfahren implementiert:

1. Das Backend empfängt das start-Signal vom autodarts.boards-Kanal.
2. Anstatt zu versuchen, die unvollständigen Daten zu verarbeiten, agiert die Funktion orchestrate\_match\_start\_and\_finish nur als **Trigger**. Sie sendet einen leeren PATCH-Request an den /throws-Endpunkt des Matches.
3. Dieser "leere" Aufruf veranlasst den Autodarts-Server den Spielzustand intern zu finalisieren und einen **vollständigen** game-update über den WebSocket zu senden.
4. Dieses vom Server gesendete Event enthält nun garantiert alle Informationen (inklusive zufälliger Startwerte) und wird vom Backend als das **erste, maßgebliche Event** zur Initialisierung der Anzeige verarbeitet.

## Vereinheitlichte Event-Verarbeitung

Alle eingehenden game-update-Nachrichten vom autodarts.matches-Kanal werden nun von einem einzigen, zentralen Dispatcher (\_handle\_matches\_channel) an die jeweilige process\_match\_\*-Funktion geleitet. Diese ist somit für die Initialisierung **und** für alle nachfolgenden Aktualisierungen zuständig.

# Ablauf der Events im *Backend* am Beispiel ATC

## Mairmaid-Syntax (match-start Event)

*(Kann z.B. mit* [*https://app.diagrams.net/*](https://app.diagrams.net/) *dargestellt werden)*

graph TD

subgraph "Autodarts Server"

A["Board 'start' Event"]

B["Server sendet 'game-update' via WebSocket"]

end

subgraph "Backend Verarbeitung (Teil 1 - Trigger)"

C["\_handle\_boards\_channel (websocket\_handlers.py)"]

D["orchestrate\_match\_start\_and\_finish (match\_handler.py)"]

E["Leerer PATCH-Request (match\_handler.py)"]

end

subgraph "Backend Verarbeitung (Teil 2 - Initialisierung)"

F["\_handle\_matches\_channel (websocket\_handlers.py)"]

G["Dispatcher: GAME\_PROCESSORS (websocket\_handlers.py)"]

H["process\_match\_\* (z.B. \_atc.py)"]

I["\_handle\_matches\_channel (websocket\_handlers.py)"]

J["broadcast() (websocket\_handlers.py)"]

end

K[Backend sendet 'game-update' an Frontend]

A --> C

C --> D

D --> E

E -- Triggert --> B

B --> F

F --> G

G --> H

H -- "return event" --> I

I --> J

J --> K

## Erläuterung des Match-Start Ablaufs:

1. **Der Auslöser:**  
   Der Prozess beginnt, wenn der Autodarts-Server ein start-Event auf dem statisch abonnierten Kanal **autodarts.boards** sendet.
2. **Der Empfang:**  
   Die on\_message\_autodarts-Funktion empfängt diese Nachricht und leitet sie an den zuständigen \_handle\_boards\_channel-Handler weiter.
3. **Die Initialisierung:**  
   \_handle\_boards\_channel ruft die zentrale Funktion orchestrate\_match\_start\_and\_finish (in match\_handler.py) auf.
4. **Der "Kickstart":**  
   orchestrate\_match\_start\_and\_finish agiert nun als reiner Trigger. Anstatt selbst Daten zu verarbeiten, sendet die Funktion einen leeren PATCH-Request an den /throws-API-Endpunkt des Matches.
5. **Die Server-Reaktion:**  
   Dieser PATCH-Request veranlasst den Autodarts-Server, den Spielzustand intern zu finalisieren und eine vollständige game-update-Nachricht (die alle Startinformationen enthält) über den Kanal **autodarts.matches** zu senden.
6. **Die Verarbeitung:**  
   Diese neue Nachricht wird wieder von on\_message\_autodarts empfangen und diesmal an den \_handle\_matches\_channel-Handler weitergeleitet.
7. **Der Dispatcher:** \_handle\_matches\_channel nutzt den GAME\_PROCESSORS-Dispatcher, um die Daten an die korrekte process\_match\_\*-Funktion (z.B. process\_match\_atc) zu übergeben.
8. **Event-Erstellung & Rückgabe:** Die process\_match\_\*-Funktion erstellt das finale, vollständige game-update-Event und **gibt es zurück** an den \_handle\_matches\_channel.
9. **Der Broadcast:** \_handle\_matches\_channel empfängt das fertige Event und sendet es mittels broadcast() an den Frontend-Hub.

## Grafik (match-start Event)

Ein Bild, das Text, Screenshot, Rechteck, parallel enthält.

KI-generierte Inhalte können fehlerhaft sein.

## Mairmaid-Syntax (folgender Wurf)

graph TD

subgraph "Autodarts Server"

A["Match 'state' Event (Wurf)"]

end

subgraph "Backend Verarbeitung"

B["\_handle\_matches\_channel (websocket\_handlers.py)"]

C["Dispatcher: GAME\_PROCESSORS (websocket\_handlers.py)"]

D["process\_match\_\* (z.B. \_atc.py)"]

E["\_handle\_matches\_channel (websocket\_handlers.py)"]

F["broadcast() (websocket\_handlers.py)"]

end

G[Backend sendet 'game-update' an Frontend]

A --> B

B --> C

C --> D

D -- "return event" --> E

E --> F

F --> G

## Erläuterung des Game-Update Ablaufs (folgende Würfe):

1. **Der Auslöser:** Der Autodarts-Server sendet nach einem Wurf ein state-Update auf dem dynamisch abonnierten Kanal **autodarts.matches**.
2. **Der Empfang:** Die on\_message\_autodarts-Funktion empfängt die Nachricht und leitet sie an den \_handle\_matches\_channel-Handler weiter.
3. **Der Dispatcher:** Der GAME\_PROCESSORS-Dispatcher in \_handle\_matches\_channel identifiziert den Spielmodus und leitet die Daten an die zuständige process\_match\_\*-Funktion (z.B. process\_match\_atc) weiter.
4. **Event-Erstellung & Rückgabe:** Die process\_match\_\*-Funktion verarbeitet die Live-Daten (Scores, Würfe, Stats), erstellt das game-update-Event und **gibt es zurück** an den \_handle\_matches\_channel.
5. **Der Broadcast:** \_handle\_matches\_channel empfängt das fertige Event und sendet es mittels broadcast() an den Frontend-Hub.

## Grafik (folgender Wurf)

Ein Bild, das Text, Screenshot, Zahl, Schrift enthält.

KI-generierte Inhalte können fehlerhaft sein.

# Ablauf der Events im *Frontend* am Beispiel ATC

## Mairmaid-Syntax (game-update)

graph TD

subgraph "Backend"

A[Backend sendet 'game-update' Event]

end

subgraph "Frontend Bridge (app\_frontend.py)"

B["on\_backend\_event"]

C["socketio\_server.emit('status\_update', data)"]

end

subgraph "Browser Client (JavaScript)"

D["socket.on('status\_update')<br/>(scoreboard.js)"]

E["\_cacheLatestServerState() (speichert in appState)<br/>(scoreboard.js)"]

F["\_processGameNotifications() (zeigt Overlays etc.)<br/>(scoreboard\_helpers.js)"]

G["\_routeToGameViewUpdater() (Router-Funktion)<br/>(scoreboard\_helpers.js)"]

H["VIEW\_HANDLERS Dispatcher"]

I["update...View() (z.B. view\_x01.js)"]

J["renderGameTable() / renderPlayerCards()<br/>(utils\_frontend.js)"]

end

L[UI wird aktualisiert]

A --> B

B --> C

C --> D

D --> E

E --> F

F --> G

G --> H

H --> I

I --> J

J --> L

## Erläuterung des Match-Start Ablaufs im Frontend:

1. **Empfang:** Der Frontend-Hub (app\_frontend.py) empfängt ein Event vom Backend und leitet es als status\_update an alle verbundenen Browser weiter.
2. **Zustand speichern:** Im Browser empfängt scoreboard.js dieses Event. Die Funktion \_cacheLatestServerState()speichert die neuen Daten unverändert im globalen appState.
3. **Spezial-Events behandeln:** Die Funktion \_processGameNotifications()(in scoreboard\_helpers.js) prüft auf Zustände wie leg\_won oder match\_won, um Overlays, Feuerwerk etc. anzuzeigen.
4. **Der Router:** Die Funktion \_routeToGameViewUpdater()-Funktion wird aufgerufen. Sie agiert als zentraler Router.
5. **Sortierung:** Die \_routeToGameViewUpdater()(in scoreboard\_helpers.js) ruft nun die zentralen Rendering-Funktionen (renderGameTable oder renderPlayerCards) in utils\_frontend.js auf. **Innerhalb dieser Funktionen** wird die Spielerliste aus dem appState genommen und anhand des mitgesendeten display\_order-Feldes sortiert. Dies stellt sicher, dass die Spieler für die Anzeige immer in der gleichen, stabilen Reihenfolge sind.
6. **Ansicht-spezifisches Update:** \_routeToGameViewUpdater() identifiziert den Spielmodus und ruft die zuständige update...View()-Funktion (z.B. updateX01View()) auf.
7. **Finale Darstellung:** Die update...View()-Funktion orchestriert die finale Darstellung: Sie erstellt ein ViewModel, rendert den oberen Fokus-Bereich (renderFocusArea) und ruft die Rendering-Funktionen (renderGameTable / renderPlayerCards) auf, welche die Tabelle bzw. die Karten mit der **bereits korrekt sortierten Spielerliste** aufbauen.
8. **UI-Update:** Die Benutzeroberfläche ist nun mit der stabilen Spielerreihenfolge aktualisiert.

## Grafik

Ein Bild, das Text, Reihe, Screenshot, Quittung enthält.

KI-generierte Inhalte können fehlerhaft sein.

## Ablaufdiagramm der Kommunikation

Dieses Diagramm zeigt, wie der Ablauf der Kommunikation zwischen Backend und Autodarts\_Server läuft.

sequenceDiagram

participant Backend as Ihr Backend

participant Keycloak as Autodarts Login

participant AutodartsAPI as Autodarts REST API

participant LocalBoard as Local Board Manager API

participant WebSocket as Autodarts WebSocket

Backend->>Keycloak: Authentifizierungsanfrage

Keycloak-->>Backend: Access Token

Backend->>WebSocket: Verbindungsaufbau (mit Token)

WebSocket-->>Backend: Verbindung erfolgreich

Backend->>AutodartsAPI: GET /boards/{boardId} (ruft IP ab)

AutodartsAPI-->>Backend: Board-IP

WebSocket->>Backend: 'start'-Event (vom Board-Kanal)

Backend->>AutodartsAPI: PATCH /matches/{matchId}/throws (Kickstart)

AutodartsAPI-->>Backend: HTTP 200 OK

WebSocket->>Backend: 'game-update' Event

loop Live-Spiel

WebSocket->>Backend: 'game-update' Event (nach jedem Wurf)

Backend->>Backend: Daten verarbeiten...

end

Note over Backend, LocalBoard: Optionale Fernsteuerung

Backend->>LocalBoard: PUT /api/detection/start (Beispiel)

LocalBoard-->>Backend: HTTP 200 OK

## Erläuterung des Ablaufs:

1. **Authentifizierung & WebSocket-Verbindung:** Das Backend meldet sich am Autodarts-Server an und baut eine dauerhafte WebSocket-Verbindung auf, über die es die statischen Kanäle wie autodarts.boards abonniert.
2. **IP-Adresse abrufen:** Das Backend sendet eine Anfrage an die **Autodarts REST API**, um die lokale IP-Adresse des Board Managers zu ermitteln und für spätere, optionale Fernsteuerungsbefehle zu speichern.
3. **Match-Start-Signal (WebSocket):** Wenn ein neues Match startet, empfängt das Backend ein einfaches start-Signal über den autodarts.boards-Kanal.
4. **"Kickstart" (REST API):** Anstatt nun selbst die Daten per GET-Request zu holen, sendet das Backend einen leeren PATCH-Request an den /throws-Endpunkt des Matches. Dieser Aufruf dient als Trigger, um den Server zu veranlassen, den Spielzustand intern zu finalisieren.
5. **Vollständige Daten (WebSocket):** Als Reaktion auf den PATCH-Request sendet der Server nun von sich aus eine vollständige game-update-Nachricht über den autodarts.matches-Kanal. Diese Nachricht enthält garantiert alle Informationen, auch für Zufalls-Modi.
6. **Live-Betrieb (WebSocket):** Alle weiteren Würfe werden als game-update-Nachrichten über den WebSocket gesendet.
7. **Optionale Fernsteuerung (Local API):** Befehle, die über die UI des Scoreboards ausgelöst werden (z.B. "Board Start"), werden vom Backend direkt an die API des **lokalen Board Managers** gesendet.

## GrafikEin Bild, das Text, Screenshot, Software, Design enthält. KI-generierte Inhalte können fehlerhaft sein.

# Kommunikation: Der WebSocket Abonnement-Prozess

Die Echtzeit-Kommunikation zwischen dem Backend und dem Autodarts-Server basiert auf einem **Publish/Subscribe-Modell** (auch Pub/Sub genannt). Anstatt den Server ständig nach neuen Daten zu fragen (Polling), baut das Backend eine permanente WebSocket-Verbindung auf und "abonniert" bestimmte Nachrichtenkanäle.

Der entscheidende Aspekt dieser Architektur ist, dass die Abonnements **statisch** und **dynamisch** erfolgen, um die Datenmenge effizient zu halten.

## Phase 1: Der Verbindungsaufbau

Der Verbindungsaufbau wird durch die Bibliothek websocket-client gesteuert und bleibt unverändert. Er schafft die technische Grundlage für die Kommunikation, indem er eine permanente, authentifizierte Verbindung zum Server herstellt und Callback-Funktionen (on\_open, on\_message etc.) registriert.

## Phase 2: Statische und Dynamische Kanal-Abonnements

Sobald die Verbindung steht, wird die on\_open\_autodarts-Funktion ausgeführt. Hier beginnt der zweistufige Abonnement-Prozess.

**Statische Abonnements (Die Grundversorgung)**

Zuerst werden die allgemeinen Kanäle abonniert. Diese dienen als "Alarmglocken" oder "Nachrichten-Ticker", die dem Backend mitteilen, *dass* etwas Wichtiges passiert ist.

Python

# In on\_open\_autodarts() in websocket\_handlers.py

# Abonnement für Board-Events (z.B. "Ein Match hat begonnen!")

params = {"channel": "autodarts.boards", "type": "subscribe", ...}

websocket\_connection.send(json.dumps(params))

# Abonnement für User-Events (z.B. "Du hast eine Lobby betreten!")

params = {"channel": "autodarts.users", "type": "subscribe", ...}

websocket\_connection.send(json.dumps(params))

**Dynamische Abonnements (Bedarfsgesteuert)**

Dies ist der intelligente Teil der Logik. Anstatt von Anfang an alle potenziellen Match- oder Lobby-Details zu abonnieren, wartet das Backend auf eine "Alarmglocke" von den statischen Kanälen. Sobald eine solche Nachricht eintrifft, wird gezielt ein neues, dynamisches Abonnement nur für das relevante, spezifische Thema (z.B. eine konkrete Match-ID) abgeschlossen.

**Beispielhafter Ablauf bei Match-Start:**

1. Eine Nachricht auf dem autodarts.boards-Kanal meldet: "Match mit ID 'xyz-123' wurde gestartet."
2. Diese Nachricht löst die Funktion orchestrate\_match\_start\_and\_finish in match\_handler.py aus.
3. orchestrate\_match\_start\_and\_finish agiert nun als reiner Trigger: Es sendet einen leeren PATCH-Request an den /throws-Endpunkt des Matches, um die vollständigen Spieldaten anzufordern.
4. **Gleichzeitig** werden in orchestrate\_match\_start\_and\_finish die dynamischen Abonnements für die Kanäle autodarts.matches und autodarts.boards.events für die spezifische Match-ID eingerichtet:

Python

# In orchestrate\_match\_start\_and\_finish() in match\_handler.py

paramsSubscribeMatchesEvents = {

"channel": "autodarts.matches",

"type": "subscribe",

"topic": g.active\_match\_id + ".state"

}

websocket\_connection.send(json.dumps(paramsSubscribeMatchesEvents))

1. Der Server verarbeitet den PATCH-Request und sendet daraufhin die vollständigen Startdaten über den soeben abonnierten autodarts.matches-Kanal.

Derselbe dynamische Prozess findet statt, wenn eine Lobby betreten wird, um den autodarts.lobbies-Kanal zu abonnieren.

.

**Analogie: Zeitungsabonnement**

* **Statisches Abo:** Sie abonnieren die tägliche Hauptausgabe einer Zeitung. Sie erhalten die Schlagzeilen des Tages.
* **Dynamisches Abo:** Eine Schlagzeile lautet: "Spannender Bericht über Thema X". Sie rufen beim Verlag an und bestellen gezielt das Sonderheft nur zu diesem Thema. Sobald das Thema nicht mehr relevant ist, kündigen Sie das Sonderheft-Abo wieder (unsubscribe).

Diese Vorgehensweise stellt sicher, dass das Backend nur die Daten empfängt, die es aktuell auch wirklich benötigt, was die Kommunikation sehr effizient macht.

## Sicherstellung der Verbindungsstabilität (Ping/PongHeart-Beat)

Langlebige WebSocket-Verbindungen können durch Netzwerkkomponenten (Router, Firewalls) unterbrochen werden, wenn über einen längeren Zeitraum keine Daten fließen (Idle-Timeout). Um diese unerwünschten Verbindungsabbrüche zwischen dem Backend und dem Autodarts-Server zu verhindern, wurde ein proaktiver "Herzschlag"-Mechanismus implementiert.

Im websocket\_handlers.py werden beim Start der Verbindung zwei Parameter an die run\_forever()-Methode übergeben:

* ping\_interval=25: Weist den Client an, alle 25 Sekunden ein kleines "Ping"-Paket an den Autodarts-Server zu senden.
* ping\_timeout=10: Definiert, dass die Verbindung als getrennt betrachtet wird, wenn der Server nicht innerhalb von 10 Sekunden auf einen Ping mit einem "Pong" antwortet.

Dieser konstante, leichte Datenverkehr signalisiert allen Geräten auf der Übertragungsstrecke, dass die Verbindung aktiv ist und erhalten bleiben muss, was die Stabilität des Systems erheblich erhöht.

# API-Endpunkte des Autodarts-Servers

## REST API Endpunkte (HTTP)

Diese werden für einmalige Abfragen von Daten genutzt, z.B. um den Zustand beim Laden einer Seite zu holen. Die Basis-URL ist https://api.autodarts.io/.

* **GET /gs/v0/matches/{matchId}**
  + **Zweck:** Dient dem Abrufen von detaillierten Daten zu einem spezifischen Match. Dies ist exakt der Aufruf, den auch das Backend in match\_handler.py verwendet, um die initialen Spieldaten zu laden.
* **PATCH /gs/v0/matches/{matchId}/throws**
  + **Zweck (NEU):** Dieser Endpunkt ist der Schlüssel für den "Kickstart"-Mechanismus. Das Senden eines leeren JSON-Objekts ({}) mit dieser Methode an den Server veranlasst diesen, den internen Zustand eines neu erstellten Matches zu finalisieren und einen vollständigen game-update per WebSocket zu senden. Dies umgeht das Problem unvollständiger Startdaten bei Zufalls-Modi.
* **GET /ls/v0/lobbies**
  + **Zweck:** Dient der Verwaltung von Lobbies, also dem Erstellen, Beitreten und Abfragen von Spiel-Lobbys.
* **GET /us/v0/users/{userId}**
  + **Zweck:** Dient dem Abrufen von Benutzerprofilen und Statistiken.
* **GET /bs/v0/boards**
  + **Zweck:** Dient der Verwaltung und Abfrage von Informationen zu den Boards.

## WebSocket Endpunkt

Dieser wird für die dauerhafte Echtzeit-Kommunikation genutzt, um Live-Daten vom Server zu empfangen.

* **/ms/v0/subscribe**
  + **Zweck:** Dies ist die Adresse für den WebSocket-Server. Die WebApp verbindet sich mit dieser URL und sendet dann subscribe-Nachrichten, um die bekannten Kanäle (autodarts.matches, autodarts.lobbies, etc.) zu abonnieren.

## API-Endpunkte des lokalen Boardmanagers

### Board-Steuerung

Diese Endpunkte steuern direkt den Zustand des Boards.

* **PUT /api/start**
  + **Beschreibung:** Startet die Wurf-Erkennung. Entspricht der start\_board()-Funktion.
* **PUT /api/stop**
  + **Beschreibung:** Stoppt die Wurf-Erkennung. Entspricht der stop\_board()-Funktion.
* **POST /api/reset**
  + **Beschreibung:** Setzt den Zustand des Boards zurück (z.B. nach einem verklemmten Dart). Entspricht der reset\_board()-Funktion.
* **POST /api/restart**
  + **Beschreibung:** Startet den gesamten Board-Manager-Dienst auf dem Host-System neu. Entspricht der restart\_board()-Funktion.

### Status- und Informationsabfragen

* **GET /api/state/stats**
  + **Beschreibung:** Ruft allgemeine Live-Statistiken ab, insbesondere die **durchschnittliche** FPS über alle Kameras.
  + **Antwort-Beispiel:** {"resolution":{"width":320,"height":180},"fps":0}
* **GET /api/cams/stats**
  + **Beschreibung:** Ruft detaillierte Live-Statistiken für jede einzelne Kamera ab, inklusive der individuellen FPS-Werte.
  + **Antwort-Beispiel:** {"resolution": {"width": 1280, "height": 720}, "fps": [30, 30, 30]}
* **GET /api/cams/state**
  + **Beschreibung:** Gibt den allgemeinen Zustand der Kameras zurück.
  + **Antwort-Beispiel**: {"isOpened": true, "isRunning": true}
* **GET /api/version**
  + **Beschreibung:** Gibt die Version des Board-Manager-Dienstes zurück.
  + **Antwort-Beispiel**: 0.26.15
* **GET /api/devices**
  + **Beschreibung:** Listet alle erkannten Kamera-Geräte am Host-System auf.
  + **Antwort-Beispiel:**

[

{

"card": "Autodarts Vision Cam: Autodarts",

"bus": "usb-0000:00:14.0-2",

"vendorId": 3141,

"productId": 6421,

"formats": [

{

"path": "/dev/video0",

"name": "MJPG",

"resolutions": [

{

"width": 1280,

"height": 720,

"framerates": [30]

},

{

"width": 800,

"height": 600,

"framerates": [30]

},

{

"width": 640,

"height": 480,

"framerates": [30]

}

]

}

]

},

{

"card": "Autodarts Vision Cam: Autodarts",

"bus": "usb-0000:00:14.0-3",

"vendorId": 3141,

"productId": 6421,

"formats": [

{

"path": "/dev/video2",

"name": "MJPG",

"resolutions": [

{

"width": 1280,

"height": 720,

"framerates": [30]

},

{

"width": 800,

"height": 600,

"framerates": [30]

},

{

"width": 640,

"height": 480,

"framerates": [30]

}

]

}

]

},

{

"card": "Autodarts Vision Cam: Autodarts",

"bus": "usb-0000:00:14.0-4",

"vendorId": 3141,

"productId": 6421,

"formats": [

{

"path": "/dev/video4",

"name": "MJPG",

"resolutions": [

{

"width": 1280,

"height": 720,

"framerates": [30]

},

{

"width": 800,

"height": 600,

"framerates": [30]

},

{

"width": 640,

"height": 480,

"framerates": [30]

}

]

}

]

}

]

# Deployment und Betrieb

## NGINX-Konfiguration:

Alle Backend-Pfade sind unter einem einzigen /api-Präfix gebündelt, was die NGINX-Konfiguration und die allgemeine Architektur vereinfacht.

Zudem ist unter https://<DomainName/IP>/api eine Übersichtsseite verfügbar.

Hier eine beispielhafte (funktionierende) NGINX-Konfiguration für Frontend und Backend.

map $http\_upgrade $connection\_upgrade {

default upgrade;

'' close;

}

# Upstream für den HTTPS-Dienst auf Port 6001

upstream darts\_hub\_backend {

server 10.0.1.112:6001;

}

# Upstream für den HTTP-Dienst auf Port 6002

upstream darts\_hub\_frontend {

server 10.0.1.112:6002;

}

# NEU: Upstream für den neuen, einfachen HTTP-CMD-Frontend-Service (Port 6003)

upstream darts\_cmd\_frontend {

server 10.0.1.112:6003;

}

server {

listen 443 ssl;

listen [::]:443 ssl;

server\_name ~^(darts-hub|d)\.himue\.(com|lan|local)$;

charset utf-8;

# zusätzliche Zeiel wäre auch möglich

#server\_name ~^d\.himue\.(com|lan|local)$;

# Zertifikate laden

include snippets/certhimue.conf;

# Location-Block für den HTTPS-Backend-Service

# Zugriff über: https://darts-hub.himue.com/api/...

location /api {

proxy\_pass https://darts\_hub\_backend;

# NGINX anweisen, das selbst-signierte Zertifikat zu ignorieren

proxy\_ssl\_verify off;

# Standard-Header für die Weiterleitung

proxy\_set\_header Host $host;

proxy\_set\_header X-Real-IP $remote\_addr;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

proxy\_set\_header X-Forwarded-Proto $scheme;

proxy\_http\_version 1.1;

proxy\_buffering off;

# Wichtig für WebSockets (Socket.IO)

proxy\_set\_header Upgrade $http\_upgrade;

proxy\_set\_header Connection $connection\_upgrade;

}

# Location-Block für den HTTP-Frontend-Service (Port 6001)

# Zugriff über: https://darts-hub.himue.com/

location / {

proxy\_pass https://darts\_hub\_frontend/;

# NGINX anweisen, das selbst-signierte Zertifikat vom Frontend zu akzeptieren

proxy\_ssl\_verify off;

proxy\_set\_header Host $host;

proxy\_set\_header X-Real-IP $remote\_addr;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

proxy\_set\_header X-Forwarded-Proto $scheme;

proxy\_http\_version 1.1;

proxy\_set\_header Upgrade $http\_upgrade;

proxy\_set\_header Connection $connection\_upgrade;

proxy\_buffering off;

}

}

## Bash-Skripte

### Frontend Developing: start-dev.sh

#!/bin/bash

python3 app\_frontend.py 2>&1

### Frontend Produktion: start-prod.sh

#!/bin/bash

#Es gibt beim Beenden des gunocorn Servers ein Traceback. Das ist aber im Endeffekt nur eine harmlose Infomeldung, die hier unterdrückt w>

gunicorn --config config\_gunicorn\_frontend.py app\_frontend:app 2>&1 | grep -v "SystemExit"

### Backend Developing: start-dev.sh

#!/bin/bash

python3 app\_backend.py

### Backend Produktion: start-prod.sh

#!/bin/bash

#Es gibt beim Beenden des gunocorn Servers ein Traceback. Das ist aber im Endeffekt nur eine harmlose Infomeldung, die hier unterdrückt w>

gunicorn -c gunicorn.conf.py "app\_backend:app" 2>&1 | grep -v "SystemExit"

## Datenbank

### SQL-Befehle zur Erstellung der Tabellen

-- Tabellen für X01

CREATE TABLE `players\_x01` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`name` varchar(255) NOT NULL,

`is\_registered` tinyint(1) NOT NULL DEFAULT 0,

`average` decimal(5,2) DEFAULT NULL,

`created\_at` timestamp NOT NULL DEFAULT current\_timestamp(),

`updated\_at` timestamp NOT NULL DEFAULT current\_timestamp() ON UPDATE current\_timestamp(),

PRIMARY KEY (`id`),

UNIQUE KEY `name` (`name`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_general\_ci;

CREATE TABLE `games\_history\_x01` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`player\_id` int(11) NOT NULL,

`match\_id` varchar(255) NOT NULL,

`leg\_number` int(11) NOT NULL,

`leg\_average` decimal(5,2) NOT NULL,

`leg\_points` int(11) NOT NULL,

`leg\_darts` int(11) NOT NULL,

`finished\_at` timestamp NOT NULL DEFAULT current\_timestamp(),

PRIMARY KEY (`id`),

KEY `player\_id` (`player\_id`),

CONSTRAINT `games\_history\_x01\_ibfk\_1` FOREIGN KEY (`player\_id`) REFERENCES `players\_x01` (`id`) ON DELETE CASCADE

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_general\_ci;

-- Tabellen für Cricket

CREATE TABLE `players\_cricket` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`name` varchar(255) NOT NULL,

`is\_registered` tinyint(1) NOT NULL DEFAULT 0,

`mpr` decimal(5,2) DEFAULT NULL,

`created\_at` timestamp NOT NULL DEFAULT current\_timestamp(),

`updated\_at` timestamp NOT NULL DEFAULT current\_timestamp() ON UPDATE current\_timestamp(),

PRIMARY KEY (`id`),

UNIQUE KEY `name` (`name`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_general\_ci;

CREATE TABLE `games\_history\_cricket` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`player\_id` int(11) NOT NULL,

`match\_id` varchar(255) NOT NULL,

`leg\_number` int(11) NOT NULL,

`leg\_marks` int(11) NOT NULL,

`leg\_darts` int(11) NOT NULL,

`finished\_at` timestamp NOT NULL DEFAULT current\_timestamp(),

PRIMARY KEY (`id`),

KEY `player\_id` (`player\_id`),

CONSTRAINT `games\_history\_cricket\_ibfk\_1` FOREIGN KEY (`player\_id`) REFERENCES `players\_cricket` (`id`) ON DELETE CASCADE

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_general\_ci;

-- Tabellen für Tactics

CREATE TABLE `players\_tactics` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`name` varchar(255) NOT NULL,

`is\_registered` tinyint(1) NOT NULL DEFAULT 0,

`mpr` decimal(5,2) DEFAULT NULL,

`created\_at` timestamp NOT NULL DEFAULT current\_timestamp(),

`updated\_at` timestamp NOT NULL DEFAULT current\_timestamp() ON UPDATE current\_timestamp(),

PRIMARY KEY (`id`),

UNIQUE KEY `name` (`name`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_general\_ci;

CREATE TABLE `games\_history\_tactics` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`player\_id` int(11) NOT NULL,

`match\_id` varchar(255) NOT NULL,

`leg\_number` int(11) NOT NULL,

`leg\_marks` int(11) NOT NULL,

`leg\_darts` int(11) NOT NULL,

`finished\_at` timestamp NOT NULL DEFAULT current\_timestamp(),

PRIMARY KEY (`id`),

KEY `player\_id` (`player\_id`),

CONSTRAINT `games\_history\_tactics\_ibfk\_1` FOREIGN KEY (`player\_id`) REFERENCES `players\_tactics` (`id`) ON DELETE CASCADE

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_general\_ci;

-- Tabellen für ATC

CREATE TABLE `players\_atc` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`name` varchar(255) NOT NULL,

`is\_registered` tinyint(1) NOT NULL DEFAULT 0,

`hit\_rate` decimal(5,4) DEFAULT NULL, -- z.B. 0.7780 für 77.8%

`created\_at` timestamp NOT NULL DEFAULT current\_timestamp(),

`updated\_at` timestamp NOT NULL DEFAULT current\_timestamp() ON UPDATE current\_timestamp(),

PRIMARY KEY (`id`),

UNIQUE KEY `name` (`name`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_general\_ci;

-- Speichert die Details jedes einzelnen gespielten ATC-Legs

CREATE TABLE `games\_history\_atc` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`player\_id` int(11) NOT NULL,

`match\_id` varchar(255) NOT NULL,

`leg\_number` int(11) NOT NULL,

`leg\_hit\_rate` decimal(5,4) NOT NULL,

`leg\_darts` int(11) NOT NULL,

`finished\_at` timestamp NOT NULL DEFAULT current\_timestamp(),

PRIMARY KEY (`id`),

KEY `player\_id` (`player\_id`),

CONSTRAINT `games\_history\_atc\_ibfk\_1` FOREIGN KEY (`player\_id`) REFERENCES `players\_atc` (`id`) ON DELETE CASCADE

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_general\_ci;

-- Tabellen für Count Up

CREATE TABLE `players\_countup` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`name` varchar(255) NOT NULL,

`is\_registered` tinyint(1) NOT NULL DEFAULT 0,

`ppr` decimal(5,2) DEFAULT NULL, -- Points Per Round

`created\_at` timestamp NOT NULL DEFAULT current\_timestamp(),

`updated\_at` timestamp NOT NULL DEFAULT current\_timestamp() ON UPDATE current\_timestamp(),

PRIMARY KEY (`id`),

UNIQUE KEY `name` (`name`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_general\_ci;

-- Speichert die Details jedes einzelnen gespielten Count Up-Legs

CREATE TABLE `games\_history\_countup` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`player\_id` int(11) NOT NULL,

`match\_id` varchar(255) NOT NULL,

`leg\_number` int(11) NOT NULL,

`leg\_points` int(11) NOT NULL,

`leg\_darts` int(11) NOT NULL,

`finished\_at` timestamp NOT NULL DEFAULT current\_timestamp(),

PRIMARY KEY (`id`),

KEY `player\_id` (`player\_id`),

CONSTRAINT `games\_history\_countup\_ibfk\_1` FOREIGN KEY (`player\_id`) REFERENCES `players\_countup` (`id`) ON DELETE CASCADE

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_general\_ci;

-- Tabellen für Segment Training

CREATE TABLE `players\_segment\_training` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`name` varchar(255) NOT NULL,

`is\_registered` tinyint(1) NOT NULL DEFAULT 0,

`hit\_rate` decimal(5,4) DEFAULT NULL,

`created\_at` timestamp NOT NULL DEFAULT current\_timestamp(),

`updated\_at` timestamp NOT NULL DEFAULT current\_timestamp() ON UPDATE current\_timestamp(),

PRIMARY KEY (`id`),

UNIQUE KEY `name` (`name`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_general\_ci;

-- Speichert die Details jedes einzelnen gespielten Segment Training-Legs

CREATE TABLE `games\_history\_segment\_training` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`player\_id` int(11) NOT NULL,

`match\_id` varchar(255) NOT NULL,

`leg\_number` int(11) NOT NULL,

`leg\_hit\_rate` decimal(5,4) NOT NULL,

`leg\_darts` int(11) NOT NULL,

`finished\_at` timestamp NOT NULL DEFAULT current\_timestamp(),

PRIMARY KEY (`id`),

KEY `player\_id` (`player\_id`),

CONSTRAINT `games\_history\_segment\_training\_ibfk\_1` FOREIGN KEY (`player\_id`) REFERENCES `players\_segment\_training` (`id`) ON DELETE CASCADE

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_general\_ci;

## Python-Umgebung (venv)

Es wird dringend empfohlen, das Backend in einer dedizierten virtuellen Python-Umgebung (venv) auszuführen, um Abhängigkeitskonflikte zu vermeiden.

#### Umgebung erstellen und aktivieren:

Bash

# Erstelle eine neue venv im Projektverzeichnis

python3 -m venv venv

# Aktiviere die venv (Linux/macOS)

source venv/bin/activate

# Aktiviere die venv (Windows)

.\venv\Scripts\activate

**12.4.1 Abhängigkeiten installieren (requirements.txt)**

Erstelle eine Datei requirements.txt im Hauptverzeichnis des Backends mit folgendem Inhalt. Die Pakete können dann gesammelt mit pip install -r requirements.txt installiert werden.

# requirements.txt

# Core

Flask-SocketIO

gunicorn

gevent-websocket

python-dotenv

requests

rich

# Datenbank

mariadb

# WebSocket-Client & Authentifizierung

websocket-client

python-keycloak

cryptography

# System-Tools

psutil

# Weiteres

## Zentrales Logging

Die gesamte Logging-System nutzt das rich-Modul inklusive einer zentralen setup\_logger-Funktion, die Farben erzwingt und ein benutzerdefiniertes Theme verwendet.

## Resilientes Frontend:

Der Frontend-Server fragt nach einem Neustart oder Reconnect jetzt aktiv den letzten Spielzustand vom Backend ab. Das verhindert das Zurückspringen auf die Startseite und macht die Anwendung deutlich robuster.

# Schnittstelle für Frontend-Befehle und die /cmd-Seite

Um dem Frontend die Möglichkeit zu geben, gezielt Aktionen im Backend auszulösen (z.B. das Starten des Boards oder das Abrufen von Live-Daten), wurde eine robuste, asynchrone Anfrage-Antwort-Schnittstelle auf Basis von Socket.IO implementiert. Die /cmd-Seite dient als Referenzimplementierung und als Diagnosewerkzeug für diese Schnittstelle.

## Die command/command\_response-Schnittstelle

Die Schnittstelle basiert auf zwei dedizierten Socket.IO-Events, die über die Frontend-Bridge (app\_frontend.py) weitergeleitet werden:

* **command**: Vom Browser an das Backend gesendet, um eine Aktion auszulösen.
* **command\_response**: Vom Backend an den Browser gesendet, um das Ergebnis einer Aktion zurückzumelden.

### Ablauf der Kommunikation

Der gesamte Prozess ist so gestaltet, dass er asynchron ist und die Frontend-Bridge als reiner, aber fehlertoleranter Vermittler agiert.

1. **Frontend (cmd.js):** Eine sendCommand-Hilfsfunktion erstellt eine Anfrage. Sie generiert eine eindeutige callback\_id, speichert die zugehörige Callback-Funktion in einem lokalen Dictionary und sendet das command-Event an die Frontend-Bridge.
2. **Frontend-Bridge (app\_frontend.py):**
   * Der @socketio\_server.on('command')-Handler empfängt die Anfrage vom Browser.
   * Er prüft, ob die interne Verbindung zum Backend aktiv ist. Wenn ja, leitet er die Anfrage 1:1 an das Backend weiter. Wenn nein, sendet er sofort eine Fehlermeldung an den anfragenden Browser zurück.
3. **Backend (webserver\_handler.py):**
   * Der @socketio.on('command')-Handler (handle\_command) empfängt die Anfrage von der Bridge.
   * Ein command\_dispatcher-Dictionary leitet die action an die zuständige Python-Funktion weiter (z.B. get\_stats()). Diese Funktionen befinden sich in der Datei local\_board\_client.py.
   * Das Ergebnis der Funktion wird in ein Antwort-Objekt verpackt.
   * socketio.emit('command\_response', ...) sendet die Antwort (inklusive der originalen callback\_id) zurück an die Frontend-Bridge.
4. **Frontend-Bridge (app\_frontend.py):**
   * Ein separater @sio\_client.on('command\_response')-Handler lauscht auf Antworten vom Backend.
   * Er leitet die Antwort-Nachricht 1:1 an alle verbundenen Browser weiter.
5. **Frontend (cmd.js):**
   * Der zentrale socket.on('command\_response', ...)-Listener empfängt die Antwort.
   * Er nutzt die callback\_id aus der Antwort, um die ursprünglich gespeicherte Callback-Funktion zu finden und mit den Ergebnisdaten auszuführen.

### Diagramm: Ablauf der command-Schnittstelle

graph TD

subgraph "Browser (z.B. cmd.js)"

A["Aktion (z.B. Button-Klick)"] --> B["sendCommand(aktion, params, callback)"];

B --> C["socket.emit('command', ...)<br/>speichert Callback"];

D["socket.on('command\_response', ...)"] --> E["Führt gespeicherten Callback<br/>mit Antwort-Daten aus"];

end

subgraph "Frontend-Bridge (app\_frontend.py)"

F["@socketio\_server.on('command')"] --> G["g.sio\_client.emit('command', ...)<br/>leitet Anfrage weiter"];

H["@sio\_client.on('command\_response')"] --> I["g.socketio\_server.emit('command\_response', ...)<br/>leitet Antwort weiter"];

end

subgraph "Backend (webserver\_handler.py)"

J["@socketio.on('command')"] --> K["Dispatcher findet & ruft Funktion auf"];

K --> L["Funktion holt Daten<br/>(z.B. von local\_board\_client)"];

L --> M["socketio.emit('command\_response', ...)<br/>sendet Ergebnis"];

end

C -- Anfrage --> F;

G -- Anfrage --> J;

M -- Antwort --> H;

I -- Antwort --> D;

### Grafik

Ein Bild, das Text, Screenshot, Rechteck, Diagramm enthält.

KI-generierte Inhalte können fehlerhaft sein.

# Beispielanwendung: Das Standalone CMD-Dashboard

Um die strikte Trennung von Backend und Frontend und die Flexibilität der Architektur zu demonstrieren, wurde eine komplett eigenständige Frontend-Anwendung erstellt.

Diese "Mini-Anwendung" ist ein perfektes Beispiel dafür, wie beliebige Clients (eine Web-App, eine mobile App, ein Skript) über die definierte command/command\_response-Schnittstelle mit dem zentralen himues-darts-hub-Backend interagieren können, ohne dass am Backend selbst Änderungen vorgenommen werden müssen. Die Anwendung läuft auf einem eigenen Port (6003) und ist unter einer eigenen Subdomain (test-d.local.lan) erreichbar.

## Anwendungsfall: Das CMD-Dashboard

Das CMD-Dashboard ist eine eigenständige Single-Page-Anwendung, die als Steuerungs- und Diagnose-Dashboard dient. Sie zeigt Live-Daten vom loaklen Board-Manager an und erlaubt das Auslösen von Aktionen wie "Start", "Stop" oder "Reset" des Boards.

## Funktionsweise der automatischen Aktualisierung

Die Seite wird durch aktives, clientseitiges Polling aktualisiert.

1. **Initialisierung:** Nach dem Laden der Seite und dem Aufbau der Socket.IO-Verbindung zum eigenen Server (app\_cmd.py) wird in cmd.js ein setInterval gestartet, der die updateDashboard-Funktion jede Sekunde ausführt.
2. **Parallele Anfragen:** Die updateDashboard-Funktion startet über Promise.all mehrere Anfragen parallel, um alle benötigten Daten (Konfiguration, Statistiken, etc.) gleichzeitig vom Backend anzufordern. Diese command-Events werden vom CMD-Frontend (app\_cmd.py) empfangen und an das Haupt-Backend weitergeleitet.
3. **Warten auf Antworten:** Das Haupt-Backend sendet die Antworten als command\_response-Events zurück an das CMD-Frontend, welches sie wiederum an den Browser weiterleitet. Promise.all im Browser wartet, bis für alle Anfragen eine Antwort eingetroffen ist.
4. **Rendering:** Sobald alle Daten verfügbar sind, wird die renderTable-Funktion aufgerufen, welche die komplette HTML-Tabelle mit den neuen Daten neu aufbaut.
5. **Wiederholung:** Nach einer Sekunde löst der setInterval den Prozess erneut aus.

## Diagramm: Der Dashboard-Aktualisierungszyklus

Dieses Diagramm zeigt den Kommunikationsfluss für die neue, eigenständige Anwendung.

graph TD

subgraph "Phase 1: Initialer Seitenaufbau"

A["Browser ruft httpstest-d.local.lan auf"] --> B["cmd\_page()<br/>(app\_cmd.py)"];

B --> C["cmd.js wird ausgeführt<br/>($(document).ready)"];

C --> D["WebSocket-Verbindung wird zum<br/>CMD-Frontend aufgebaut"];

D --> E["'connect'-Event wird empfangen"];

E --> F["startDashboardInterval()<br/>(cmd.js)"];

end

subgraph "Phase 2: Aktualisierungs-Loop (Browser)"

G["setInterval löst aus:<br/>updateDashboard()"] --> H["Promise.all startet 4 parallele Anfragen"];

H --> I["sendCommand('get\_...')..."];

I --> J["4x socket.emit('command')<br/>an CMD-Frontend"];

end

subgraph "Kommunikations-Bridge"

subgraph "CMD-Frontend (app\_cmd.py)"

K["Empfängt 4 'command'-Events vom Browser"] --> L["Leitet 4 'command'-Events<br/> an Haupt-Backend weiter"];

M["Empfängt 4 'command\_response'-Events<br/> vom Haupt-Backend"] --> N["Leitet 4 Antworten<br/>an Browser weiter"];

end

subgraph "Haupt-Backend (app\_backend.py)"

O["Empfängt 4 'command'-Events"] --> P["Führt Aktionen aus<br/>(z.B. get\_stats())"];

P --> Q["Sendet 4 'command\_response'-Events<br/>an CMD-Frontend"];

end

end

subgraph "Phase 2: Anzeige (Browser)"

R["Empfängt 4 'command\_response'-Events"] --> S["Promise.all ist komplett"];

S --> T["renderTable()<br/>(cmd.js)"];

end

F --> G;

J --> K;

L --> O;

Q --> M;

N --> R;

## Grafik

Ein Bild, das Schublade, Kommode enthält.

KI-generierte Inhalte können fehlerhaft sein.

## Nginx

Wenn NGINX genutzt werden soll, muss die NGINX-Konfiguration folgendermaßen erweitert werden:

# NEU: Server-Block für das CMD-Frontend

server {

listen 443 ssl;

listen [::]:443 ssl;

server\_name ~^test-d\.himue\.(com|lan|local)$; # Dein neuer Domain-Name

# Dieselben Zertifikate wie der Hauptserver verwenden

include snippets/certhimue.conf;

location / {

# proxy\_pass auf den Upstream für Port 6003

# Wichtig: KEIN Schrägstrich am Ende

proxy\_pass https://darts\_cmd\_frontend;

# Standard-Header für Proxy & WebSockets

proxy\_set\_header Host $host;

proxy\_set\_header X-Real-IP $remote\_addr;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

proxy\_set\_header X-Forwarded-Proto $scheme;

proxy\_http\_version 1.1;

proxy\_set\_header Upgrade $http\_upgrade;

proxy\_set\_header Connection $connection\_upgrade;

proxy\_buffering off;

}

}