Dokumentation der Software-Architektur

Prettner, Stephan
s.prettner@oth-aw.de

Rube, Alexander
a.rube@oth-aw.de

Schafberger, Tobias
t.schafberger@oth-aw.de

Schneider, Oliver
o.schneider1@oth-aw.de

Schuster, Felix
f.schuster@oth-aw.de

Stangl, Dennis
d.stangl@oth-aw.de

Zuber, Maximilian
m.zuber@oth-aw.de

***Abstract*— In diesem Technical Report wird die Architektur vorgestellt, die für die Projektarbeit im Fach Big Data und Cloud-basiertes Computing (BDCC) eingesetzt wird. Ziel dieser Projektarbeit ist es, die Themengebiete Big Data, Cloud Computing und Künstliche Intelligenz zu verknüpfen.**

I. Allgemeine Struktur

Die technische Struktur des Projekts orientiert sich stark an gängigen Strukturen aus dem Bereich der Microservices. Dies bietet den Vorteil, dass einzelne Teile beliebig skaliert werden können, um dadurch die Anforderungen bestens umsetzen zu können.
Alle Nutzerinteraktionen mit dem Projekt laufen über das .NET Core Projekt. Dieses bietet Backend-Funktionen für ein zukünftig geplantes Frontend, worüber ein Nutzer alle Anfragen bequem über eine graphische Oberfläche stellen kann. Zudem fragt das .NET Core Projektdaten von der OpenDota API ab und speichert diese in ein MongoDB Replica Set. Ebenso werden Nutzeranfragen an den KI-Bereich weitergegeben.
Zum Trainieren der künstlichen Intelligenz, sowie zum Auswerten von Spielen, welche ein Nutzer angeben kann, wird ein Jupyter-Notebook-Server verwendet. Dieser kann über einen Jupyter-Kernel-Gateway per REST-Schnittstelle angesprochen werden. Auf diese Weise wird er mit Daten befüllt, welche in einem eigenen MongoDB Replica Set gespeichert werden. Außerdem kann er über REST-Befehle gesteuert werden, um zum Beispiel anhand von vorhandenen Daten einen Algorithmus zu trainieren oder aber ein Spiel eines Nutzers auszuwerten.
Die einzelnen Komponenten werden in einer YAML-Datei konfiguriert, um die Umgebungsvariablen sowie die unterschiedlichen Abhängigkeiten zwischen diesen Containern zu definieren. Mithilfe von Docker-Compose können diese dann auf dem Host-System gestartet werden. Dies bietet den Vorteil, dass die Laufzeitumgebung bei jedem Entwickler, sowie beim Deployen der Anwendung, bereits richtig konfiguriert ist und somit keine Probleme bereitet.
Eine grafische Darstellung kann in Fig. 1 eingesehen werden.

II. .NET Core

Für das zentrale Backend wurde das .NET Core Framework von Microsoft gewählt, da dieses in C# entwickelt wurde, womit die meisten Projektmitglieder bereits Erfahrungen sammeln konnten. Zudem bietet Microsoft Docker-Images für .NET Core an.
Um die Grundidee von Microservices auch im Backend korrekt aufgreifen und umsetzen zu können, wird das Domain-driven Design bei der Entwicklung angewandt. Dieses sieht die Unterteilung des Projekts in die drei Komponenten „API“, „Data“ und „Domain“ vor. Die Komponente API beinhaltet alle Funktionalitäten zum Starten und Steuern des Projekts. In diesem befindet sich das .NET Core Framework mit seinen Controllern, welche die REST-Schnittstellen definieren, sowie Konfigurationen wie die „appsettings.json“ Datei für Umgebungsvariablen oder die „Startup.cs“ Datei, in welcher der Dependency Injector konfiguriert wird, um Interfaces und dazugehörige Implementierungen an Konstruktoren verschiedener Klassen weitergeben zu können. Hierfür besitzt die API-Komponente Abhängigkeiten zu sowohl der Data- als auch der Domain-Komponente.
Die Data-Komponente ist von der Domain-Komponente abhängig und implementiert die in der Domain-Komponente definierten Repository-Interfaces als vollständige Klassen. Dies bietet den Vorteil, dass die konkrete Implementierung der Datenhaltung unabhängig vom eigentlichen Code ist, was einen Austausch der verwendeten Datenbank, zum Beispiel von MongoDB auf MSSQL, ohne Änderungen am eigentlichen Code der Domain ermöglicht.
Die Domain-Komponente besitzt keine Abhängigkeiten auf eines der anderen Projekte und implementiert jegliche Logik, welche zur Umsetzung einer Aufgabe nötig ist. Sie bettet alle verwendeten Modelle ein und definiert die Funktionen der Repositories in Interfaces.
Diese Aufteilung der Aufgabenbereiche ermöglicht zudem ein besseres Testen der einzelnen Komponenten.

A. Tests für die Matchverwaltung

Für die Umsetzung eines Testprojekts zur Matchverwaltung, die in einer .NET Core-Umgebung realisiert ist, kommt das unter der MIT-Lizenz stehende Open-Source Test-Framework NUnit zum Einsatz. Mit diesem lassen sich Testmethoden erstellen und ausführen, die beliebige Methoden des Backends für die Matchverwaltung unter vorher festgelegten Voraussetzungen ausführen und durch Assert-

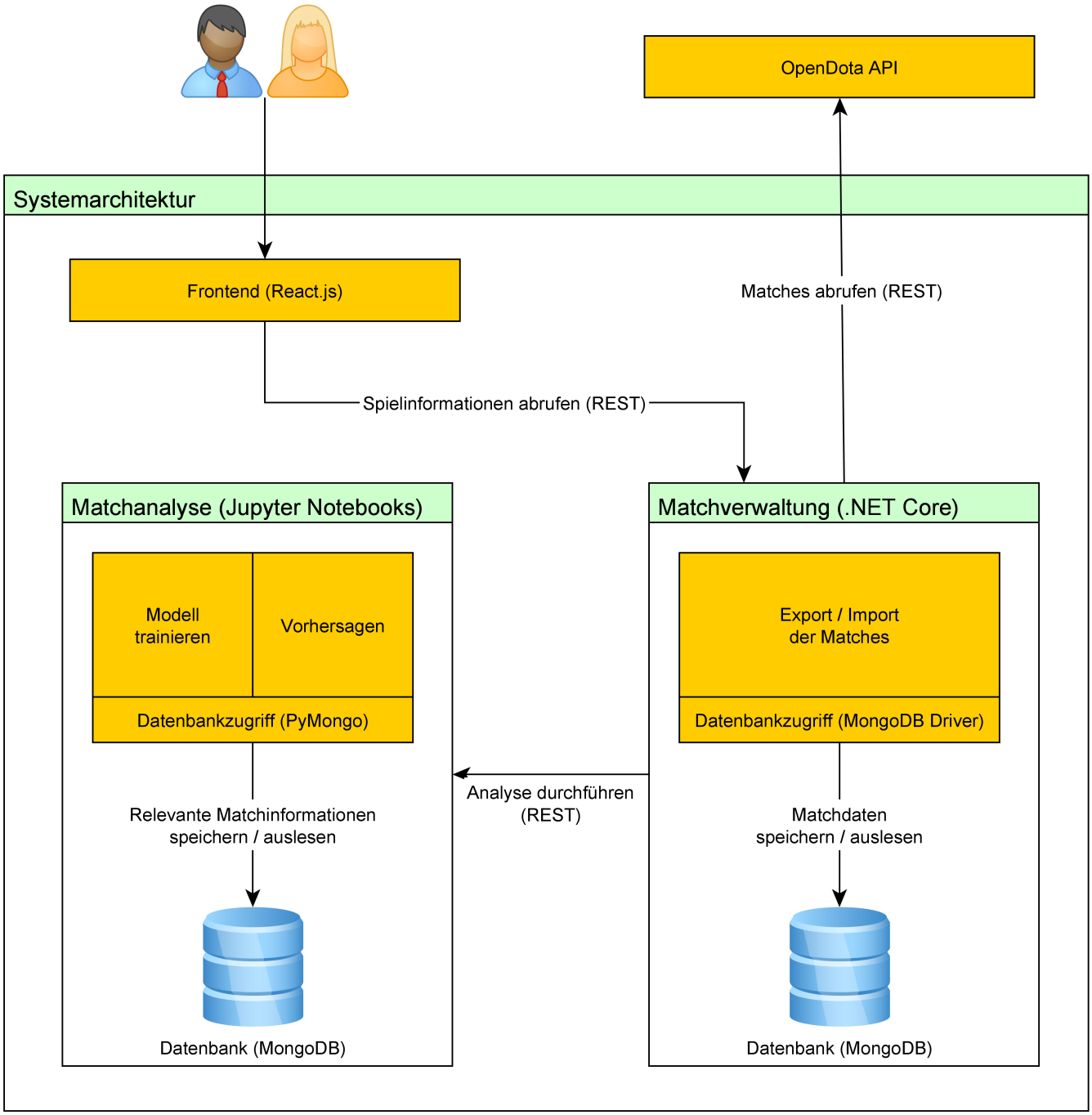

Fig. 1. Grafische Darstellung der verwendeten Architektur

Anweisungen prüfen, ob das System wie erwartet reagiert. Das Testprojekt, das auf die Funktionalität von NUnit zurückgreift, bündelt dabei sowohl Unit-Tests zur Sicherstellung der Funktionstüchtigkeit der in der Matchverwaltung implementierten Methoden als auch diverse Integrationstests. Letztere prüfen die Lauffähigkeit und Korrektheit externer Systeme und den dazugehörigen Schnittstellen. Dazu gehören die Aufrufe der verwendeten MongoDB-Datenbank zur Verwaltung darin gespeicherter Matchdaten, die Nutzung der REST-Schnittstelle, die OpenDota zur Sammlung von Spieldaten zu „Dota 2“ zur Verfügung stellt, als auch die Steuerung der Matchanalyse, die separat in einem Jupyter-Notebook-Server realisiert ist.
Die Ausführung aller in dem Projekt vorhandenen Tests ermöglicht dabei einen sofortigen Überblick über die korrekte Implementierung der Komponenten und zeigt potentiellen Handlungsbedarf an.

B. *OpenDota*

Für die Beschaffung der benötigten Informationen aus den Spieldaten wird die Programmierschnittstelle (API) des Open

Source Projekts „OpenDota“ verwendet. In der hier verwendeten kostenlosen Variante können bis zu 60 Anfragen pro Minute an die Webseite gestellt werden. Ein „TimeLimiter“ nach dem „Token bucket"-Prinzip sorgt für die Einhaltung dieser Begrenzung.
Die Informationsbeschaffung erfolgt entweder mit zufälligen Spielen oder den Spielen eines gewünschten Spielers. Dazu werden wiederholt Anfragen an den OpenDota-Server gesendet, um mehrere eindeutige Match-IDs zu bekommen und alle Attribute der jeweiligen Spiele anfordern zu können. Mittels der Steam32-ID eines Spieleraccounts können so ebenfalls gezielt Spiele des jeweiligen Spielers betrachtet werden.
Durch eine verringerte Datenstruktur, in der nur die nötigsten Attribute enthalten sind, wird die Speicherbelastung von circa 215 kB pro Spiel auf ungefähr 15 kB reduziert. Jedes angeforderte Spiel wird aufgrund der Anfragenbegrenzung in der MongoDB-Datenbank gespeichert.

III. Jupyter-Notebooks

A. Interface zwischen .Net-Core- und Python-Backend

Das Interface für die Interaktion zwischen dem .NET Core- und Python-Backend wird mit einem Jupyter-Server realisiert, welcher als sogenannter Jupyter-Kernel-Gateway aufgesetzt wird. Dadurch ist es möglich, verschiedene REST-Schnittstellen über ein Jupyter-Notebook zu definieren, worüber HTTP-Anfragen gestellt werden können. In diesem Projekt werden ausschließlich POST-Anfragen an das Python-Backend gesendet. Die Rückgabe des Aufrufs sind HTTP Status Codes.

B. Schnittstellen

Das Gateway ist über die lokale URL über den Port „8898“ erreichbar und bietet folgende Schnittstellen:

- /writematch: schreibt übergebene Daten eines einzelnen Matches in die MongoDB/KI
  - Parameter: Daten des zu übertragenden Matches im JSON-Format
  - Return: None
- /deletematch: löscht ein Match aus der MongoDB/KI
  - Parameter: Match-ID des zu löschenden Matches
  - Return: None
- /trainmodel: startet das Training des KI-Modells
  - Parameter: Name des zu trainierenden Modells
  - Return: None
- /predict: startet die Vorhersage des KI-Modells
  - Parameter: Name des zu trainierenden Modells
  - Return: Beschreibung des Ergebnisses als String

C. KI-Modell