Design Document

Christian Stricker David Klopp Markus Vieth

2. Dezember 2015

Inhaltsverzeichnis

Teil I

Architectural Design

Kapitel 1

Einleitung

Im Folgenden werden in diesem Dokument verschiedene Perspektiven des zu entwickelnden Systems betrachtet. Dazu wird das System in Teilsysteme zerlegt und deren Verhalten aufgezeigt.

Das System, sowie alle Angaben zum System, beziehen sich dabei auf das „Requirements Document for TODO“ vom 20. November 2015.

Kapitel 2

Externe Sicht

Das System, als Web-Applikation, interagiert mit anderen Systemen in seiner Umgebung. TODO kommuniziert zur Übertragung von Daten mit mehreren Clients, welche Anfragen senden und Antworten empfangen, und mit weiteren Servern um Daten in den Datenbanken auszutauschen.

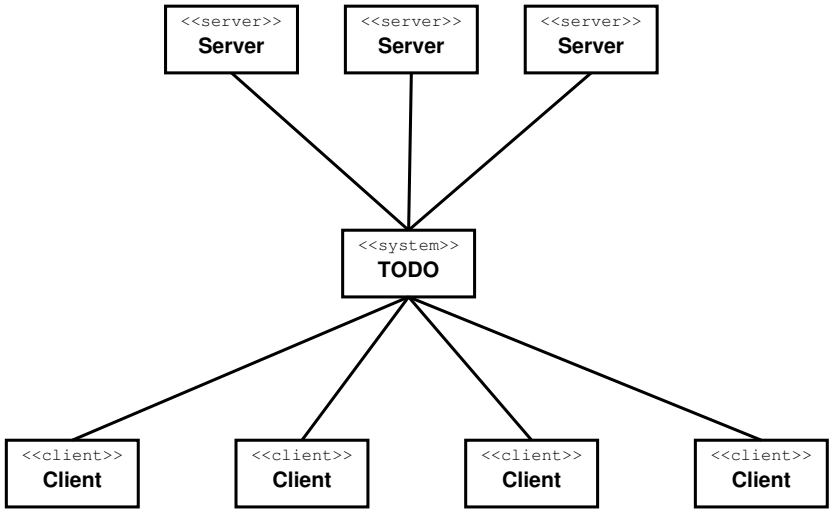

Abbildung 2.1: Context Diagram des Systems im Bezug zu seiner Umgebung

Die Kommunikation zwischen dem TODO und anderen Servern läuft dabei über das REST-Interface ab. Auch die Clients nutzen REST um Anfragen an das System zu stellen. Die Verbindung wird über HTTPS aufgebaut.

Kapitel 3

Interaktionssicht

Kapitel 4

Struktursicht

4.1 Genutzte architektur Pattern

4.1.1 Client-Server

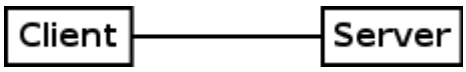

Abbildung 4.1: Client-Server-Modell

Client Der Client ist der User, der Datensätze, Algorithmen auf den Server hochladen, diese dann dort berechnen lassen und Pakete downloaden kann.

Server Der Server bietet dem User den Dienst an, Modelle anhand von schon vorhandenen oder vom User hoch geladenen Datensätzen und Algorithmen zu erstellen. Des Weiteren dient der Server als Datenbank von schon mit verschiedenen Datensätzen erstellten Modellen.

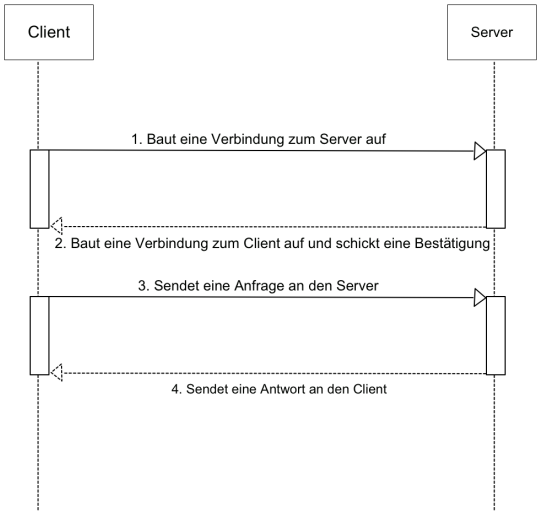

Abbildung 4.2: Client-Server-Sequenzdiagramm

Der Client versucht eine Verbindung zum Server aufzubauen, der Server versucht ebenfalls bei Anfrage eine Verbindung zum Client Aufzubauen und schickt dem Client eine Bestätigung, das die Verbindung steht. Danach kann der Client Datensätze/Algorithmen hochladen und dem Server eine Anfrage zum Modelle downloaden schicken.

Was spricht für das Client-Server-Modell?

Das Client-Server-Modell wird verwendet, wenn eine Datenbank oder ein Service von verschiedenen Orten her abgerufen werden soll. Dies ist für beides der Fall. Der User kann global auf den Server zugreifen und Daten hoch und runter laden.
Des Weiteren sollen mehrere gleiche Server online sein, damit viele User-anfragen auf mehrere Server verteilt werden können und somit schneller bearbeitet werden können. Der User sieht aber nur den einen Server. Wenn ein Server offline (ausfällt/gewartet)ist und es sind mehrere online, so bekommt der User davon nichts mit und wird auf einen anderen Server geleitet.
Nachteile sind:
-Die Leistung des Systems ist unberechenbar, wenn die verschiedenen Service im Server-Netzwerk verteilt ist. Dieser Fall existiert in unserem System nicht.
-Es gibt Management Probleme, wenn Server in relativ Unabhängigen Besitz ist. Da die Server unabhängig arbeiten und nur zusammenarbeiten, wenn schon vorhandene Modelle/Datensätze auf einem anderen Server angefragt werden, ist dieser Nachteil zu vernachlässigen.

4.1.2 Microkernel

Das System stellt seine Funktionalität über die WEKA-Libary zur Verfügung. Damit diese arbeiten kann, werden Algorithmen benötigt. Um eine dynamische Ergänzung der Algorithmen zu ermöglichen wird WEKA als Mikrokernel implementiert. So können die Algorithmen als interne Server, wenn benötigt, geladen werden und neue Algorithmen können hinzugefügt werden, ohne das der WEKA-Quellcode bearbeitet werden muss. Alle Anfragen an WEKA laufen dabei über eine Datenschnittstelle, welche verschiedene Funktionalitäten für den Client bereitstellt. So kann die Datenschnittstelle zurückgeben, welche Algorithmen von einem bestimmten Datensatz unterstützt werden oder ob ein zu erstellendes Modell bereits in der Datenbank vorhanden ist. WEKA übernimmt die Berechnung eines Modells und die Auswertung eines Datensatzes (bzw. eines Algorithmus). Der Web-Server übernimmt die Rolle eines Adapters, welcher die REST-Anfragen des Clients auswertet und weitere Instruktionen einleitet.

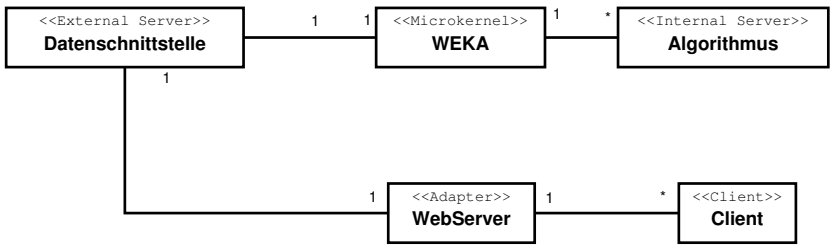

Abbildung 4.3: Microkernel-Pattern mit WEKA

4.1.3 Reflection

Um dynamisch später hinzugefügte Algorithmen auch in der Ein- und Ausgabe zu unterstützen, werden die Komponenten „PluginLoader“ und „Plugin“ verwendet. Der PluginLoader handhabt die Plugins und erstellt diese falls notwendig. Um dies zu tun, hat der PluginLoader Zugriff auf die Algorithmen. Somit kann der PluginLoader ein Plugin für die Eingabe von Parametern erzeugen, indem er sich von dem entsprechenden Algorithmus die Anzahl der benötigten Parameter ausgeben lässt. Analog kann der PluginLoader ein Plugin zur Ausgabe erzeugen, indem er sich zurückgeben lässt wie die Ausgabedaten aussehen und diese Anhand des gewünschten Ausgabeformats entweder als plain-text, html-text, JSON, PNG-Grafik oder SVG+XML-Grafik aufbereitet.

4.1.4 Layer

Um den Zugriff auf die Datenbank zu beschränken, wird zwischen die Datenschnittstelle und die Datenbank eine Benutzerverwaltung geschaltet. Diese

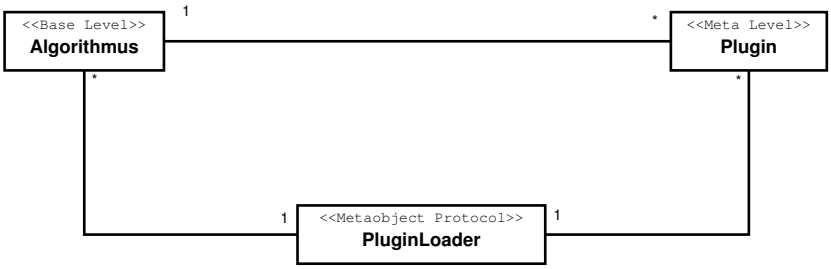

Abbildung 4.4: Reflectoin-Pattern mit PluginLoader

überprüft nun bei jeder Anforderung auf einen Zugriff auf die Datenbank, ob dieser vom aktuellen Nutzer erlaubt ist oder nicht.

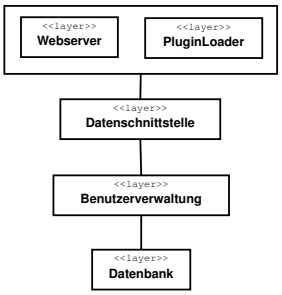

Abbildung 4.5: Layer-Pattern mit Benutzerverwaltung

Das Layer-Pattern hilft einen unerlaubten Zugriff auf die Datenbank zu verhindern und stellt sicher, dass jeder zugriff erst über die Benutzerverwaltung erlaubt wird. Ein weiterer Vorteil stellt die Flexibilität der einzelnen Schichten dar, so kann die Datenschnittstelle statt der lokalen Datenbank auch die Datenbank eines anderen Servers über REST abgerufen, da die Kommunikationswege innerhalb des Layer-Pattern identisch sind.

4.1.5 Model-View-Controller

Plugin

Um bestimmte Darstellungskomponenten, im weiteren als Plugins bezeichnet, in das UI (in unserem Fall die Website) einzubinden wird das MVC-Pattern verwendet. Solche Plugins können z.B. die Ausgabe von Ergebnissen als Text oder Grafik sein, aber auch Eingabemasken zum Erstellen von Modellen.

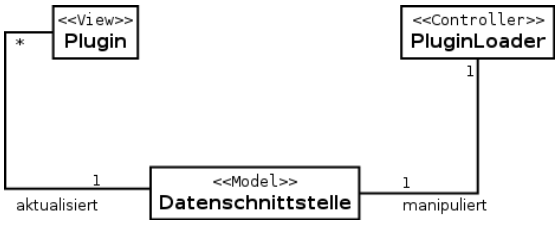

Abbildung 4.6: MVC-Pattern zum Website anzeigen

Sollte der Benutzer z.B. ein Modell generieren wollen oder sich ein Ergebnis anzeigen lassen, so formuliert der PluginLoader eine Anfrage an die Datenschnittstelle um mögliche Änderungen an diese zu senden. Nach diesem Aktualisierungsprozess wird ein Plugin durch den PluginLoader generiert, das über die Datenschnittstelle die notwendigen Informationen bezieht. Sollten hierbei Anfragen an andere Server nötig sein, so werden diese ebenfalls von der Datenschnittstelle mittels des REST-Protokolls durchgeführt. Das fertige Plugin wird nun in das Haupt-UI integriert.

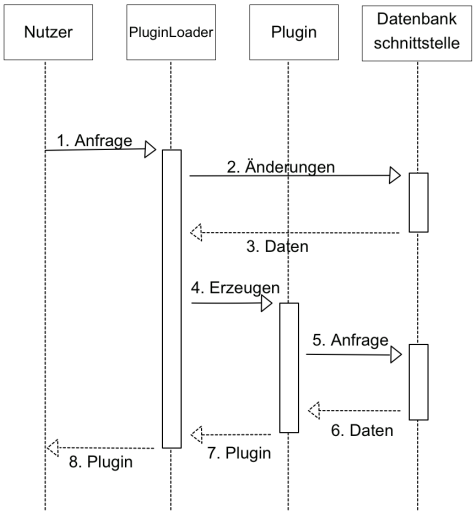

Abbildung 4.7: MVC-Pattern Sequenz zum Generieren der Website

Was spricht für Model-View-Controller?

Das MVC-Pattern ermöglicht es das System leicht zu erweitern. Durch die Einbindung verschiedener Plugins in das System, wird gewährleistet, das auf neue Gegebenheiten schnell reagiert werden kann. Sollte z.B. ein Algorithmus eine spezifische Ausgabe benötigen, könnte speziell für diesen ein neues Plugin erstellt werden. Des Weiteren erleichtert diese Aufteilung die Wartung des Systems enorm, da einzelne Plugins für sich getestet werden können, ohne in das Gesamtsystem eingebettet zu sein.

4.2 Nicht genutzte architektur Pattern

4.2.1 Presentation-Abstraction-Control

Was spricht gegen Presentation-Abstraction-Control?

Bei klar definierten Benutzer-System Interaktionen ist eine verschachtelte Struktur mittels Agenten, wie sie PAC vorsieht, zu umfangreich und nicht notwendig. Die Stärke von PAC liegt darin verschiedene, bestehende Teilsystem miteinander zu verknüpfen. In unserem System ließe sich dies beispielsweise auf die Datenbank und die Implementierung von WEKA anwenden. Da allerdings nur diese beiden Komponenten Daten generieren können, ist es einfacher diese über eine übergeordnete Komponente anzusprechen, als die hohe Komplexität des Pattern in Kauf zu nehmen. Der Fokus unsere Implementierung des Systems liegt auf der Erweiterbarkeit des UIs, die durch MVC leichter und effektiver gegeben ist.

4.2.2 Pipe and Filter

Was spricht gegen Pipe-and-Filter-Modell?

Dieses Modell ist zu mächtig, da die Ströme zwischen Server und Client nicht gekapselt oder verschlüsselt werden sollen. Server und Client müssten immer beim Aufbauen einer Verbindung eine Verschlüsselung vereinbaren und die Datenströme verschlüsseln und entschlüsseln. Dies vermindert oder verhindert sogar spätere Veränderungen vorzunehmen nd verbraucht unnötige Rechenzeit, da keine hoch sensiblen Daten ausgetauscht werden.

4.2.3 Blackboard

Was spricht gegen Blackboard?

Das hauptsächliche Einsatzgebiet des Blackboard-Patterns besteht darin, komplexe Sachverhalte auf kleiner Teilprobleme zu reduzieren und diese von Experten zu lösen. Anschließend wird das Gesamtergebnis aus den einzelnen Teilergebnissen zusammengesetzt. Unser System hingegen hat einen klar definierte sequenzielle Programmablauf:

- Benutzereingabe
- Auswertung der Eingabe
- Datenbankzugriff oder Berechnung.

Es macht daher keinen Sinn Teilprobleme bilden zu wollen. Bei der spezifischen Implementierung eines Algorithmus könnte dieses Pattern eventuell Anwendung finden.

4.2.4 Broker

Was spricht gegen Broker-Modell?

Das Broker-Modell vermindert die Performance, da alle Komponente des Systems nur indirekt angesprochen werden. Des Weiteren hängt die Kommunikation der einzelnen Systeme von vielen Komponenten ab und ist deshalb Fehleranfällig. Sinnvoll wäre der Broker nur, wenn viele verschiedene Clients auf viele verschiedenen Server auf viele verschiedene Service zugreifen wollen. Die Anzahl der Services unseres Servers ist sehr überschaubar und deshalb ist der Broker zu ineffizient für unseren Server.