

LSP Dungeon

Ziel dieser Ausarbeitung ist die Umsetzung einer Erweiterung für VS-Code. Die Erweiterung soll bei der Entwicklung der Dungeon DSL helfen. Hierfür werden Code-Highlighting, Fehlermeldungen und Completions umgesetzt, die für eine angenehmere Implementierung sorgen. Im Verlaufe der Dokumentation wird erläutert, welche Schritte durchlaufen wurden um die drei geforderten Komponenten zu implementieren.

Dungeon DSL

Grundsätzlich ist die Dungeon DSL eine Domain Specific Language (DSL), die es ermöglicht interaktive Lernszenarien in Form eines RPG-Games zu erstellen. Diese Lernszenarien können mit Hilfe der DSL "programmiert" werden indem verschiedene Aufgabentypen (Tasks) in Abhängigkeit zueinander gebracht werden. Anhand der Aufgaben und ihren Abhängigkeiten werden automatisch Level generiert, in welchen der Spieler durch verschiedene Interaktionen Aufgaben lösen muss. Dateien, die Dungeon DSL Code beinhalten haben die Endung **.dng**.

Grundsätzlich bauen sich die Dungen DSL-Dateien aus den immer gleichen Bestandteilen auf, die definiert werden. Diese Bestandteile sind für die Erstellung der Erweiterung von großer Relevanz.

1. **Aufgabendefinitionen und Abhängigkeiten der Aufgaben** Auf Grundlage der verschiedenen bereitgestellten Aufgabentypen werden Aufgaben definiert, die mithilfe von einem *dependency_graphs* in Abhängigkeit zueinander gesetzt werden.
2. **Definition von Entitäten und Items** Es besteht die Möglichkeit Entitäten oder Items zu definieren. Diesen können verschiedene Components zugewiesen werden die die Eigenschaften bestimmen. Mit den Entitäten und Items kann an anderer Stelle auf bestimmte Ereignisse reagiert werden und verschiedene Aktionen durchgeführt werden. Üblicherweise werden Entitäten oder Items in den Event-Handlern oder Szenario-Buildern instanziiert und verwendet.
3. **Definition Event-Handler und Szenario-Buildern** Event-Handler und Szenario-Builder werden mithilfe von Funktionen umgesetzt und dienen dazu die Level genauer zu beschreiben und auf bestimmte Ereignisse zu reagieren.

Die vorgestellten Bestandteile sind für die Entwicklung der Erweiterung von großer Relevanz. Die Bezeichner für die verschiedenen Tasks, Entitäten, Graphen, Items oder Funktionen sollen sowohl im Syntaxhighlighting beachtet werden, als auch innerhalb der Completions vorhanden sein. Hinzu kommen klassische Kontrollstrukturen und weitere Bestandteile der DSL wie z.B. Variablen-Definitionen und Deklarationen die auch beachtet werden sollen. Auch für die Fehlerdiagnose ist der Aufbau der genannten Bestandteile relevant um beispielsweise zu prüfen, ob Tasks korrekt definiert wurden.

Grundlagen

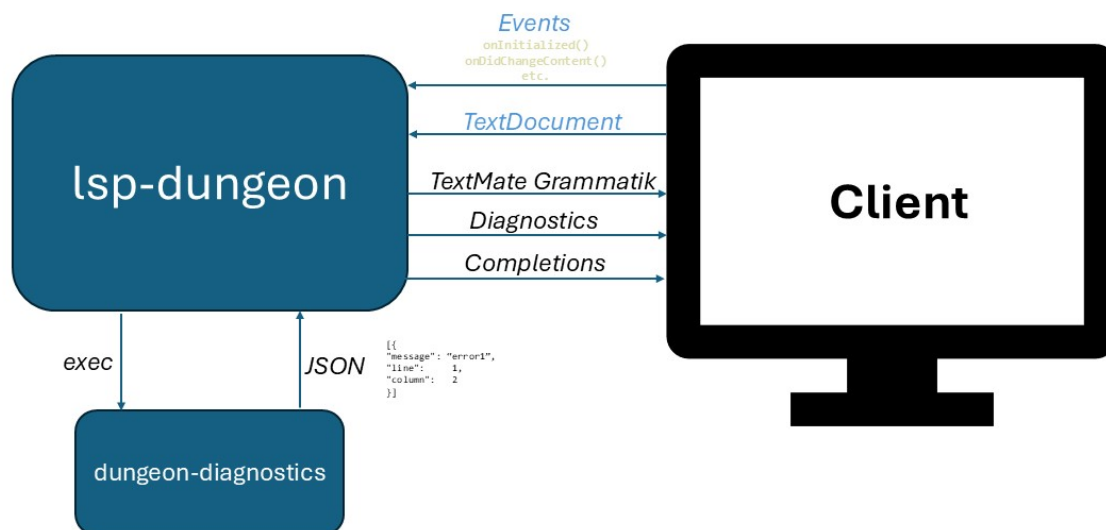
Als Grundgerüst für die Umsetzung des LSP Servers wurde das Projekt [lsp-sample von Microsoft](#) verwendet und an die Anforderungen für die Dungeon-Programmiersprache angepasst. Dieses Projekt ist eine beispielhafte Umsetzung, welche die Grundfunktionen der Bibliothek [vscode-languageserver-node](#) implementiert hat. Wie der Aufbau eines LSP-Servers aussieht wird im Kapitel [LSP Server](#) erläutert. Im Gegensatz zur Fehleranalyse und den Completions wird für das Code-Highlighting kein LSP Server benötigt,

sondern wird mithilfe einer TextMate-Grammatik definiert. TextMate Grammatiken werden in der Regel in JSON-Dokumenten beschrieben und definieren Regeln und Muster zum Einfärben bestimmter Strukturen im gegebenen Quellcode. Hierfür wurde sich am [Syntax Highlight Guide](#) orientiert. Das Code-Highlighting ist in einem eigenen kleinem Projekt (syntax-highlighting) umgesetzt und muss einzeln als weitere Extension hinzugefügt werden. Somit ergeben sich in der Gesamtheit 3 Unterprojekte:

1. **syntax-highlighting** Beinhaltet die Umsetzung zum hervorheben bestimmter Code-Bestandteile für bessere Lesbarkeit.
2. **lsp-dungeon** Implementiert die LSP-Server Extension unter Verwendung der Dungeon-Diagnostics.jar.
3. **dungeon-diagnostics** Java-Projekt, das mithilfe von ANTLR genauere Syntax und Semantik-Analysen auf dem Dungeon-Code durchführen kann. Liefert als Output die vom **lsp-dungeon** Projekt verwendete Dungeon-Diagnostics.jar.

Architektur

Wie die einzelnen Unterprojekte miteinander und mit dem Client interagieren wird in der folgenden Abbildung verdeutlicht:



Ziele

Das Hauptziel des Projekts ist die Bereitstellung von einer oder mehreren funktionierenden Erweiterungen für die Dungeon DSL. Das Projekt soll zukünftig gut erweiterbar sein, da hinzukommende Anforderungen an beispielsweise weitere semantische Analysen oder neue Completion-Komponenten gut umgesetzt werden können. Die Installation der Erweiterungen wird für die IDE Visual Studio Code bereitgestellt und genauer erläutert.

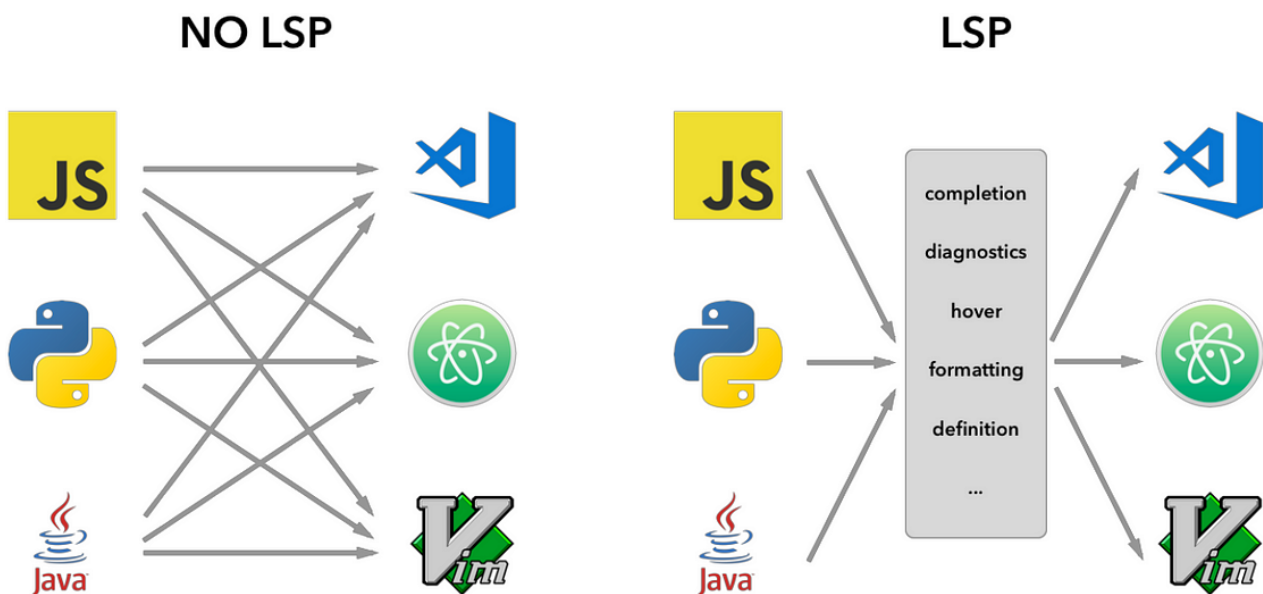
Mit Blick auf das **Syntax-Highlighting**, was die am wenigsten umfangreiche Erweiterung ist soll ein vollständiges Syntax-Highlighting umgesetzt werden, das alle relevanten Code-Bestandteile der DSL abdeckt und nicht zusätzlich erweitert werden muss.

Für die Fehleranalyse und Visualisierung der Fehler soll ein Gerüst bereitgestellt werden, welches die grundlegenden Syntax- und ausgewählte Semantik-Analysen durchführen und im Code markieren kann. Da der Bereich der Fehleranalyse von Programmiersprachen speziell im Semantik-Bereich sehr umfangreich umgesetzt werden kann ist das Hauptziel eine erweiterbare Architektur bereitzustellen, welche es ermöglicht weitere Fehleranalysen nachträglich hinzuzufügen. Hierfür soll eine geeignete Schnittstelle bereitgestellt werden, mit welcher gefundene Fehler im passenden Format an den LSP-Server übergeben werden können. Für eine detailliertere Syntaxanalyse wird die vorhandene ANTLR-Grammatik teilweise erweitert. Hierfür werden die in der DSL oft verwendeten Datenstrukturen (Tasks, Graph, Dungeon_Config, etc.) detaillierter von der Grammatik beschrieben um beispielsweise zu prüfen, ob bei der Initialisierung angegebene Feldnamen und die dazugehörigen Datentypen korrekt sind.

Bezogen auf die Completions für die DSL, sollen abhängig vom gegebenen Kontext passende Vorschläge vom Editor gegeben werden. Hierbei sollen zum einem die Grundlegende Bestandteile der DSL wie die unterschiedlichen Datenstrukturen (Tasks, Entitys, etc.) als auch vergebene Namen der initialisierten Objekte und zugewiesenen Variablen vorgeschlagen werden können. Außerdem sollen Snippets für die grundlegenden Datenstrukturen und Kontrollflussstrukturen erstellt werden können. Durch Snippets können komplette Datenstrukturen erstellt werden.

LSP Server

Das LSP Protokoll ist ein vielverbreiteter und beliebter Ansatz für die Entwicklung von Spracherweiterungen von Programmiersprachen. Dazu gehören abgesehen von den hier umgesetzten Erweiterungen (Diagnostics, Completions) auch Dinge wie Zusatzinformationen beim Hovern über bestimmte Code-Elemente, Formattierung oder die Möglichkeit zu der Definition von z.B. Funktionen oder Objekten zu springen.



Die Abbildung stellt dar, aus welchem Grund LSP ein beliebter Ansatz ist Erweiterung für Sprachen zu erstellen. LSP vereinheitlicht die Entwicklung von Erweiterung für verschiedene Sprachen und kann von verschiedenen IDEs verwendet werden. Ohne den einheitlichen Standard müsste jede IDE für jede Sprache eine eigene Extension bereitstellen. Dementsprechend kann ein einzelner Editor eine gute Unterstützung für viele verschiedene Sprachen bereitstellen. Des weiteren verfolgt LSP einen konsistenten Ablauf, was die Kommunikation zwischen dem Server und dem Client (IDE) angeht.

Umsetzung LSP Server

Wie bereits erwähnt wurde als Grundlage für den LSP Server das Projekt *lsp-sample* genutzt, welches die folgende Grundstruktur aufweist:

```
.
├── client // Language Client
│   ├── src
│   │   ├── test // End to End tests for Language Client / Server
│   │   └── extension.ts // Language Client entry point
├── package.json // The extension manifest
├── server // Language Server
│   └── src
│       └── server.ts // Language Server entry point
```

Quelle: <https://code.visualstudio.com/api/language-extensions/language-server-extension-guide>

Der relevanteste Ordner für die Umsetzung ist der Server, da *server.ts* die Datei ist, in welcher die Logik für die Fehleranalyse und die Completions implementiert wird.

Die relevantesten Stellen der *server.ts* werden im folgenden kurz vorgestellt:

1. **Imports** Zu Beginn werden die relevanten Bibliotheken importiert, die die wichtigsten Klassen und Funktionen für den LSP Server bereit stellen. Hierbei sind die wichtigsten importierten Module *vscode-languageserver/node* und *vscode-languageserver-textdocument*.

Die Bibliothek *vscode-languageserver/node* stellt alle relevanten Klassen und Funktionen für den LSP Server an sich bereit, die im Nachgang noch genauer erläutert werden. Dazu gehören unter anderem *createConnection*, *Diagnostic* und *CompletionItem*. Mit *createConnection* wird eine Connection zum Client hergestellt. Auf dieser Connection wartet der Server auf die Events *onInitialize*, *onDidChangeContent* oder *onCompletion* und reagiert in der implementierten Art und Weise. *Diagnostic* ist die Klasse die verwendet wird um gefundene Fehler oder Warnings zu verpacken und an den Client weiterzugeben. Das selbe gilt für *CompletionItem*, welches die Datenstruktur für die möglichen Completions ist die an den Client gegeben werden.

Die andere Bibliothek *vscode-languageserver-textdocument* bietet die Funktionalitäten zur Arbeit mit dem Text-Dokument, auf welches die Fehleranalyse und die Completions angewendet werden sollen. Dazu gehört den kompletten Text-Inhalt des gegebenen Dokumentes zu verwalten, die aktuelle Position des Cursors des Users im Dokument zu bekommen und den Text-Inhalt mithilfe von z.B. Regex genauer zu analysieren.

Fehleranalyse

Für die Fehleranalyse wird ANTLR benötigt um Textdokumente zu parsen und mit der definierten Grammatik zu vergleichen. Für diesen Ansatz wurde sich entschieden, da bereits eine Grammatik zur Verfügung steht, die verwendet werden kann um die Syntax der Dungeon DSL zu prüfen. Alternativ könnte der Textinhalt mithilfe von Regexen analysiert werden. Dieser Ansatz erwies sich jedoch mit Blick auf komplexere oder verschachtelte Code-Strukturen, die in der Dungeon DSL vorkommen können als ungeeignet.

Im Gegensatz zum LSP-Server, welcher als Node-Projekt umgesetzt wurde, wurde das Projekt für den Parser/Interpreter als Java-Projekt aufgesetzt. Dementsprechend wird eine Schnittstelle zwischen dem Node- und dem Java-Projekt benötigt, da die Projekte unabhängig voneinander implementiert sind. Da der

Parser/Interpreter lediglich die Aufgabe hat alle Fehlermeldungen aus einem Textdokument zurückzugeben wurde sich dafür entschieden das Java-Projekt als ausführbare JAR in das Node-Projekt einzubinden. Im Node-Projekt liegt dementsprechend nur die fertige JAR und der Nutzer hat keinen Einblick in die Funktionalität der JAR, da diese aus einem separaten Unterprojekt gebaut wurde. Dieses Unterprojekt trägt den Namen Dungeon-Diagnostics.

Dungeon-Diagnostics

Das Unterprojekt Dungeon-Diagnostics ist ein Java-Projekt, welches als Argument einen Dokumentpfad entgegennimmt und dieses Dokument parsed und mit einem Visitor weitere Analysen durchführt und gefundene Probleme ausgibt. Das Projekt ist wie folgt aufgebaut:

```
Dungeon-Diagnostics/
├── out/                                # Outputdateien
│   ├── artifacts/
│   │   ├── Dungeon_Diagnostics.jar
│   │   └── Dungeon-Diagnostics.jar
├── src/                                # Quellcode
│   ├── Main.class
│   ├── ErrorInfo.class
│   ├── CustomErrorListener.class
│   └── DiagnosticsVisitor.class
└── DungeonDiagnostics.g4                # Grammatik
```

In der Ordnerstruktur wurden einige Dateien ausgelassen, welche zusätzlich zu den aufgeführten Dateien vorhanden sind. Dabei handelt es sich um die von ANTLR generierten Dateien wie den Lexer und den Parser, welche nicht händisch angepasst werden. Der Grundstein des Projekts ist die Grammatik

DungeonDiagnostics.g4. Anhand dieser Grammatik werden alle relevanten Dokumente generiert, die für die weitere Analyse relevant sind (Lexer, Parser, Visitor, etc.). Der generierte Parser und Lexer werden wie bereits erwähnt nur generiert und nicht angepasst. Der Visitor hingegen bietet die Möglichkeit mit dem gewünschten Verhalten implementiert zu werden. Hierdurch bietet sich die Möglichkeit weitere Semantische Analysen durchzuführen.

Für eine detailliertere Analyse der oft verwendeten Strukturen (tasks, graphs und entitys) wurden diese in der Grammatik als einzelne Regeln hinzugefügt und beschrieben, welche Felder Teil dieser Strukturen sind. Auf diese Weise kann beim parsen geprüft werden, ob z.B. *description* ein Feld von *single_choice_task* ist und ob der zugewiesene Datentyp passt.

Mithilfe der Grammatik lässt sich nun überprüfen, ob ein Dokument geparsed werden kann und wo Syntax-Fehler auftreten. Mit ANTLR können für die Grammatik Lexer, Parser und Visitor erstellt werden. Diese werden in der Main.class verwendet um einen CharStream erst mit dem Lexer in Token umzuwandeln und danach mit dem Parser zu parsen. Der Inhalt des CharStreams ist der Inhalt des zu analysierenden Dokuments. Zum laden des Dokuments muss beim Aufruf des Programms ein Kommandozeilenargument mit dem Pfad zum zu analysierenden Dokument übergeben werden. Der Default-Errorlistener wurde mit einem CustomErrorlistener überschrieben, welcher alle Errors in einer Liste speichert und diese zurückgeben kann. Um den Output im LSP

Server verarbeiten zu können, werden die zurückgegeben Fehler von einer Hilfsfunktion in das folgende JSON-Format gebracht:

```
[{
  "message": string,
  "line": int,
  "column": int
}]
```

Das generierte JSON mit allen gefundenen Fehlern wird über den stdout als Schnittstelle zurückgegeben und kann auf diese Weise vom LSP-Server verwendet werden.

Verarbeitung der ermittelten Fehler

Zum aufrufen der JAR, welche die Fehleranalyse auf dem Dokument durchführt wird, wird die Funktion `exec` aus dem `child_process`-Modul verwendet um ein Shell-Kommando durchführen zu können. Außerdem wird `promisify` aus dem `utils`-Modul verwendet um die `exec`-Funktion wie einen Promise zu verwenden und asynchron aufrufen zu können.

Das Ziel ist es das Dokument, welches sich beim Client aktuell in Bearbeitung befindet bei jeder Änderung zu analysieren. Dementsprechend wird bei jedem `onDidChangeContent`- und `onDidSave`-Event die Funktion `validateTextDocument` aufgerufen, die dafür verantwortlich ist das übergebene TextDocument zu analysieren und alle auftretenden Diagnostics zurückzugeben. Hierfür wird der Pfad des übergebenen Dokuments extrahiert und beim Aufruf des Shell-Kommandos als Übergabe-Parameter übergeben. Außerdem wird der Pfad der Dungeon-Diagnostics-JAR übergeben und das Shell-Kommando wie folgt ausgeführt.

```
const { stdout, stderr } = await execPromise(`java -jar ${javaJarPath} ${filePath}`);
```

Nach der Ausführung des Shell-Kommandos wird überprüft, ob es Fehler gab oder, ob der stdout mit sinnvollem Inhalt gefüllt wurde. Ist dies der Fall und im stdout finden sich Fehler im JSON-Format, werden diese in das Format der `Diagnostic`-Klasse gebracht und zurückgegeben. Hierfür wird der stdout mithilfe von `JSON.parse` in ein output-Objekt gespeichert, welches dem Aufbau des übergebenen JSON-Inhalts entspricht. Dementsprechend kann nun mit dem folgenden Code jeder Error in die Liste der gefundenen Fehler übertragen werden:

```
output.errors.forEach((err: any) => {
    diagnostics.push({
        severity: DiagnosticSeverity.Error,
        range: {
            start: { line: err.line, character: err.column },
            end: { line: err.line, character: err.column }
        },
        message: err.message,
        source: 'ex'
    });
    console.error(`Fehler: ${err.message}, Zeile: ${err.line},
Spalte: ${err.column}`);
});
```

Über severity wird angegeben um was für eine Art von Diagnostic es sich handelt. In diesem Fall wurde nur nach Errors und nicht nach Warnings gesucht, weswegen alle Diagnostics die severity vom Typ *DiagnosticSeverity.Error* bekommen. Durch range wird festgelegt an welcher Stelle der Fehler aufgetreten ist. Außerdem wird die zugehörige Error-Message festgelegt. Nun existiert eine komplette Liste von Diagnostics, die alle relevanten Informationen für den Client beinhalten um die Fehler inklusive Fehlermeldung im Dokument zu markieren.

Code-Completions

Für die Code-Completions wird im Gegensatz zur Fehleranalyse keine Analyse des Codes mit ANTLR durchgeführt, sondern ein Regex-Ansatz verfolgt. Dieser Ansatz kann dementsprechend komplett innerhalb des Servers umgesetzt werden und benötigt keinen Schnittstellenauf. Für "intelligenter" Completion-Vorschläge könnte eine genauere Analyse des Codes mithilfe eines Visitors durchgeführt werden.

Für den verwendeten Regex-Ansatz ist das Ziel, dass je nachdem in was für einem Kontext sich der User im Dokument des Clients befindet passende Vorschläge gegeben werden. Hierfür wird auf verschiedene Completion-Kontexte geprüft, welche unterschiedliche Completion-Items ausgeben.

Completion-Kontexte

- **Bezeichner** Für diesen Kontext wird geprüft, ob das vom Kontext erwartete nächste Token ein Bezeichner ist. Dementsprechend wird geprüft, ob das Wort welches sich vor der Cursor-Position befindet ein Schlüsselwort wie z.B. `var` oder `single_choice_task` ist. In diesem Fall sollen keine Vorschläge gegeben werden, da der Bezeichner individuell bestimmt werden muss.
- **Zeilenanfang** An Zeilenanfängen sollen Schlüsselwörter vorgeschlagen werden.
- **Typdefinition** Nach `:`, `('` oder `,` sollen Typdefinitionen angezeigt werden wie *string*, *int* und *boolean*.
- **Zuweisung** Nach `=` sollen bereits deklarierte Variablen oder Objekte vorgeschlagen werden. Außerdem sollen definierte Funktionen und Dungeon DSL Standard-Funktionen wie `instantiate()` vorgeschlagen werden.
- **Objektinitialisierung** Wenn ein Objekt initialisiert wird sollen an den Zeilenanfängen zwischen den geschweiften Klammern die Felder der dazugehörigen Klasse angezeigt werden.
- **Punktnotationen** Wenn ein `.` nach dem Bezeichner eines initialisierten Objekts eingegeben wird, sollen alle möglichen Felder des dazugehörigen Objekt-Typen zurückgegeben werden.

Snippets

Mit Snippets können komplette Code-Strukturen erstellt werden. Außerdem kann der User mit 'Tab' durch die anzupassenden Stellen der Snippets springen und diese direkt definieren. Auf diese Weise können z.B. schnell Tasks erzeugt und definiert werden. Diese Felder können über `${index:description}` in den Snippet-Strings eingefügt werden. Snippets werden unabhängig vom Kontext angezeigt und sind an ein label gebunden, welches bestimmt, bei welchen Zeichenfolgen das Snippet vorgeschlagen wird.

Syntax-Highlighting

Für das Syntax-Highlighting der Dungeon DSL wurde eine TextMate-Grammatik zusammen mit einer Konfigurationsdatei verwendet, die die Grundlage für die farbliche Hervorhebung verschiedener

Sprachbestandteile und die Code-Editor-Funktionalität legt. Für die Verwendung in der Extension musste die `package.json` unter dem Punkt *contributes*

TextMate-Grammatik (`dng.tmLanguage.json`)

Die TextMate-Grammatik definiert verschiedene Muster (Patterns) zur Erkennung von Schlüsselwörtern, Operatoren, Zahlen, Zeichenketten, Kommentaren und Variablen. Diese Muster ermöglichen es, Teile des Codes in verschiedenen Farben und Stilen darzustellen.

scopeName: "source.dng" gibt den Geltungsbereich der DSL an. Dieser wird verwendet, um das Syntax-Highlighting auf Dateien mit der Endung `.dng` anzuwenden.

patterns: Eine Liste von Regeln zur Erkennung von spezifischen Sprachmerkmalen. Ein Pattern besteht in der Regel aus *name* und *match*.

- **Keywords:** Die Regel `keyword.control.dng` verwendet einen regulären Ausdruck `(\b(fn|for|while|if|else|return|entity_type|item_type|var|graph|type|true|false)\b)`, um Schlüsselwörter wie `fn`, `for`, `while` usw. zu erkennen und als Kontrollstrukturen zu markieren. Operatoren: Mit `keyword.operator.dng` werden Operatoren wie `->`, `=`, `!=`, `==` usw. hervorgehoben.
- **Zahlen:** `constant.numeric.dng` hebt numerische Werte hervor, sowohl ganze Zahlen als auch Dezimalzahlen. Zeichenketten: Es gibt separate Muster für einfache (`string.quoted.single.dng`) und doppelte (`string.quoted.double.dng`) Anführungszeichen. Innerhalb der Zeichenketten werden Escape-Sequenzen (`\\`) zusätzlich erkannt und als `constant.character.escape.dng` hervorgehoben.
- **Kommentare:** Sowohl einzeilige Kommentare (`comment.line.double-slash.dng` mit `//`) als auch mehrzeilige Blockkommentare (`comment.block.dng` mit `/* ... */`) werden hervorgehoben.
- **Variablen:** Mit `variable.other.dng` werden Variablen erkannt, die einem typischen Bezeichner-Muster folgen `(\b[a-zA-Z][a-zA-Z0-9]*\b)`.

Language Configuration (`language-configuration.json`)

Diese Datei definiert ergänzende Informationen für die Bearbeitungsfunktionen im Editor:

- **Kommentare:** Die Dungeon DSL unterstützt einzeilige (`//`). Diese werden hier für die automatische Formatierung und Kommentierung definiert.
- **Klammerpaare:** Die Liste unter "brackets" gibt die gültigen Klammerpaare an, die in der DSL verwendet werden, wie `{}`, `[]` und `()`.
- **Automatisches Schließen von Zeichen:** "autoClosingPairs" gibt an, welche Zeichenpaare automatisch geschlossen werden, wenn das erste Zeichen getippt wird. Dies gilt für Klammern sowie für Anführungszeichen.
- **Einfassen von Text:** "surroundingPairs" gibt an, welche Zeichenpaare verwendet werden können, um einen markierten Text einzufassen, wie Klammern oder Anführungszeichen.

Installationen

JAR Updaten

Falls Änderungen am Verhalten der Diagnostics durchgeführt werden und eine neue JAR erzeugt wurde, muss diese JAR in den Ordner `server\jars` eingefügt und die alte JAR gelöscht werden.

Eine neue JAR kann erstellt werden indem im Projekt `dungeon-diagnostics` das Artifact `dungeon-diagnostics:jar` gebaut wird. Hierfür in IntelliJ Build > Build Artifacts... und `dungeon-diagnostics:jar` auswählen. Danach befindet sich die JAR im Ordner `out/artifacts/Dungeon_Diagnostics_jar`. Diese kann kopiert werden und an die oben genannte Stelle im `lsp-dungeon` Projekt eingefügt werden.

Erstellen der .vsix-Datei:

Erstelle die .vsix-Datei:

Um eine .vsix-Datei zu erstellen, wird das `vsce`-Tool verwendet. Falls dies noch nicht installiert wurde kann es mit dem folgenden Befehl installiert werden: `npm install -g vsce`

Danach in das Verzeichnis der Extension navigieren und den folgenden Befehl ausführen: `vsce package`

Dies erstellt eine .vsix-Datei im aktuellen Verzeichnis.

Installation der .vsix-Datei:

Um die .vsix-Datei in VS Code zu installieren:

VS Code öffnen. Zu Extensions gehen (Ctrl+Shift+X). Auf das Drei-Punkte-Menü oben rechts klicken und „Extension aus VSIX installieren...“ wählen. Wähle die erstellte .vsix-Datei aus und installiere sie. Extension ist nun lokal in VS Code installiert und kann wie jede andere Extension verwendet werden.

Verwendete Technologien

ANTLR

Für die Syntax-Analyse wurde [ANTLR \(v4.13.2\)](#) als Bibliothek in IntelliJ verwendet.

vscode-languageserver/node

Für die Implementierung des LSP Servers in Node wurde die Library [vscode-languageserver/node](#) verwendet. Diese stellt alle wichtige Klassen, Funktionen und Handler bereit die zur Implementierung des LSP notwendig sind bereit.

vscode-languageserver-textdocument

[vscode-languageserver-textdocument](#) ist ein Node Package für die Verarbeitung des Inhalts von Text-Dokumenten.

TextMate

Quellen
