Vergleich der Extension-APIs in Visual Studio Code und IntelliJ IDEA

Philipp Seiringer

BACHELORARBEIT

eingereicht am

Fachhochschul-Bachelorstudiengang

Software Engineering

in Hagenberg

im Februar 2024

Betreuung:

Dr. Josef Pichler

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt. Die vorliegende, gedruckte Arbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Hagenberg, am 1. Februar 2024

Philipp Seiringer

Inhaltsverzeichnis

Kurzfassung

Abstract

This should be a 1-page (maximum) summary of your work in English.

Kapitel 1

Einleitung

1.1 Motivation

SoftwareentwicklerInnen arbeiten täglich mit verschiedensten Werkzeugen und Entwicklungsumgebungen, sogenannten IDEs (=Integrated Development Environment). Diese Plattformen bieten teils sehr unterschiedliche Funktionalitäten, die die Softwareentwicklung erleichtern sollen. Dabei bieten sie Unterstützung für verschiedenste Programmiersprachen und Technologien und binden zahlreiche Werkzeuge für spezifische Anwendungsfälle ein. Aufgrund des immer rascher werdenden Entstehens von neuen Technologien bieten mehr und mehr IDEs Möglichkeiten zur Entwicklung von eigenen Plugins, welche dann auch an andere EntwicklerInnen bereitgestellt werden können. So können in kürzester Zeit neue Technologien unterstützt werden und EntwicklerInnen haben selbst die Macht darüber zu entscheiden welche Plugins sie nutzen möchten und welche nicht.

Vor der Entwicklung solcher Plugins ist es wichtig zu entscheiden für welche IDE das Plugin erstellt werden soll. Dabei spielen Aspekte wie zum Beispiel die Einfachheit und Flexibilität in der Entwicklung, der Umfang an angebotener Funktionalität, die Möglichkeit die Nutzerinteraktion und somit die User Experience zu steuern und viele weitere eine Rolle. Diese Bachelorarbeit versucht in diesen Bereichen einen Überblick zu schaffen und vergleicht hierfür die Plugin Entwicklung in zwei der momentan beliebtesten IDEs, Visual Studio Code und IntelliJ IDEA. Durch den Vergleich der beiden Produkte und dem Herausarbeiten und Aufbereiten der Unterschiede wird es anderen EntwicklerInnen erleichtert diese Entscheidung zu treffen.

1.2 Ziel

1.3 Aufbau

Kapitel 2

Grundlagen der Plugin Entwicklung

2.1 Entwicklungsumgebungen

2.1.1 Visual Studio Code

2.1.2 IntelliJ IDEA

2.2 Programmiersprachen

2.2.1 TypeScript

Die TypeScript Programmiersprache wurde erstmalig am 1. Oktober 2012 [17] von Microsoft in Form eines open-source Projekts veröffentlicht. Designed wurde sie von Anders Hejlsberg, der auch an der Entwicklung von C# beteiligt war.

Die grundsätzliche Idee der Sprache ist, eine typsichere, kompilierte, und somit bessere Version von JavaScript zu sein. JavaScript ist aufgrund des Erfolgszugs des Internets zu einer sehr wichtigen Sprache geworden und war auch schon 2012 aus den TOP Listen für Programmiersprachen nicht mehr wegzudenken [8][21] [22]. Webseiten setzen heute sehr stark auf JavaScript, um durch interaktive Elemente die User Experience zu verbessern oder um neue Funktionalität anbieten zu können. Durch das Node.js runtime environment kann JavaScript nicht mehr nur im Browser verwendet werden, sondern es können auch Desktop, Server oder Mobile Anwendungen in JavaScript entwickelt werden. Durch diesen großen Umfang an Möglichkeiten die JavaScript dadurch bietet werden natürlich auch immer größere Projekte damit entwickelt. Und hier kommen die großen Schwächen von JavaScript immer mehr zu tragen. Je größer die Projekte werden und je mehr EntwicklerInnen an einem Projekt mitarbeiten, desto mehr Fehler entstehen aufgrund der fehlenden Typsicherheit und des fehlenden Compilerschrittes. Diese Schwachstellen versucht TypeScript nun auszubessern.

TypeScript code wird mithilfe des TypeScript Compilers „tsc“ in einfache JavaScript Dateien transpiliert. Dadurch kann auf die Popularität und Verbreitung von JavaScript aufgebaut werden und TypeScript ist überall dort verwendbar, wo JavaScript ausführbar ist. Weiters ist TypeScript ist ein Superset von JavaScript. Es gilt also: „Any valid .js file can be renamed .ts and be compiled with other TypeScript file.” [23].

Jedoch bietet TypeScript eine Menge von Vorteilen gegenüber ihrer Basissprache.

- Durch den Kompilierschritt mit dem tsc Compiler wird der Code vor der Ausführung automatisch auf Validität geprüft. Es entfällt also die Notwendigkeit für einen zusätzliches Linting Tool wie JSLint. Dieser Compile-Schritt kann natürlich auch in eine CI/CD Pipeline eingebunden werden, um auch bei Merges Feedback über die Validität des Codes zu erhalten.
- Durch die statische Typisierung können Missverständnisse über die Verwendung von Variablen vermieden werden. Auch die Unterstützung durch verschiedene IDEs, zum Beispiel mittels IntelliSense kann durch die Typen verbessert werden. Dies ist nicht nur bei der Zusammenarbeit hilfreich, sondern kann auch die Arbeit jeder einzelnen EntwicklerIn beschleunigen.
- In TypeScript können Klassen erstellt werden, deren Properties mit Zugriffsmodifikatoren (private/public) versehen sind.
- TypeScript unterstützt Vererbung, Interfaces und generische Programmierung.
- In TypeScript können bereits bestehende JavaScript Bibliotheken wiederverwendet werden. Weiters ist es möglich durch zusätzliche Dateien Typinformationen zu den bestehenden Bibliotheken zu liefern.

2.2.2 Java

Die Entwicklung der Programmiersprache Java begann im Jahr 1991 und sie wurde von den James Gosling, Mike Sheridan und Patrick Naughton designed. [24] Java wurde erstmals im Jahr 1995 von Sun Microsystems veröffentlicht. Im Januar 2010 wurde Sun Microsystems dann von der Oracle Corporation übernommen, welche seitdem auch Java weiterentwickelt.

Das Design und vor Allem die Syntax der Sprache war stark von C und C++ inspiriert, um anderen Entwicklern einen leichten Umstieg auf das neue Java zu ermöglichen. Allerdings versuchte Java die teils sehr komplexen (wenn auch effektiven) Sprachfeatures von C++ etwas zu vereinfachen. Java sollte eine simple, objektorientierte und robuste Sprache werden. Die Funktionalität die Java zu dem großen Erfolg verhalf, den sie später hatte, war das „write once, run anywhere“ (WORA) Prinzip. Im Gegensatz zu den zuvor gängigen Programmiersprachen muss Java nämlich für bestimmte Hardwarearchitekturen kompiliert werden. Java Programme werden zu einer Art Zwischensprache, dem sogenannten Java Bytecode kompiliert. Dieser Bytecode kann dann von einer Java Virtual Machine (JVM) ausgeführt werden. Diese JVM ist im Grunde ein eigenständiges Programm welche mit dem Java Runtime Environment (JRE) mitgeliefert wird. Ein einmal kompiliertes Java Programm kann also auf allen Geräten ausgeführt werden, auf denen ein passendes JRE installiert ist. So ist es zum Beispiel auch möglich Java für die Entwicklung von Android nativen Apps auf Mobilgeräten zu benutzen.

Ein weiterer Vorteil gegenüber älteren Sprachen wie C++ ist die automatisierte Speicherverwaltung. Diese funktioniert mithilfe eines sogenannten „garbage collectors“ welcher nicht mehr benötigten Speicher am Heap bereinigt und freigibt. Man kann also beliebig neue Objekte im Speicher allokieren und muss sich nicht um die deallokierung der zuvor erstellten Objekte kümmern. Auf diese Weise können häufige Programmierfehler wie Memory Leaks fast vollständig unterbunden werden.

Java unterstützt sowohl das objektorientierte, das prozedurale als auch das funktionale Programmierparadigma. Der Fokus liegt allerdings stark auf der Objektorien-

tierung. Dabei bietet Java Funktionalitäten zur Abstraktion durch Verwendung von Klassen, Information Hiding mithilfe von Zugriffsmodifikatoren (public/private/protected/package), Vererbung, Interfaces, Polymorphismus, Überladen von Methoden, generischer Programmierung, Exception Handling und vieles mehr.

2.3 Aufbau der Plugin API

2.3.1 Visual Studio Code

2.3.2 IntelliJ IDEA

2.4 Funktionalität der Plugin API

2.4.1 Visual Studio Code

2.4.2 IntelliJ IDEA

2.4.3 IntelliJ Flora Plugins

In der Plugin Dokumentation von JetBrains wird zu Beginn empfohlen sich noch einmal gründlich zu überlegen, ob man für die von einem gewünschte Funktionalität wirklich ein vollwertiges Plugin benötigt. Häufig kommt es nämlich vor, dass nur bestimmte kleine Tasks innerhalb des IDEs automatisiert werden sollen. Hierfür schlägt JetBrains einige leichtgewichtige Alternativen vor. Eine nennenswerte Alternative ist das „Flora Plugin“ für das IntelliJ IDEA.

Flora kann über die Einstellungen des IntelliJ IDEA im Abschnitt „Plugins“ installiert werden.

Das Plugin sucht dann in den geöffneten Projektverzeichnissen nach ausführbaren JavaScript oder Kotlin Script „micro plugin“ Dateien. Diese müssen sich in einem Ordner namens „.plugins“ befinden und auf „.plugin.js“ oder „.plugin.kts“ enden. Innerhalb diese Plugin Dateien kann über die Variable „ide“ auf die angebotene Schnittstelle zugegriffen werden. Diese erlaubt es unter anderem Actions, Keyboard Shortcuts, Services und ToolWindows zu erstellen.

Flora Plugins bieten sich vor allem dann an, wenn eine projektspezifische Aufgabe automatisiert werden soll. Hier sind vor Allem die Leichtgewichtigkeit der Plugins und die Schnelle, mit der ein einfaches Plugin entwickelt werden kann, von großem Vorteil. Weiters spricht für diesen Anwendungsfall, dass der Plugin Code direkt im Projektordner abgelegt wird und somit auch in einem Version Control System wie Git mit abgelegt werden kann.

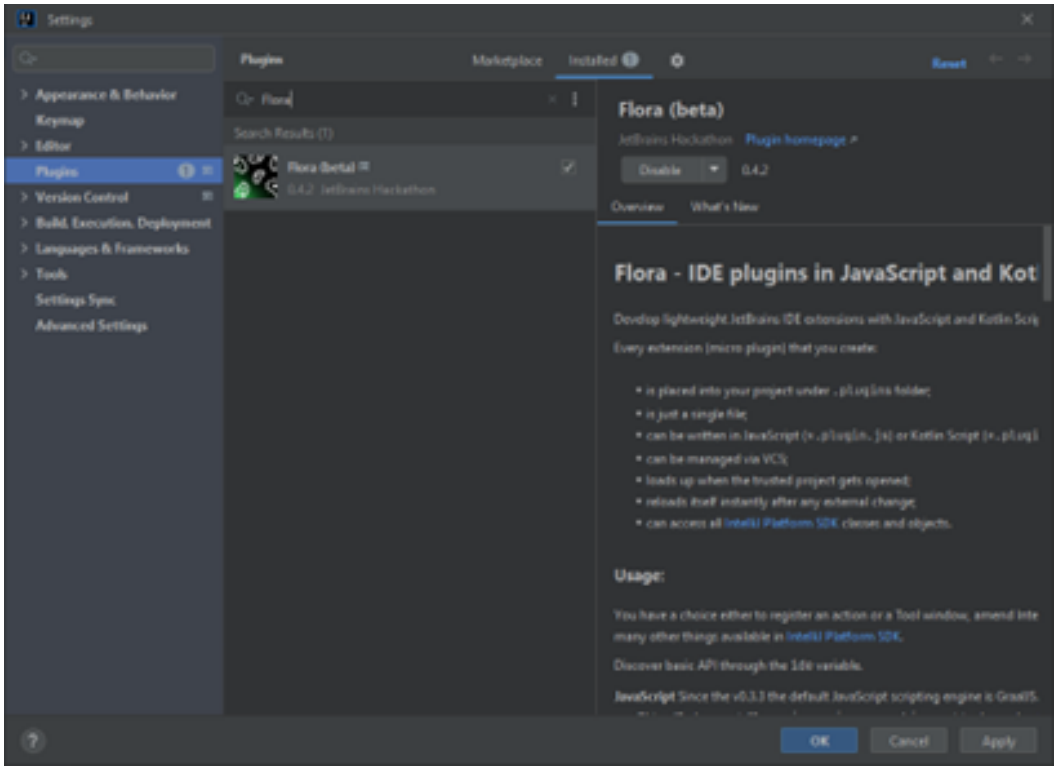


Abbildung 2.1: Flora Plugin im IntelliJ Plugin Marketplace.

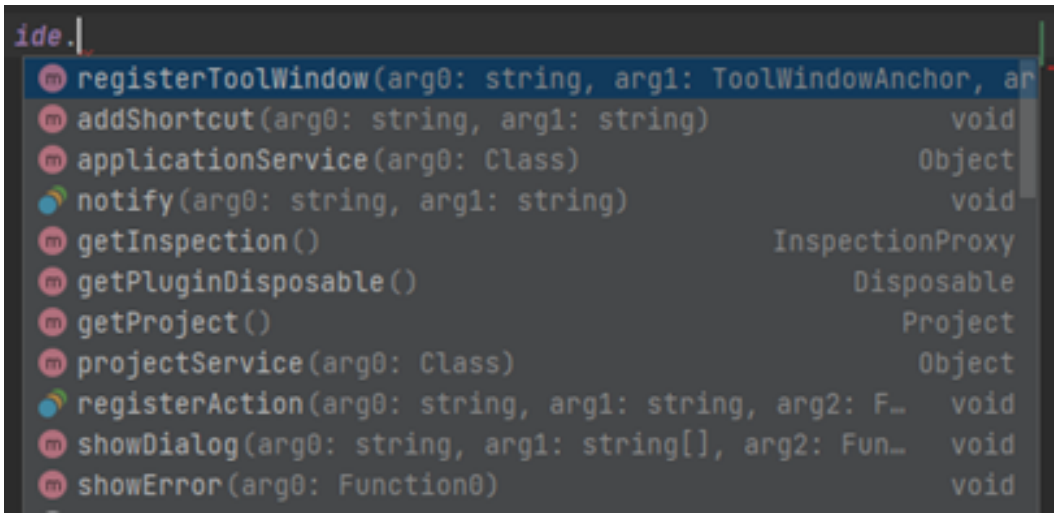


Abbildung 2.2: Übersicht über die API des Flora Plugins.

Kapitel 3

Anforderungen an den Prototyp

3.1 Aufbau

Kapitel 4

Entwicklung des Prototpys für Visual Studio Code

4.1 Design

4.2 Implementierung

4.2.1 Aufsetzen des Projektes

Aufbau der Ordnerstruktur

4.2.2 Entwicklung

4.3 Tests

4.4 Publishing

4.5 CI/CD

Kapitel 5

Entwicklung des Prototpys für IntelliJ

5.1 Design

5.2 Implementierung

5.2.1 Aufsetzen des Projektes

Aufbau der Ordnerstruktur

5.2.2 Entwicklung

5.3 Tests

5.4 Publishing

5.5 CI/CD

Kapitel 6

Bewertungskriterien

6.1 Popularität der Entwicklungsumgebung

6.1.1 Visual Studio Code

6.1.2 IntelliJ IDEA

6.2 Performance

6.2.1 Visual Studio Code

6.2.2 IntelliJ IDEA

6.3 Feature Umfang

6.3.1 Visual Studio Code

6.3.2 IntelliJ IDEA

6.4 Intuitivität der API

6.4.1 Visual Studio Code

6.4.2 IntelliJ IDEA

6.5 Dokumentation der API

6.5.1 Visual Studio Code

6.5.2 IntelliJ IDEA

6.6 Testbarkeit des Plugins

6.6.1 Visual Studio Code

6.6.2 IntelliJ IDEA

6.7 Möglichkeiten des Publishings

6.7.1 Visual Studio Code

6.7.2 IntelliJ IDEA

6.8 Installationsprozess des Plugins

Kapitel 7

Vergleich der Kriterien

7.1 Popularität der Entwicklungsumgebung

7.1.1 Visual Studio Code

7.1.2 IntelliJ IDEA

7.1.3 Vergleich

7.2 Performance

7.2.1 Visual Studio Code

7.2.2 IntelliJ IDEA

7.2.3 Vergleich

7.3 Feature Umfang

7.3.1 Visual Studio Code

7.3.2 IntelliJ IDEA

7.3.3 Vergleich

7.4 Intuitivität der API

7.4.1 Visual Studio Code

7.4.2 IntelliJ IDEA

7.4.3 Vergleich

7.5 Dokumentation der API

7.5.1 Visual Studio Code

7.5.2 IntelliJ IDEA

7.5.3 Vergleich

7.6 Testbarkeit des Plugins

7.6.1 Visual Studio Code

7.6.2 IntelliJ IDEA

Kapitel 8

Conclusion

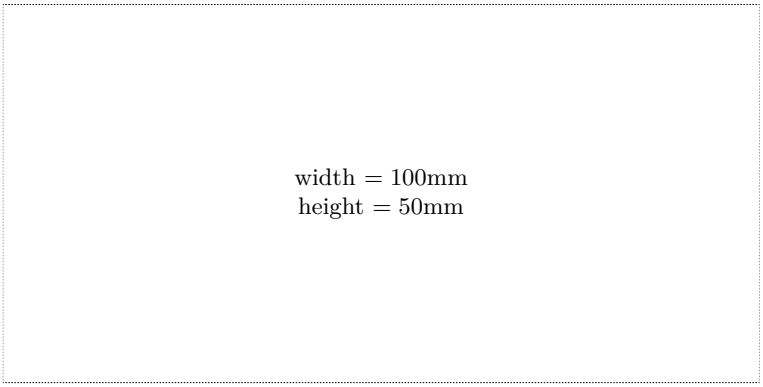
Anhang A

Technische Informationen

Quellenverzeichnis

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —