

```
val pluginName = plugin.getName
logger.info s"Loading plugin '$pluginName' now."

// Check if the plugin folder contains a build.sbt and a src folder
val pluginContent = plugin.listFiles

if (!pluginContent.exists(_.getName.equalsIgnoreCase("build.sbt"))) {
  logger.warn "Unable to find build.sbt-file. "
}

if (!pluginContent.exists(_.getName.equalsIgnoreCase("src"))) {
  logger.warn "Unable to find src folder. "
```

Eine neue Plugin-Architektur für Code Overflow

📅 16. Januar 2018 (<http://sebinside.de/2018/01/16/eine-neue-plugin-architektur-fuer-code-overflow/>)

👤 seb (<http://sebinside.de/author/seb/>) ➦ Coding (<http://sebinside.de/category/coding/>)

Code Overflow ist der Coding-Livestream von Andre, Dennis und mir – alle drei Master-Informatik-Studenten am KIT, Karlsruhe. Unser erstes und größtes Projekt ist ein Framework namens „Chat Overflow“, das in Quasi-Echtzeit den Livestream-Chat auswertet. Nach unserem zweiten Event im November 2017 haben wir uns dazu entschieden, die Software grundlegend neu zu entwerfen. Die Plugin-Architektur, die ich dafür Anfang des Jahres entworfen habe, möchte ich heute vorstellen.

„Chat Overflow hat das Ziel Livestreams interaktiver zu gestalten, in dem der Chat in Echtzeit ausgewertet und die Nachrichten in vielfacherweise weiterverarbeitet werden.“ –

So oder so ähnlich könnte man das, was wir im letzten Jahr programmiert haben, in einem Satz zusammenfassen. Als wir 2016 mit der Entwicklung begonnen hatten, wussten wir allerdings noch nicht, auf was wir uns einlassen:

1. Die **Komplexität** der Software war höher als erwartet: Den Livestream-Chat praktisch auswerten, braucht dann doch etwas mehr als nur ein paar Nachrichten zu lesen und ein bisschen was auf der Konsole auszugeben. Schon

früh haben wir dafür ein simples Framework entworfen, was uns später viel Arbeit abgenommen hat.

2. Das **Interesse** war viel höher als erwartet: Wir hätten nicht gedacht, dass Chat Overflow ein so großen Erfolg mit sich bringt. Klar, wenn der Chat zum Spammen aufgerufen wird, lässt er sich das nicht zwei Mal sagen, aber dass wir sowohl auf YouTube als auch Twitch auf Coding-Streams in Deutschland jeweils weit mehr als 1000 Zuschauer haben würden, hatten wir uns nicht vorgestellt.
3. Die **Möglichkeiten** von Chat Overflow sind sogar sehr viel höher als erwartet: Seit dem Stream im November habe ich schon wieder über 30 neue Anwendungsfälle für die Auswertung des Livestream-Chats aufgeschrieben, und da draußen gibt es noch so viel mehr... Chat-Nachrichten auszuwerten und dabei fast alle Metadaten zu ignorieren, ist erst der Anfang.

Aufgrund dieser Punkte – und der Tatsache, dass es einfach ein mega cooles Projekt ist – haben wir uns dazu entschieden, nicht nur die Entwicklung fortzuführen, sondern Chat Overflow von Zeile 1 grundlegend **neu zu entwickeln**.

Die neue Chat Overflow Architektur

Ich möchte in diesem Blog-Post die neue Architektur nicht im Detail erklären. Zum einen würde das den Umfang wesentlich sprengen, zum anderen ist hier aber noch gar nicht alles fertig. Bis jetzt haben wir nur skizziert, was das Framework alles mitbringen muss – und das ist so einiges mehr, als bisher erwartet. Hier ein paar Punkte:

- Erweiterte Nutzung von **Metadaten** über Chat-Nachrichten hinaus
- Keine Beschränkung auf Chatnachrichten, sondern auch von **Events** wie Subs, Donations usw.
- **Keine Exklusivität** eines gleichzeitigen Providers (z.B. Twitch), sondern parallele Nutzung verschiedener Quellen
- Keine Exklusivität eines Auswertungs-Projekts, sondern die Möglichkeit viele **Projekte gleichzeitig** auszuführen, inklusive anwenderfreundlicher Oberfläche, Einstellungen usw.

- Aufteilung von Framework, API und Auswertungs-Projekt, inklusive dynamischem Nachladen und getrennter
- Entwicklung mit Hilfe einer neuen **Plugin-Architektur**

Während die Punkte 1-4 vernünftig durch Vererbung und Multithreading umgesetzt werden können, hat uns die Plugin-Architektur tatsächlich vor eine **Herausforderung** gestellt. Wie entwirft man so eine Architektur in Scala auf der JVM mit Hilfe von SBT, um eine hohe Unabhängigkeit und einen schnellen Workflow zu ermöglichen? Das habe ich mir Anfang des Jahres angeschaut.

Grundlagen

Fangen wir von vorne an, denn das hier ist alles tatsächlich komplexer als ich ursprünglich gedacht hatte. Viele Stunden habe ich mit Google verbracht, weil sich mit der Kombination von Scala + Java-Plugin-Architektur tatsächlich noch nicht so viele beschäftigt haben. In diesem Kapitel möchte ich deswegen zwei Dinge klären: Zum einen die **zugrundeliegende Technik**, zum anderen den groben Aufbau der Architektur. Der Code ist natürlich öffentlich auf Github (<https://github.com/sebinside/PluginFrameworkTest>). Fangen wir mit der Technik an:

- **Scala.** Wundert niemanden, Scala ist seit 2 Jahren die Sprache meiner Wahl #1, ich entwickle quasi (fast) alles in Scala. Scala ist mein Java, mein Python, ...
- **SBT.** „*Scala Build Tool*“ ist die Logik hinter der Herstellung, dem „Build“ eines Scala-Projekts. SBT kümmert sich um die Abhängigkeiten (und ist hierbei Maven/Ivy-kompatibel) und SBT bietet hierfür auch eigene DSLs (Domain specific Languages, also spezialisierte Programmiersprachen für einen Anwendungsfall) an. Das Spannende ist: SBT ist auch in Scala programmiert. Unterm Strich programmiert man sich hier als in Scala ein Programm, dass ein Scala Programm baut 😊
- **Multi-Projekte.** Multi-Projekte / Sub-Projekte sind ein Feature von SBT. Letztlich ermöglicht dieses Feature, dass sowohl Framework, API als auch Plugins im selben Ordner liegen können und trotzdem als eigenständige Module gehandhabt werden.

- **Git.** Logisch, müssen wir nicht weiter darüber reden, dass eigene Plugins auch eigene Repositories haben können (in meiner Testimplementierung nicht weiterverfolgt)
- **JVM und Classloader.** Das ist mit der kniffligste Part. Mit Hilfe eines eignen Classloaders werden zur Laufzeit des Frameworks alle Plugins, die sich in eigenen JAR-Dateien befinden, nachgeladen. Hierfür müssen beide Seiten eine API implementieren, der Rest ist dann Reflection-Logik.

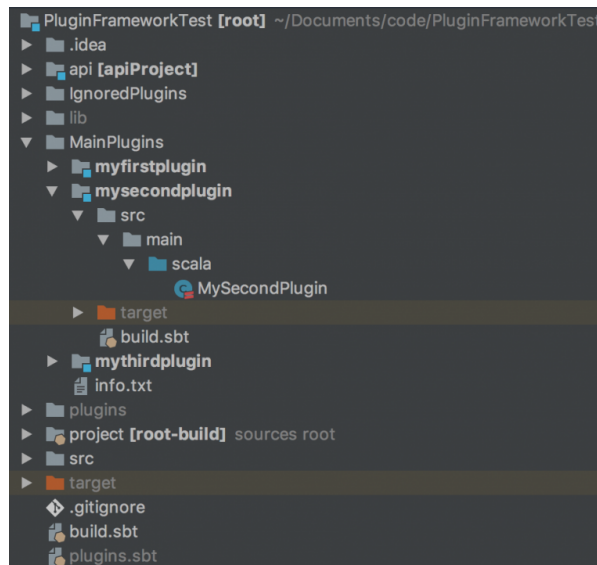
Die Entwicklung der Architektur war **zweigeteilt**: Die Build-Umgebung SBT musste im ersten Schritt erweitert werden, um Sub-Projekte (Plugins) korrekt zu erkennen, zu kompilieren, zu verpacken und zu exportieren. Im zweiten Teil werden die zuvor verpacken Plugins dann mit eigener Logik geladen und ausgeführt.

Der große Vorteil dieser Architektur: **Minimale Bindung**. Sowohl Framework als auch Plugin hängen nur von einem separaten API-Projekt ab und können separat voneinander erstellt werden. So kann beides beim Endnutzer ohne weiteren Aufwand variabel kombiniert werden. Schauen wir uns jetzt die einzelnen Schritte genauer an.

Erstellen und Suchen

Alles beginnt mit dem Erstellen (**SBT create**) eines neuen Subprojekts (aka. Plugin) im Verzeichnis des Frameworks. Wie dieses Projekt von Git verwaltet wird, kann natürlich frei eingestellt werden. Grundsätzlich könnte man den Prozess des Erstellens auch jedes Mal von Hand machen – aber natürlich habe ich hier für einen SBT Task erstellt.

Ein Sub-Projekt befindet sich in einem **eigenen Ordner**, hat eine eigene Build-Definition, eigenen Quelltext und eigene Tests. Theoretisch ist es also ein eigenständiges Projekt, das auch ohne Framework lauffähig wäre – auch wenn das natürlich keinen Sinn ergibt.



Auf dem Bild sieht man die Ordner Struktur des Testprojekts. Es kann mehrere Plugin-Basisordner geben in denen sich die gerade beschriebenen Plugins befinden. Im nächsten Schritt werden automatisiert Plugins gesucht (**SBT fetch**) und die Build-Definition des Frameworks aktualisiert. Auch wiederum etwas, was man auch von Hand machen könnte. Für die eben vorgestellte Ordner Struktur wird folgende Build-Datei erstellt:

```
1 lazy val myfirstplugin = (project in file("MainPlugins/myfirstplugin")).dependsOn(apiProject)
2 lazy val mysecondplugin = (project in file("MainPlugins/mysecondplugin")).dependsOn(apiProject)
3 lazy val mythirdplugin = (project in file("MainPlugins/mythirdplugin")).dependsOn(apiProject)
4
5 lazy val apiProject = project in file("api")
6
7 lazy val root = (project in file(".")).aggregate(apiProject, myfirstplugin, mysecondplugin, mythirdplugin)
```

Auffällig sind hier zunächst die einzelnen Variablen. Jedes (Sub-) Projekt erhält seine eigene Build-Variable. Zuletzt wird auch das Root-Projekt erstellt und mit den Sub-Projekten **aggregiert**. Dieser Schritt ist notwendig, damit beim Neu-Kompilieren und Ausführen des Frameworks auch alle Plugins auf Änderungen untersucht und ggf. neu erstellt werden.

Interessant ist hier auch der **dependsOn()**-Teil. Dieser fügt bei jedem Projekt eine Abhängigkeit zum API-Projekt hinzu, welche später für das Laden der Plugins benötigt wird. SBT kümmert sich beim Start dann automatisch darum, dass die Abhängigkeit zur API in den Classpath eingefügt wird.

Nachdem die Build-Datei für alle Plugins durch den selbstentwickelten SBT-Task aktualisiert wird, erfolgt ein manueller **Reload** von SBT. Das sorgt dafür, dass SBT alle Build-Dateien (also eben auch die gerade eben frisch generierte Plugin-Datei) neu geladen und auf Änderungen untersucht werden. Somit sollten alle Sub-Projekte (= Plugins) nach diesem Schritt korrekt erkannt worden sein.

Nach diesem Schritt erfolgt das Verpacken und Exportieren.

Verpacken und Exportieren

Nach dem Ausführen von `sbt fetch` sind alle Plugins registriert und auch die Abhängigkeiten zum Framework geklärt. Beim kompilieren mit SBT werden veränderte Plugins automatisch mit-kompiliert, beim Verpacken durch ***sbt package*** automatisch in JAR-Dateien ins ***target***-Verzeichnis verpackt.

Der letzte, neu entwickelte SBT-Task (***sbt copy***) kümmert sich jetzt um den **Export** der frischen JAR-Dateien, welche die Plugin-Logik beinhalten. Diese Dateien müssen in einem separaten Verzeichnis gesammelt werden, damit das Framework zur Laufzeit weniger Arbeit hat. Dieses Vorgehen macht es auch möglich, dass bereits kompilierte Plugins manuell hinzugefügt werden können, sowohl während der Entwicklung, als auch beim Anwender.

Wirklich spannend ist das aber nicht: Die Target-Verzeichnisse aller registrierter Plugins werden auf JAR-Dateien untersucht, diese werden anschließend in zwei eigenen Plugin-Ordern zusammen kopiert. Damit ist der SBT-Teil der Entwicklung abgeschlossen.

Java Plugin-Architektur

Was jetzt noch fehlt, ist das eigentliche **Nachladen** der zuvor erstellen Plugins. SBT kümmert sich zwar darum, dass die Plugins zur Laufzeit auch wirklich in aktueller Version bereitstehen, aber das Laden und Ausführen ist natürlich Job des Frameworks (sprich: Chat Overflow).

Ich habe hier im Titel bewusst **Java** dazugeschrieben, denn hierbei ist quasi keine gesonderte Scala-Logik notwendig. Die Idee der Architektur lässt sich schnell zusammenfassen:

- Jedes **Plugin** implementiert ein Java Interface (erfüllt Scala Trait)
- Das **Framework** implementiert ein Java Interface (erfüllt Scala Trait)
- Beide Interfaces (Traits) werden in einer gesonderten API definiert, die beide Teile kennen
- Beim Start des Frameworks werden alle JAR-Dateien (= Plugins) auf diese Implementierungen untersucht und zur Laufzeit zusammengesteckt. Fertig!

Als Vorbild hat mir dieser (<https://www.java-blog-buch.de/d-plugin-entwicklung-in-java/>) Post gedient, für Chat Overflow werden wir hier aber nochmal viel erweitern müssen.

Fazit

Ich hatte mir für die Entwicklung des Plugin-Frameworks zwei Ziele gesetzt:


1. **Maximale Unabhängigkeit:** Bis auf die Abhängigkeit zum API-Projekt sollen alle Teile (Framework und einzelne Plugins) vollständig unabhängig voneinander entwickelt und ausgeliefert werden.
2. **Schneller Workflow:** Alle repetitiven Aufgaben sollen hinter SBT Tasks versteckt werden, welche wiederum in IntelliJ Run Configurations gekapselt werden. Bei der Entwicklung soll sich alles so anfühlen, als hätte man nur auf den IDE-typischen Play-Button gedrückt. Außerdem soll der Build-Prozess nicht länger als ein paar Sekunden dauern.


Die von mir entwickelte Architektur erfüllt beide Ziele erfolgreich. Trotzdem sind wir noch lange nicht fertig: Neben den typischen Aufgaben wie Refactoring und Dokumentation ist hier die Anpassung der simplen Plugin-Architektur an die Domäne von Chat Overflow notwendig. Vor allem die Nutzbarkeit über einen längeren Zeitraum steht hier im Fokus. Damit verschiedenen Versionen von Plugins und Framework nicht zum Absturz führen (und sogar nicht mal zur Laufzeit ein Problem darstellen), ist hier noch **viel zusätzliche Arbeit notwendig**.


Ich hoffe, dass ich in diesem Blog-Post eine gute Einführung in die Thematik und die Probleme einer Plugin-Architektur geben konnte. Unterm Strich bin ich selbst überrascht, wie **viel Komplexität** hier bei der eigentlichen Entwicklung abgenommen werden kann. Und ich war etwas überrascht, wie viel komplexer die ganze Thematik ist. Ich bin sehr darauf gespannt, dieses Projekt zum ersten Mal im Kontext von Chat Overflow laufen zu sehen!

Den Quellcode der Test-Implementierung der Plugin-Architektur gibt es bei mir auf GitHub (<https://github.com/sebinside/PluginFrameworkTest>)!

Teilen mit:

 E-Mail (<http://sebinside.de/2018/01/16/eine-neue-plugin-architektur-fuer-code-overflow/?share=email&nb=1>)

 Facebook 2 (<http://sebinside.de/2018/01/16/eine-neue-plugin-architektur-fuer-code-overflow/?share=facebook&nb=1>)

 Twitter (<http://sebinside.de/2018/01/16/eine-neue-plugin-architektur-fuer-code-overflow/?share=twitter&nb=1>)

Gefällt mir:

Lade ...

[code overflow](http://sebinside.de/tag/code-overflow/) (<http://sebinside.de/tag/code-overflow/>) [java](http://sebinside.de/tag/java/) (<http://sebinside.de/tag/java/>)

[multi project](http://sebinside.de/tag/multi-project/) (<http://sebinside.de/tag/multi-project/>)

[plugin architektur](http://sebinside.de/tag/plugin-architektur/) (<http://sebinside.de/tag/plugin-architektur/>) [sbt](http://sebinside.de/tag/sbt/) (<http://sebinside.de/tag/sbt/>)

[scala](http://sebinside.de/tag/scala/) (<http://sebinside.de/tag/scala/>)

4 Gedanken zu „Eine neue Plugin-Architektur für Code Overflow“

TSCHAEGGIE

16. Januar 2018 um 17:23 Uhr (<http://sebinside.de/2018/01/16/eine-neue-plugin-architektur-fuer-code-overflow/#comment-131>)

Wäre für deinen Anwendungsfall das OSGi-Framework nicht perfekt gewesen?
Modularisierung zur Laufzeit, Versionierung und eigene Classloader sind doch genau was ihr braucht.

Oder wolltest du selber mit den Java Classloadern Spaß haben? ^^

ANTWORTEN ([HTTP://SEBINSIDE.DE/2018/01/16/EINE-NEUE-PLUGIN-ARCHITEKTUR-FUER-CODE-OVERFLOW/?REPLYTOCOM=131#RESPOND](http://sebinside.de/2018/01/16/eine-neue-plugin-architektur-fuer-code-overflow/?replytocom=131#respond))

SEB ([HTTP://SEBINSIDE.DE](http://sebinside.de))

16. Januar 2018 um 18:00 Uhr (<http://sebinside.de/2018/01/16/eine-neue-plugin-architektur-fuer-code-overflow/#comment-132>)

Sehr guter Kommentar! Tatsächlich habe ich mir OSGi auch angeschaut, weil ich damit schon im Rahmen der Eclipse-Plugin-Entwicklung zu tun hatte. Allerdings hatte ich Respekt vor dem Aufsetzen einer eigenen OSGi-Umgebung, im Internet hatte ich nicht die motivierendsten Erfahrungsberichte gelesen. Außerdem dachte ich, es ist sicher eine gute Fingerübung, mal selbst so eine Architektur zu entwerfen. Wir haben hier glücklicherweise immer nur eine 1-n Beziehung aus Framework (inkl. Providern wie z.B. Twitch-Chat) und Plugins. Hätten wir als Anforderung, auch noch Provider zur Laufzeit zu laden und damit ein n-m-Beziehung, hätte ich definitiv direkt zu OSGi gegriffen.

NTWORTEN ([HTTP://SEBINSIDE.DE/2018/01/16/EINE-NEUE-PLUGIN-ARCHITEKTUR-FUER-CODE-OVERFLOW/?REPLYTOCOM=132#RESPOND](http://sebinside.de/2018/01/16/eine-neue-plugin-architektur-fuer-code-overflow/?replytocom=132#respond))

DAVID ([HTTP://GRAVATAR.COM/ULLMIE02](http://gravatar.com/ullmie02))

17. Januar 2018 um 14:33 Uhr (<http://sebinside.de/2018/01/16/eine-neue-plugin-architektur-fuer-code-overflow/#comment-133>)

Wie heißt das Farbschema vom Titelbild?

ANTWORTEN ([HTTP://SEBINSIDE.DE/2018/01/16/EINE-NEUE-PLUGIN-ARCHITEKTUR-FUER-CODE-OVERFLOW/?REPLYTOCOM=133#RESPOND](http://sebinside.de/2018/01/16/eine-neue-plugin-architektur-fuer-code-overflow/?replytocom=133#respond))

SEB (HTTP://SEBINSIDE.DE)

24. Januar 2018 um 14:39 Uhr (<http://sebinside.de/2018/01/16/eine-neue-plugin-architektur-fuer-code-overflow/#comment-135>)

Selbst erstellt, das Vorbild war das dunkle Thema von eclipse

NTWORTEN (<HTTP://SEBINSIDE.DE/2018/01/16/EINE-NEUE-PLUGIN-ARCHITEKTUR-FUER-CODE-OVERFLOW/?REPLYTOCOM=135#RESPOND>)

KOMMENTAR VERFASSEN

Gib hier deinen Kommentar ein ...

◀ [Let's Play Automatisierung \(http://sebinside.de/2017/06/26/lets-play-automatisierung/\)](http://sebinside.de/2017/06/26/lets-play-automatisierung/)

SOCIAL MEDIA

(htt (htt (htt (htt
p://t p://f p://y (htt
witt aceb outu p://g
er.c ook. be.c ithu
om/ com om/ b.co
skat /seb skat m/s
e70 insid e70 ebin
2) e) 2) side)

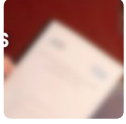
BLOG VIA E-MAIL ABONNIEREN

Gib Deine E-Mail-Adresse an, um diesen Blog zu abonnieren und Benachrichtigungen über neue Beiträge via E-Mail zu erhalten.

E-Mail-Adresse

ABONNIEREN

BELIEBTE BEITRÄGE



Rückblick auf den Bachelor (<http://sebinside.de/2017/04/13/rueckblick-auf-den-bachelor/>)

13 Apr , 2017



Programmierparadigmen
(<http://sebinside.de/2016/07/02/programmierparadigmen/>)

02 Jul , 2016



Webseiten kommen und gehen (<http://sebinside.de/2016/06/15/webseiten-kommen-und-gehen/>)

15 Jun , 2016

KATEGORIEN

📁 Allgemein (<http://sebinside.de/category/allgemein/>)

📁 Coding (<http://sebinside.de/category/coding/>)

📁 Minecraft (<http://sebinside.de/category/minecraft/>)

📁 Studium (<http://sebinside.de/category/studium/>)



NEUESTE KOMMENTARE

💬 Bowser bei Hallo. äh.. Welt! (<http://sebinside.de/2016/03/22/hallo-aeh-welt/#comment-138>)

💬 daeaeavaeae bei Rückblick auf den Bachelor (<http://sebinside.de/2017/04/13/rueckblick-auf-den-bachelor/#comment-137>)

💬 coernerbrot bei Programmierparadigmen
(<http://sebinside.de/2016/07/02/programmierparadigmen/#comment-136>)

💬 seb (<http://sebinside.de>) bei Eine neue Plugin-Architektur für Code Overflow
(<http://sebinside.de/2018/01/16/eine-neue-plugin-architektur-fuer-code-overflow/#comment-135>)

💬 Sven (<https://usacookie.de>) bei Rückblick auf den Bachelor
(<http://sebinside.de/2017/04/13/rueckblick-auf-den-bachelor/#comment-134>)

ARCHIV

Januar 2018 (<http://sebinside.de/2018/01/>)

Juni 2017 (<http://sebinside.de/2017/06/>)

Mai 2017 (<http://sebinside.de/2017/05/>)

April 2017 (<http://sebinside.de/2017/04/>)

Oktober 2016 (<http://sebinside.de/2016/10/>)

September 2016 (<http://sebinside.de/2016/09/>)

August 2016 (<http://sebinside.de/2016/08/>)

Juli 2016 (<http://sebinside.de/2016/07/>)

Juni 2016 (<http://sebinside.de/2016/06/>)

April 2016 (<http://sebinside.de/2016/04/>)


März 2016 (<http://sebinside.de/2016/03/>)

SOCIAL MEDIA



(htt (htt (htt (htt
 p://t p://f p://y (htt
 witt aceb outu p://g
 er.c ook. be.c ithu
 om/ com om/ b.co
 skat /seb skat m/s
 e70 insid e70 ebin
 2) e) 2) side)

FEED

 (<http://sebinside.de/feed/>) RSS - Beiträge (<http://sebinside.de/feed/>)

 (<http://sebinside.de/comments/feed/>) RSS - Kommentare (<http://sebinside.de/comments/feed/>)



Ich heiße Sebastian, studiere Informatik am KIT in Karlsruhe und betreibe den YouTube Kanal skate702 (<http://youtube.com/skate702>). Auf diesem Blog geht es um Technik, Software und Minecraft!

Impressum (<http://skate702.de/impressum/>) © 2016 skate702 – Theme von Colorlib (<http://colorlib.com/>) Powered by
 WordPress (<http://wordpress.org/>)