Architekturmanagement mit jQAssistant und Asciidoc

Welchen Nutzen hat eine aufwändig zu pflegende Architekurdokumentation, wenn deren Inhalt sich weit vom realen Geschehen im Code entfernt hat? Keinen, aber in der Realität ist dieser Zustand sehr oft anzutreffen. Im Folgenden soll an einem realen Beispiel ein leichtgewichtiger Ansatz beschrieben werden, der relevante Informationen direkt im Quellcode vorhält und nach Wunsch in jedem Build die Ist-Architektur mit dem gewünschten Soll-Zustand abgleicht.

Architekturentscheidungen sind sehr grundlegender Natur und haben langfristige Auswirkungen auf Struktur und Qualität eines Softwaresystems. Daher ist es notwendig, diese effizient im Entwicklungsteam zu kommunizieren. Dies geschieht üblicherweise in Form von Dokumentation. Die Existenz von Diagrammen oder verbalen Beschreibungen stellt aber noch nicht sicher, dass entsprechende Vorgaben auch wirklich eingehalten werden. Daher ist es sinnvoll, entsprechende Absicherungsmaßnahmen im Entwicklungsprozess zu etablieren. Üblicherweise kommen hierfür Code-Reviews zum Einsatz, deren Wirkmächtigkeit aber stark davon abhängt, wie groß der Umfang der Änderungen ist, wie gut Reviewer selbst über einzuhaltende Bedingungen informiert sind, wie gründlich sie arbeiteten usw. Ein weiteres Problem von Reviews ist die recht lange Feedbackschleife: Änderungen sind bereits implementiert, müssen aber ggf. im Nachhinein noch einmal korrigiert werden. Es entstehen möglicherweise hohe Aufwände, die bei bestehendem Termindruck auch schon einmal dafür sorgen können, dass die Dinge nicht mehr angefasst werden.

Aus diesen Gründen empfiehlt es sich, die Einhaltung von architekturbezogenen Regeln nach Möglichkeit automatisiert durch Werkzeuge im Build- und Integrationsprozess zu überprüfen. Ein aufgedeckter Fehler wird unmittelbar an den Entwickler zurückgemeldet, dieser kann zeitnah reagieren. Dazu muss er jedoch oft den Kontext des Problems verstehen, d. h. er benötigt Informationen über dahinterstehende Architekturentscheidungen. Diese findet er üblicherweise in der Dokumentation. Es liegt also nahe, die beiden Aspekte miteinander zu verknüpfen: die Dokumentation und die Ausführung von Architekturregeln. Die Kombination aus jQAssistant [1] und Asciidoc adressiert genau diesen Aspekt und soll im Folgenden an einem Open-Source-Projekt näher erläutert werden.

Fallbeispiel: eXtendedObjects

jQAssistant integriert sich in den Build-Prozess von Projekten, liest strukturelle Informationen (Maven-Module, Packages, Klassen, etc.) ein und speichert diese als Graph in einer Neo4j-Datenbank. Zur Realisierung des letztgenannten Aspekts kommt ein Object-Graph-Mapper (OGM) zum Einsatz: eXtendedObjects [2], im Folgenden kurz "XO" genannt. Seine Funktionalität ähnelt der von JPA (Mappings und Abfragen), ist aber stärker auf Graphen sowie deren wesentlich höhere Flexibilität in Bezug auf Datenmodellierung ausgerichtet. XO zerfällt in folgende grundlegende Struktur:

Framework:

- Application Programming Interface (API): Client-API, welches von Anwendungen benutzt werden kann.
- Service Provider Interface (SPI): Definiert die Schnittstelle, welche ein spezifischer Datastore (z. B. Neo4j) umsetzen muss.
- Implementierung: Umsetzung des API, dabei wird das SPI genutzt. Der konkrete Datastore ist konfigurierbar.

Neo4j-Datastore:

- API: datastore-spezifisches API, enthält z. B. Definitionen von Annotationen
- SPI: Gemeinsam genutzte Abstraktionen der Neo4j-Datastore-Implementierungen
- Embedded: Implementierung eines Neo4j-Datastores, der eine eingebettete Datenbank (lokales Verzeichnis oder In-Memory) benutzt
- Remote: Implementierung eines Neo4j-Datastores, der per Remote-Protokoll (Bolt) mit einer entfernten Neo4j-Instanz kommuniziert

Die Beschreibung der Elemente stellt eine Komponentensicht der Architekturdefinition dar. Es fehlen aber zwei Dinge: die erlaubten Abhängigkeiten und die Abbildung auf die entsprechenden Strukturelemente im Code.

Abbildung 1: Screenshot der gerenderten Tabelle

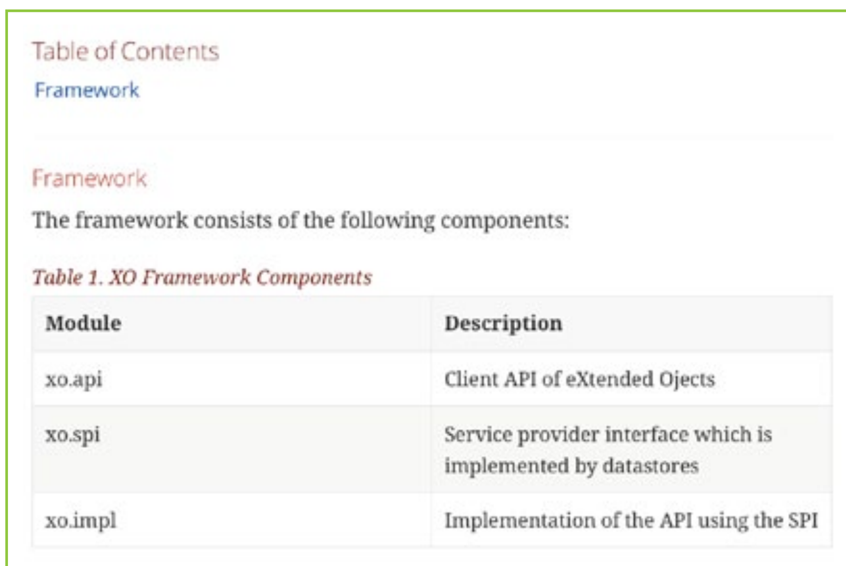


Table of Contents

Framework

Framework

The framework consists of the following components:

Table 1. XO Framework Components

Module	Description
xo.api	Client API of eXtended Ojects
xo.spi	Service provider interface which is implemented by datastores
xo.impl	Implementation of the API using the SPI

Architektur als Dokumentation im Code

Der Zweck von XO besteht darin, als Bibliothek in andere Anwendungen eingebettet zu werden. Dabei sollen jeweils nur die Teile verwendet werden, die wirklich benötigt werden. Daraus ergibt sich schnell die Konsequenz, dass XO nicht monolithisch ausgeliefert werden kann, sondern in verschiedene JAR-Artefakte zerfällt, die den oben beschriebenen Komponenten entsprechen. Die Architekturdefinition findet also auf der Ebene von Artefakten statt, die im konkreten Fall durch entsprechende Maven-Module erzeugt werden. Alternativ käme die Abbildung von Komponenten auf Packages (z. B. web, service, model) oder Klassentypen (z. B. Controller, Service, Repository) in Frage, das ergibt im Kontext von XO aber keinen Sinn.

Die verbale Beschreibung dieser Konzepte lässt sich gut in Asciidoc, einer handlichen und mächtigen Markup-Language, formulieren und ist im Fall von XO in der Wurzel des Projektes unter "jqassistant/architecture.adoc" abgelegt. Diese und andere Dateien im gleichen Ordner werden im Build-Prozess nach HTML und PDF transformiert. Das folgende Snippet umfasst die oben genannten Strukturbeschreibungen und wird als Tabelle (siehe Abbildung 1) gerendert:

```
==== Framework
The framework consists of the following
components:
[options="header"]
.XO Framework Components
|====
| Module | Description
| xo.api | Client API of eXtended Ojects
| xo.spi | Service provider interface which
is implemented by datastores
| xo.impl | Implementation of the API
using the SPI
|====
```

Die Pflege der Dokumentation im Source-Code-Repository des Projektes hat einen entscheidenden Vorteil: notwendige Änderungen der Architektur im Code können bei Bedarf unmittelbar in die Dokumentation einfließen und umgekehrt. Die parallele Entwicklung in Branches stellt dabei kein Hindernis dar, zumal sich Asciidoc-Dateien ohne Probleme mergen lassen.

Architekturdefinition

Im nächsten Schritt werden die Abbildungen der Komponenten auf Maven-Module vorgenommen. Dies erfolgt nicht verbal, sondern über ein eingebettetes Source-Code-Fragment, welches eine entsprechende Cypher-Query [3] beinhaltet:

```
[[architecture:Framework]]
[source,cypher,role=concept,requiresConcept
s="maven:MainArtifact",severity=critical]
.Adds the label 'Framework' and 'Compo-
nent' to the framework artifacts.
----
MATCH
   (frameworkComponent:Main:Artifact)
WHERE
  frameworkComponent.name in [
    "xo.api",
    "xo.spi",
    "xo.impl"
  ]
SET
  frameworkComponent:Framework:Component
RETURN
  frameworkComponent
----
```

Es handelt sich um eine Regel, die durch jQAssistant ausgeführt werden kann. Die Kopfdaten umfassen eine eindeutige, referenzierbare Id (`architecture:Framework`) sowie Meta-Informationen wie die Rolle (`role=concept`) und Abhängigkeiten zu anderen Regeln (`requiresConcepts="…"`).

Ein Konzept dient der Anreicherung des Graphen um weitere Informationen. Im konkreten Fall werden allen Artefakten, welche durch das referenzierte Konzept `maven:MainArtifact` bereits mit dem Label `Main` versehen wurden, und deren Name in der gegebenen Liste enthalten ist, zwei weitere Labels `Framework` und `Component` hinzugefügt. Diese Begriffe stammen aus der XO-eigenen Architekturwelt.

Die Regel ist darüber hinaus mit `severity=critical` versehen. Bei Umbenennung einer Komponente liefert die Cypher-Query keine Ergebnisse, dies wird von jQAssistant als Verletzung interpretiert und kann je nach Konfiguration zum Abbruch des Builds führen. In diesem Fall ist das sehr sinnvoll: die Codestrukturen stimmen dann nicht mehr mit der Architekturdefinition überein.

Damit lässt sich eine weitere Regel definieren, welche die erlaubten Abhängigkeiten zwischen den Framework-Komponenten definiert:

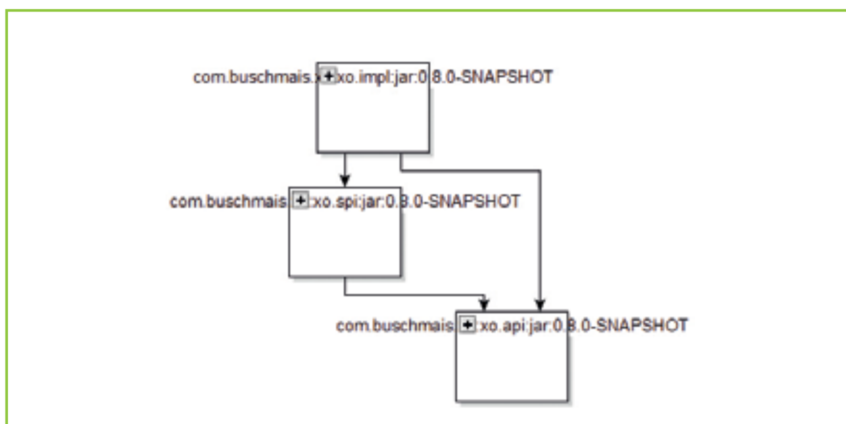
```
[[architecture:FrameworkDependencyDefinition]]
[source,cypher,role=concept,requiresConcepts=
"architecture:Framework",reportType=graphml]
.Defines the allowed dependencies between
the framework artifacts.
----
MATCH
  (api:Framework:Component{name:"xo.api"}),
  (spi:Framework:Component{name:"xo.spi"}),
  (impl:Framework:Component{name:"xo.impl"})
CREATE
  (spi)-[d1:DEFINES_DEPENDENCY]->(api),
  (impl)-[d2:DEFINES_DEPENDENCY]->(api),
  (impl)-[d3:DEFINES_DEPENDENCY]->(spi)
RETURN
  *
----
```

jQAssistant stellt sicher, dass zunächst `maven:MainArtifact` sowie `architecture:Framework` ausgeführt werden. Das Konzept `architecture:FrameworkDependencyDefinition` ermittelt nun darauf aufbauend die einzelnen Framework-Komponenten jeweils anhand ihres Namens und erzeugt erlaubte Abhängigkeitsbeziehungen mit dem Typ `DEFINES_DEPENDENCY`.

In den Metadaten dieses Konzepts ist darüber hinaus der Report-Typ `graphml` spezifizert (`reportType="graphml"`). Aufgrund dessen wird das Ergebnis der RETURN-Klausel als Datei `architecture_FrameworkDependencyDefinition.graphml` ausgegeben, welche sich mit yEd [4] öffnen lässt. Nach Anwendung eines hierarchischen Layouts (Layout --> Hierarchical, Alt-Shift-H) kann die in Abbildung 2 sichtbare Soll-Architektur visuell überprüft werden.

Es könnte nun der Einwand folgen, dass Maven bereits ein Abhängigkeitsmanagement zur Verfügung stellt und auf diesem Wege eine unnötige Redundanz entsteht. Dagegen sprechen zwei Argumente: Erstens ist es für Entwickler durch moderne IDEs relativ einfach, neue Abhängigkeiten in `pom.xml`-Dateien einzupflegen, die explizite Definition in der Dokumentation stellt hier ein Sicherungssystem dar. Zum Zweiten lässt sich über Maven nur schwer kontrollieren, inwiefern der Zugriff auf transitive Abhängigkeiten erlaubt sein soll oder nicht. In der beschriebenen Framework-Definition wird `xo.api` indirekt über `xo.spi` auch für `xo.impl` zugänglich, der direkte Zugriff ist hier auch explizit erlaubt. Wäre das nicht der Fall, würde die entsprechende Beziehungsdefinition in jQAssistant einfach weggelassen. Im Falle der Kontrolle via Maven müsste dies über recht komplexe und schwer erklärbare Exclusions oder Scopes (provided) realisiert werden.

Abbildung 2: Soll-Architektur



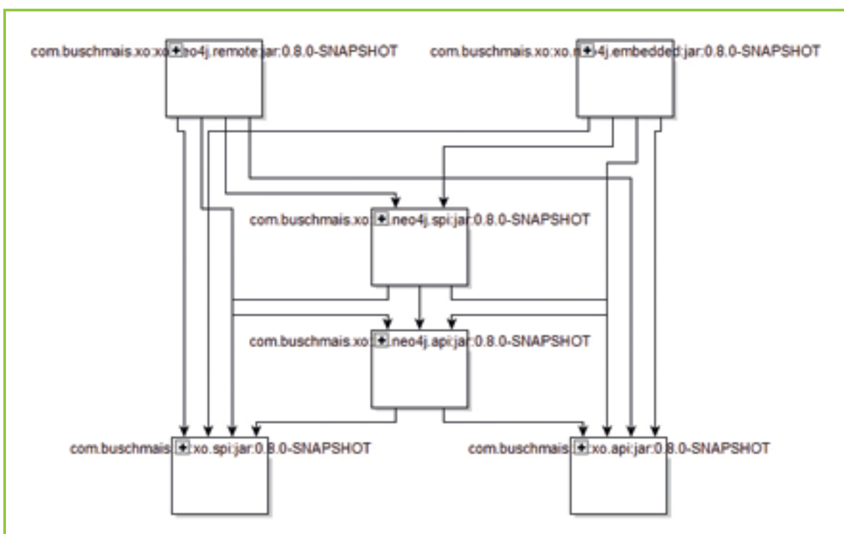
Ein wichtiger Teil von XO ist der Neo4j-Datastore und auch für ihn sollen im Folgenden Architekturkonzepte definiert werden. Zunächst erfolgt wiederum die Markierung der entsprechenden Komponenten:

```
[[architecture:Neo4jDatastore]]
[source,cypher,role=concept,requiresConcept
s="maven:MainArtifact",severity=critical]
.Adds the labels 'Datastore' and 'Compo-
nent' to the Neo4j datastore components.
----
MATCH
   (neo4jComponent:Main:Artifact)
WHERE
  neo4jComponent.name in [
    "xo.neo4j.api",
    "xo.neo4j.spi",
    "xo.neo4j.embedded",
    "xo.neo4j.remote"
  ]
SET
  neo4jComponent:Datastore:Component
RETURN
  neo4jComponent
----
```

Listing 1:
Soll-Architektur Neo4j

```
[[architecture:Neo4jDatastoreDependencyDefinition]]
[source,cypher,role=concept,requiresConcepts="architecture:Framework,
architecture:Neo4jDatastore",reportType=graphml]
.Defines the allowed dependencies of the Neo4j datastore components
----
MATCH
  (api:Framework:Component{name:"xo.api"}),
  (spi:Framework:Component{name:"xo.spi"}),
  (neo4jApi:Datastore:Component{name:"xo.neo4j.api"}),
  (neo4jSpi:Datastore:Component{name:"xo.neo4j.spi"}),
  (neo4jEmbedded:Datastore:Component{name:"xo.neo4j.embedded"}),
  (neo4jRemote:Datastore:Component{name:"xo.neo4j.remote"})
CREATE
  (neo4jApi)-[d1:DEFINES_DEPENDENCY]->(api),
  (neo4jApi)-[d2:DEFINES_DEPENDENCY]->(spi),
  (neo4jSpi)-[d3:DEFINES_DEPENDENCY]->(api),
  (neo4jSpi)-[d4:DEFINES_DEPENDENCY]->(spi),
  (neo4jSpi)-[d5:DEFINES_DEPENDENCY]->(neo4jApi),
  (neo4jEmbedded)-[d6:DEFINES_DEPENDENCY]->(api),
  (neo4jEmbedded)-[d7:DEFINES_DEPENDENCY]->(spi),
  (neo4jEmbedded)-[d8:DEFINES_DEPENDENCY]->(neo4jApi),
  (neo4jEmbedded)-[d9:DEFINES_DEPENDENCY]->(neo4jSpi),
  (neo4jRemote)-[d10:DEFINES_DEPENDENCY]->(api),
  (neo4jRemote)-[d11:DEFINES_DEPENDENCY]->(spi),
  (neo4jRemote)-[d12:DEFINES_DEPENDENCY]->(neo4jApi),
  (neo4jRemote)-[d13:DEFINES_DEPENDENCY]->(neo4jSpi)
RETURN *
----
```

Abbildung 3:
Soll-Architektur Neo4j



Anstelle des Labels `Framework` wird nun jedoch `Datastore` verwendet. Darauf basierend erfolgt die Definition erlaubter Abhängigkeiten (Listing 1, Abbildung 3).

Die CREATE-Klausel zum Erzeugen der Beziehungen ist recht umfangreich. Allerdings fällt bei genauerer Betrachtung auf, dass die Definitionen für `xo.neo4j.embdedded` (neo4jEmbedded) und `xo.neo4j.remote` (neo4jRemote) eigentlich identisch sind. Es ließe sich also im Bedarfsfall ein entsprechendes generisches Konzept extrahieren. Dies wird im Beispiel zwar nicht durchgeführt, illustriert aber die sehr mächtige Möglichkeit, Architekturdefinitionen anhand von Mustern zu erstellen.

Architekturverifikation

Das Datenmodell beinhaltet nun Komponenten sowohl aus dem Framework als auch dem Neo4j-Datastore jeweils in der Form von Knoten mit dem Label `Component` und deren definierten Beziehungen vom Typ `DEFINES_DEPENDENCY`. Das folgende Konzept erzeugt einen GraphML-Report über die vollständige Architekturdefinition:

```
[[architecture:ComponentDependencyDefinition]]
[source,cypher,role=concept,requiresConcept
s="architecture:FrameworkDependencyDefiniti
on,architecture:Neo4jDatastoreDependencyDef
inition",reportType=graphml]
.Returns all defined components and their
defined dependencies
----
MATCH
  (c:Component)
OPTIONAL MATCH
  (c)-[d:DEFINES_DEPENDENCY]->(:Component)
RETURN
  *
----
```

Das Konzept besitzt eine weitere interessante Eigenschaft: aufgrund dessen, dass es via `requiresConcepts` von allen die Architektur definierenden Konzepten abhängt, eignet es sich selbst hervorragend, um als Voraussetzung für weitere Regeln verwendet zu werden, welche die vollständigen Komponentendefinitionen benötigen. Eine solche ist der generischer `Constraint` (role=constraint), der einen Abgleich von Soll- und Ist-Stand vornimmt:

```
[[architecture:ComponentDependencyViolation]]
[source,cypher,role=constraint,requiresConce
pts="architecture:ComponentDependencyDefini
tion",severity=critical]
.There must be no dependencies between
components that are not explicitly defined
----
MATCH
  (c1:Component)-[:CONTAINS]->(t1:Type),
  (c2:Component)-[:CONTAINS]->(t2:Type),
  (t1)-[:DEPENDS_ON]->(t2)
WHERE
  c1 <> c2
  and not (c1)-[:DEFINES_DEPENDENCY]->(c2)
RETURN
  c1 as Component, t1 as Type,
  c2 as InvalidComponent,
  collect(t2) as InvalidDependencies
----
```

Die Query ist nahezu selbstbeschreibend: es werden Typen in verschiedenen Komponenten ermittelt, die voneinander abhängig sind, für die auf Komponentenebene aber keine Abhängigkeit definiert ist, d. h. keine Beziehung vom Typ `DEFINES_DEPENDENCY` existiert. Wird also ein Ergebnis zurückgeliefert, handelt es sich um eine Architekturverletzung, welche von jQAssistant berichtet wird und zum Abbruch des Builds genutzt werden kann.

Doch auch die andere Richtung kann von Interesse sein: gibt es Abhängigkeitsdefinitionen in der Architektur, welche im Code nicht (mehr) existieren? Der entsprechende Constraint sieht folgendermaßen aus:

```
[[architecture:UnusedComponentDependencyDe
finition]]
[source,cypher,role=constraint,requiresConce
pts="architecture:ComponentDependencyDefini
tion",severity=major]
.There must be no unused dependency defi-
nitions between components
----
MATCH
  (c1:Component),
  (c2:Component),
  (c1)-[:DEFINES_DEPENDENCY]->(c2)
WHERE
  c1 <> c2
  and not
    (c1)-[:CONTAINS]->(:Type)
    -[:DEPENDS_ON]->
    (:Type)<-[:CONTAINS]-(c2)
RETURN
  c1,c2
----
```

Zu guter Letzt muss noch geklärt werden, wie diese Regeln auszuführen sind. Dazu können Gruppen definiert werden, welche die Constraints direkt und die damit verbundenen Konzepte indirekt refenzieren:

```
[[architecture:Default]]
[role=group,includesConstraints="architectu
re:ComponentDependencyViolation,architectur
e:UnusedComponentDependencyDefinition"]
```

Die Gruppe `architecture:Default` selbst wird in der Datei `index.adoc` durch die Gruppe `default` referenziert. Letztere wird durch jQAssistant automatisch im Build ausgeführt, zumindest sofern es in der Konfiguration des Maven-Plugins nicht anders angegeben ist.

Zusammenfasssung

Die Architektur einer Applikation direkt im Source-Code-Repository zu dokumentieren, hat den großen Vorteil, dass einem Auseinanderdriften von Wunsch und Realität von vornherein effektiv vorgebeugt werden kann, d. h. notwendige Änderungen können mit relativ geringem Aufwand durchgeführt werden. Asciidoc eignet sich sehr gut dafür. Unter Zuhilfenahme von jQAssistant können in der Dokumentation eingebettete Regeln sogar unmittelbar ausgeführt werden. Diese umfassen zunächst Konzepte, die als Abfragen in Cypher formuliert sind und zur Anreicherung des Graphen dienen. Dazu gehört die Definition einer Soll-Architektur, welche einerseits die Abbildung von Komponenten auf Codestrukturen, andererseits die Definition erlaubter Abhängigkeitsbeziehungen umfasst. Die Konzepte selbst sind äußerst flexibel, d. h. die Abbildung von Komponenten kann über Muster erfolgen und damit Coding- bzw. Namenskonventionen erzwingen. Durch die geeignete Wahl von Abstraktionen - im Beispiel wurde `Component` verwendet - ist es dann möglich, relativ einfache und generische Constraints zu formulieren, welche den Ist-Stand des Codes mit der Soll-Architektur abgleichen.

■ Dirk Mahler

Links

[1] http://jqassistant.org
[2] https://github.com/buschmais/eXtendedObjects
[3] https://neo4j.com/developer/cypher-query-language/
[4] https://www.yworks.com/products/yed

Dirk Mahler ist als Senior Consultant bei der buschmais GbR auf dem Gebiet der Java-Enterprise-Technologien tätig. In seiner täglichen Arbeit setzt er sich mit Themen rund um Software-Architektur auseinander.

Infos und Kontakt zu jQAssistant

Beratung
www.jqassistant.de

Blog
www.jqassistant.org

GitHub
https://github.com/buschmais/jqassistant

Twitter
www.twitter.com/jqassistant

Wenden Sie sich bei Fragen bitte an:
Herrn Dirk Mahler (buschmais GbR)
Tel. +49 351 3209230
beratung@jqassistant.com

Impressum | **Herausgeber** buschmais GbR, Leipziger Straße 93, 01127 Dresden | **Telefon** +49 351 3209230 | **Telefax** +49 351 32092329 | **E-Mail** info@buschmais.com | **Internet** www.buschmais.de |

Copyright © 2017 buschmais GbR Alle Rechte vorbehalten. "buschmais" ist eine eingetragene Marke der buschmais GbR. Ihre Verwendung im Geschäftsverkehr ist nur bei ausdrücklicher, schriftlicher Erlaubnis der buschmais GbR statthaft.