Ausgabe Frühjahr 2020

www.gedoplan.de

GEDOPLAN aktuell

In dieser Ausgabe:

Liebe Leserin, lieber Leser,

in dieser Ausgabe schließen wir unsere Serie zu Domain-Driven Design mit dem letzten Artikel ab. Jens Seekamp, GEDOPLAN GmbH, beantwortet darin die Frage, wie die Gesamtstruktur eines Anwendungssystems aussehen soll: wie "Fingerfood" - abwechslungsreich, kleinteilig, bestehend aus verschiedenartigen, mundgerechten Portionen.

Angular bleibt ein zentrales Thema in der Entwicklung von Webanwendungen. Unser Experte für dieses Thema, Dominik Mathmann, stellt Ihnen Angular Materials Theming vor. Aber nicht nur als Bibliothek für einbindbare Komponenten, sondern auch als Werkzeug, um global auf das Aussehen der Komponenten einzuwirken.

Dirk Weil, Geschäftsführer GEDOPLAN GmbH, tritt gegen das Vorurteil an, JEE-Anwendungen seien immer schwergewichtig. Mit Quarkus geht das auch anders: leichtgewichtig, mit Spaß und in der Entwicklungsumgebung.

Krisenzeiten erfordern neue Ideen. GEDOPLAN IT Training hat sein Angebot auf die Anforderungen in der Coronakrise angepasst: Einen Großteil unserer Seminare bieten wir auch remote an!
Mehr dazu im Heft!

Viel Spaß beim Lesen!

Ulrich Hake | ulrich.hake@gedoplan.de

Ulrich Hake

Termine

Expertenkreis Java

Thema: **Microprofile-Anwendungen mit Quarkus – reloaded**
Ort: GEDOPLAN GmbH, Stieghorster Str. 60, 33605 Bielefeld
Termin: Donnerstag, 27.02.2020| 18:00 - 19:30 Uhr
Referent: Dirk Weil, GEDOPLAN GmbH

Thema: **Domain-Driven Design (DDD): Implementierung einer universellen Turing-Maschine**
Ort: GEDOPLAN GmbH, Stieghorster Str. 60, 33605 Bielefeld
Termin: Donnerstag, 16.01.2020 | 18:00 - 19:30 Uhr
Referent: Jens Seekamp, GEDOPLAN GmbH

Vorträge

Thema: **Mit MicroProfile und Quarkus in die Wolken**
Ort: Treffpunkt Semicolon, Köln
Termin: Dienstag, 26.05.2020 | 18:00 - 19:00 Uhr
Referent: Dirk Weil, GEDOPLAN GmbH

Thema: **Domain-Driven Design (DDD): Implementierung einer universellen Turing-Maschine**
Ort: "Goldschmiede", Köln
Termin: Freitag, 29.05.2020 | 17:30 Uhr
Referent: Jens Seekamp, GEDOPLAN GmbH

Thema: **JEE und Micro – kein Widerspruch!**
Ort: Developer Week '20, Nürnberg
Termin: Donnerstag, 02.07.2020 | 10:30 - 11:30 Uhr
Referent: Jens Seekamp, GEDOPLAN GmbH

Software-Architektur (Teil 4): Software-Komponenten - Domain-Driven Design als "Fingerfood"

Im abschließenden Teil unserer Artikel-Reihe zur Software-Architektur wollen wir uns mit der Frage beschäftigen, welche Gesamtstruktur unser Anwendungs-System haben soll. Aber was hat diese Gesamtstruktur nun mit Fingerfood - also dem Thema Essen - zu tun? Stellen wir uns einen großen Speiseteller mit einem opulenten Menü vor, welches uns später vielleicht schwer im Magen liegt. Als Alternative stellen wir uns ein abwechslungsreiches, kleinteiliges Fingerfood vor, bestehend aus verschiedenartigen, mundgerechten Portionen.

Von Jens Seekamp

Der böse Monolith

Der große, aufwändig herzustellende und manchmal auch schwer verdauliche Menüteller entspricht einer monolithischen Software-Architektur. Im Hinblick auf die von uns benutzte Java Enterprise Edition (Java EE) sprechen wir von einem **Monolithen**, wenn die Software-Architektur des Anwendungs-Systems folgende Eigenschaften aufweist:

- Alle Klassen der Anwendung liegen in einem gemeinsamen Software-Artefakt. Auf der konzeptionellen Ebene sind die Klassen in einer durchdachten Paketstruktur angeordnet, typischerweise in einer Schichten-Architektur mit z. B. den vier Ebenen (1) Präsentation, (2) Geschäftsprozesse, (3) fachliche Logik und (4) Datenzugriff.
- Der gesamten Anwendung liegt ein zentrales, umfassendes Datenmodell zugrunde - oft als Unternehmens-Datenmodell bezeichnet. Deswegen agiert die Anwendung auch auf einer zentralen (relationalen) Datenbank.
- Der Build-Prozess (Maven oder Gradle) erzeugt eine Deployment Unit für unser Anwendungs-System, also ein Web-Archiv (war-Datei) oder ganz klassisch ein Enterprise-Archiv (ear-Datei).
- Diese Deployment Unit wird auf einem Application Server (z. B. Red Hat WildFly) ausgeführt. Wegen der anwendungs-spezifischen Konfiguration des Application Servers und damit im Betrieb sich die verschiedenen Java-EE-Anwendungen "nicht in die Quere kommen", wird unser Anwendungs-System allein auf dem Application Server betrieben.

Wer von uns Java-EE-Entwicklern hätte nicht schon erfolgreich an einem solchermaßen aufgebauten Anwendungs-System mitgewirkt? Aber leider nagt der Zahn der Zeit an unserer schön konzipierten Software:

- Neue, bislang nicht bedachte Anforderungen müssen - oft leider unter Zeitdruck - umgesetzt werden. Dabei passieren leicht Verstöße gegen die Schichten-Architektur (der "Klassiker" ist die Geschäftslogik in der GUI-Klasse) oder eine neue Funktionalität passt nirgendwo so richtig hin (also bauen wir noch einen weiteren "Balkon" an eines der Stockwerke unseres Software-Gebäudes).
- Leider hat unser Projekt auch keine ausreichenden Mechanismen zur Überwachung und damit Sicherstellung der konzeptionellen Software-Architektur: Code-Reviews sind aufgrund des Programmumfangs nur partiell möglich, zum permanenten Pair-Programming fehlt die Zeit und die statische Code-Analyse überwacht zwar einfache Programmier-Richtlinien, aber nicht die eigentlich wichtigeren Architektur-Regeln.
- Auch auf der Ebene des Datenmodells und damit ebenfalls im Datenbank-Schema kann sich Wildwuchs breit machen. Eigentlich schon überfrachtete Entitäten bzw. Tabellen werden mit noch mehr Attributen bzw. Spalten erweitert, oder noch mehr Beziehungen zwischen den Fachobjekten bzw. Fremdschlüssel in den Tabellen werden hinzugefügt.

Nach einigen Jahren haben wir dann den **Big Ball of Mud** ("Matschhaufen") bzw. eine Software wie ein Wollknäuel. Es ist nicht mehr mit vernünftigem Aufwand nachvollziehbar, welche Abhängigkeiten - sprich Verwendungs- oder Aufruf-Beziehungen - zwischen den sehr vielen Java-Klassen bestehen. Die ehedem wohldurchdachte Software-Architektur ist degeneriert, weil Architektur-Regeln und Best Practices wie beispielsweise Zyklen-Freiheit an vielen Stellen verletzt werden. Refactorings oder gar ein Redesign von System-Teilen sind extrem schwierig. Unter dem Strich haben wir ein schwer verständliches und kaum wartbares Software-System.

"Wer hat's erfunden?"

Wir bleiben unserer Essens-Metapher treu, wollen aber jetzt nicht den Schweizern die Schuld in die Schuhe schieben.
Als grundlegende Ursache von vielen Problemen einer monolithischen Software-Architektur erscheint das so harmlose Java-Schlüsselwort public. Eine öffentliche Java-Klasse ist nun einmal von überall her in einer Deployment Unit aufrufbar. Und Java-Pakete definieren lediglich eine Zuordnung von Klassen zu Namensräumen, aber keine brauchbaren Zugriffs-Beschränkungen. Somit muss leider konstatiert werden, dass die "suboptimalen" Sprachmittel von Java (zumindest bis Version 8 einschließlich) als die technische Ursache für die möglichen Nachteile einer monolithischen Software-Architektur gelten müssen.

Divide and Conquer

Ein wichtiges Grundprinzip der Software-Technik lautet **Divide and Conquer** ("teile und herrsche"), d. h. Zerlegung eines komplexen Software-Systems in beherrschbare Teile. Also Fingerfood anstelle opulenter Menü-Teller.
Zwar ist auch eine Schichten-Architektur - manifestiert durch eine entsprechende Java-Paket-Struktur - innerhalb eines Monolithen eine mögliche Umsetzung dieses Grundprinzips. Um das Problem an der Wurzel zu packen, ist jedoch die Definition von Software-Komponenten gemäß dem Ansatz **Component-Based Development** (CBD) notwendig. In unserer Referenz-Implementierung Flight Information System (FIS) sprechen wir daher von einer **Business-Component** (BC) im Sinne einer Software-Komponente mit folgenden Eigenschaften:

- hat einen Namen
- hat eine starke Kohäsion
- ist in sich selbst abgeschlossen (self-contained)
- hat explizit definierte Export- und Import-Schnittstellen
- hat eine nach außen versteckte Implementierung (body)
- ist eine ausführbare Software-Einheit
- hat eine eindeutige Versionsnummer

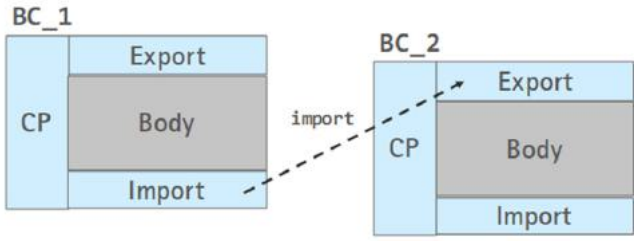
Aber was soll nun der Inhalt einer solchen Business-Component sein? Auch hier liefert abermals das Domain-Driven Design mit dem Konzept des Bounded Context (BC) eine passende Antwort. Der Ansatz besteht in einer fachlich getriebenen Strukturierung des Domain Model in mehrere, kleinere Bounded Contexts mit den folgenden Eigenschaften:

- umfasst jeweils ein abgegrenztes Geschäftsfeld der Domain
- ist eine wohldefinierte Partition bzw. Teilbereich des Domain Models
- gruppiert jeweils alle logisch eng zusammengehörigen Daten, Verarbeitungsprozesse, Ereignisse, Regeln usw.

Im Endergebnis soll jeder Bounded-Context genau einen bestimmten Teilbereich des Domain Models einkapseln, und umgekehrt sollen die verschiedenen Bounded-Contexts voneinander abgegrenzt und möglichst gut entkoppelt sein. Damit eine solche Zerlegung in Bounded Contexts gelingt, ist eine Entkopplung der Aufruf-Beziehungen auf Anwendungsebene und der Referenz-Beziehungen auf Datenebene notwendig.
Ein Blick auf den fachlichen Inhalt und die interne Strukturierung einer Business-Component offenbart somit die Verknüpfung zu den bereits besprochenen Lösungskonzepten:

- Jede Business-Component repräsentiert inhaltlich gemäß Domain-Driven Design genau einen Bounded Context des Anwendungs-Systems
  .
- Der Java-Code innerhalb jeder Business-Component ist gemäß Clean Architecture strukturiert.

Der entscheidende Unterschied zu einer reinen Paket-Aufteilung ist nun die Festlegung, dass die Import-Schnittstelle einer benutzenden Business-Component nur die Export-Schnittstelle einer benutzten Business-Component referenzieren darf. Es dürfen also nur diejenigen "Leistungen" importiert werden, die anderswo explizit exportiert werden, wie die folgende Grafik veranschaulicht:



Um eine Degeneration der Komponenten-Zerlegung dauerhaft zu unterbinden, muss die soeben formulierte Architektur-Regel auf technischer Ebene sichergestellt werden (und nicht nur als "Soll-Vorschrift" für die Programmier-Disziplin formuliert sein). Für die software-technische Umsetzung des Lösungskonzeptes Business-Component bieten sich daher durchaus unterschiedliche Möglichkeiten an.

Architektur-Ansatz 1: Der nicht mehr ganz so böse Monolith

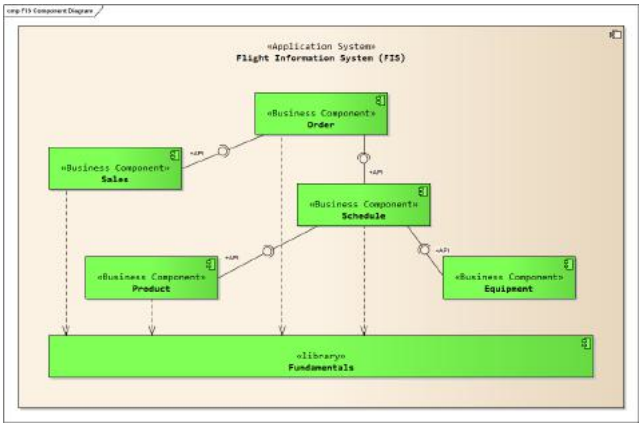
Ab Version 9 bietet Java selbst mit dem Java Platform Module System (JPMS) ein Modul-Konzept. Im Module-Descriptor module-info.java können die drei nach außen sichtbaren Sektionen **Export**, **Import** und **Common Parameters (CP)** (Schnittmenge von Export und Import) eines Moduls definiert werden, wie das folgende Beispiel demonstriert:

```
module de.gedoplan.fis.schedule {

  exports de.gedoplan.fis.schedule;                    Export

  requires de.gedoplan.fis.product;
  requires de.gedoplan.fis.equipment;                  Import

  requires transitive de.gedoplan.fis.fundamentals;
}                                                      Common Parameters
```

Über eine geeignete Paketstruktur wird die vierte, nach außen versteckte Modul-Sektion **Body** von den sichtbaren Schnittstellen so separiert, dass der Java-Compiler den Aufruf von Klassen verhindert, die im Rumpf des Moduls versteckt sind. In diesem Sinne sind JPMS-Module in den wesentlichen Punkten deckungsgleich mit dem Konzept der Business-Component; lediglich die Kriterien Ausführbarkeit und Versionierung entfallen, da der modularisierte Monolith als Ganzes ausgeführt und versioniert wird.
Der große Vorteil eines solchermaßen **modularisierten Monolithen** ist die Tatsache, dass die Kommunikation zwischen den Modulen über einfache Methodenaufrufe abgewickelt wird. Auf Ebene der Datenbank bleiben uns die referentielle Integritätsprüfung und Transaktions-Sicherheit erhalten. Und schließlich profitieren wir im Entwicklungsprozess, beim Deployment und im Betrieb von der Tatsache, dass wir nur eine Deployment Unit verwalten müssen.

Architektur-Ansatz 2: verteilte Java-EE-Anwendung

Ein anderer Weg wird in einem unserer großen Kunden-Projekte beschritten. Da die unternehmensweite Kern-Anwendung eine beachtliche Größe hat (ca. 1200 Datenbank-Tabellen, ca. 140 Java-Entwickler, geschätzt > 10 Millionen LOC) und die Anwendungs-Entwicklung in separaten, auch räumlich verteilten Teams stattfindet, wird ein verteiltes System von separaten Java Enterprise Anwendungen realisiert.
Die Business-Components werden gemäß dem DDD-Konzept Bounded Context als vertikale, fachliche Schnitte des Anwendungs-Systems umgesetzt. Die Separierung zwischen sichtbarer Schnittstelle und verstecktem Rumpf wird durch zwei Maven-Module Api und Impl erreicht. Die Impl-Projekte werden mit Maven als Web-Archive gebaut und als separate Java-EE-Anwendungen betrieben. Damit ergibt sich die folgende Anwendungs-Architektur für unsere Referenz-Implementierung *Flight Information System (FIS)*:



Die Business-Components formen zusammen mit ihren Aufruf- und Verwendungs-Beziehungen einen gerichteten Graphen. Aus verschiedenen logischen und technischen Gründen muss dieser Graph frei von Zyklen sein. Diese Forderung nach einem **directed acyclic graph (DAG)** ist eine der großen Herausforderungen bei der Komponenten-Zerlegung von Anwendungs-Systemen, insbesondere wenn es sich um ein Migrations- oder Redesign-Projekt für ein ehemals monolithisches System handelt.
Die Kommunikation zwischen den Business-Components muss nun mit den von Java EE zur Verfügung gestellten Remoting-Technologien abgewickelt werden. Die hierfür definierten, technisch orientierten Building-Blocks seien der Vollständigkeit halber kurz skizziert:

(1) Mit dem Building Block **Domain-Gateway** wird die Kommunikation zwischen den verteilt ausgeführten Business-Components ermöglicht. Über ein Domain-Gateway können ausgewählte Methoden eines Domain-Entity, Domain-Service oder Domain-Repository für den entfernten Aufruf veröffentlicht werden. Ein Domain-Gateway kann klassisch als Remote Stateless EJB oder alternativ als RESTful Webservice implementiert werden; dies entscheidet sich u. a. an der Fragestellung, ob Transaktionen zwischen den Business-Components propagiert werden müssen.

(2) Mit den drei Building Blocks **Domain-Event**, **Domain-Event-Sender** und **Domain-Event-Receiver** wird der asynchrone Austausch von geschäftlichen Ereignissen zwischen den Business-Components unterstützt. Aufgrund der Entkopplung von sendender und empfangender Klasse ist damit eine Möglichkeit eröffnet, um geschäftliche Funktionalität auch entgegen der DAG-Hierarchie der Business-Components aufzurufen - also ein Hilfsmittel zur Erreichung der Zyklen-Freiheit. Zur Implementierung des asynchronen Nachrichtenaustausches wird der Java Message Service (JMS) genutzt.
(3) Mit dem Building Block **Application-Service-Endpoint** erfolgt die Anbindung der Geschäftsvorgänge an sonstige Third-Party-Sys-

teme. Diese Endpoints für die Application-Services werden je nach Kommunikationspartner als RESTful Webservices (JAX-RS) oder SOAP Webservices (JAX-WS) implementiert.

Da in den Methoden-Signaturen der zuvor aufgeführten Building-Blocks ausschließlich Domain-Entity-Identifier, Domain-Attributes und Composite-Domain-Objects, aber keine Domain-Entities oder Domain-Values erlaubt sind, bleibt das interne Datenmodell und die Geschäftslogik in der Business-Component eingekapselt.
In dem Kunden-Projekt wird an der zentralen Unternehmens-Datenbank nach wie vor festgehalten, so dass auch hier die referentielle Integrität geprüft und transaktionale Geschäftsvorgänge über die Grenzen der Bounded Contexts hinweg realisiert werden können. Somit kann man in diesem Szenario jedoch die Business-Components nicht als **self-contained system (SCS)** betrachten, weil sie durch die übergreifenden Transaktionen sowie die gemeinsame Datenbank recht eng aneinander gekoppelt sind. Neben wenigen Nachteilen der Verteilung (z. B. Notwendigkeit der Schnittstellen-Versionierung) gewinnen wir einige Vorteile wie beispielsweise separate Release-Zyklen für die Business-Components oder individuelle Skalierbarkeit für jede Java-EE-Anwendung einzeln.

Architektur-Ansatz 3: Microservices

Aber natürlich kommen wir auch in diesem Artikel nicht um die unvermeidlichen Microservices herum - eben den wirklich handhabbaren Fingerfood-Häppchen.
Eine konsequente, kleinteilige Zerlegung des Anwendungs-Systems führt dazu, dass wir die ermittelten Bounded Contexts bzw. Business-Components gemäß dem Microservice-Ansatz implementieren. Aus der Welt der Java EE kommend beschäftigen wir uns intensiv mit der Spezifikation Eclipse MicroProfile (*https://projects.eclipse.org/projects/technology.microprofile*) und deren Implementierung durch das Microservice-Framework Quarkus (*https://quarkus.io*).
Die Kommunikation zwischen den Microservices erfolgt weitestgehend über RESTful Webservices. Zusammen mit dem Übergang zu einer dezentralen Datenhaltung je Microservice verlieren wir auf Datenbank-Ebene die referentielle Integrität und übergreifend gespannte, transaktionale Geschäftsvorgänge. Die übrigen Implikationen einer Microservice-Architektur sowie die Vor- und Nachteile dieses Lösungsansatzes können und wollen wir aus Platzgründen hier aber nicht ausdiskutieren.

Fazit

In der nunmehr abgeschlossenen Artikel-Reihe zur Software-Architektur wurde ein integrierter Architektur-Ansatz basierend auf den drei Lösungskonzepten Domain-Driven Design, Clean Architecture und Business-Components (Microservices) vorgestellt. Für diesen Ansatz liegt mit dem Flight Information System (FIS) eine vollständige Referenz-Implementierung vor. Ebenso bewährt sich die gewählte Kombination von Lösungskonzepten aktuell in einem großen Kundenprojekt. Wie gezeigt wurde, muss eine konkrete Architektur-Entscheidung jedoch immer unter Berücksichtigung verschiedener Faktoren getroffen werden; dazu zählen Firmen- und Projektgröße, fachliche Anforderungen und technische Rahmenbedingungen. Bei der Beratung unserer Kunden und in unseren Schulungsangeboten vertiefen wir diese Architektur-Diskussion auf Basis von Java EE / Jakarta EE und dem Quarkus-Framework.

Jens Seekamp [jens.seekamp@gedoplan.de]
Senior Consultant, Software-Architekt und Dozent mit langjähriger Praxiserfahrung rund um Java und Java EE bei der GEDOPLAN GmbH.

GEDOPLAN
IT Training

Remote-Schulungen – nicht nur als Alternative zum Präsenztraining!

Aufgrund der Covid-19-Problematik bieten wir jede Schulung als Remote-Schulung an - bislang mit großem Erfolg. Einige Schulungen haben wir bereits erfolgreich durchgeführt und die Teilnehmer waren sehr zufrieden. Dazu schreibt unser Geschäftsführer Dirk Weil Folgendes: "Ich persönlich habe bereits einige Remote-Schulungen mit dem Tool Zoom durchgeführt und kann sagen, dass die Teilnehmer sehr zufrieden waren und das neue Wissen trotz der räumlichen Entfernung gut aufnehmen konnten. Wir haben wie bei Präsenztrainings alle geplanten Themen behandeln können. Auch bei den Übungsaufgaben konnte ich gut mit Bildschirmfreigaben und Fernbedienung unterstützen."

Die Vorgehensweise

Der Teilnehmer entscheidet sich, ob er seinen eigenen Rechner benutzen möchte oder via Remote Desktop Protocol (RDP) einen vorbereiteten Schulungsrechner. Ein paar Tage vor der Schulung führen wir einen Zoom-Probedurchlauf durch und stellen das Tool und den Prozess in Ruhe vor. Dabei prüfen wir auch die Rechnerkonfigurationen. Die Seminarunterlagen inkl. Schulungsmaterialien erhalten die Teilnehmer per Post im Vorfeld, ebenso eine Einladung per Mail mit allen wichtigen Informationen zu der Schulung.

Die Voraussetzungen sind ein uneingeschränkter Internetzugang, Admin-Rechte für mögliche Installationen auf dem Schulungsrechner, eine Webcam (optional) und Lautsprecher. Der Teilnehmer sollte an einem ruhigen Platz sitzen.

Sie möchten unsere Remote-Trainingsmethoden kennenlernen?

Gerne stellen wir Ihnen unsere Remote-Schulungsmethode in einem persönlichen Remote-Meeting unverbindlich vor. Dirk Weil kann dabei gerne ein Kapitel aus dem Schulungsmaterial vorstellen - oder Sie erhalten einen kostenlosen 60-minütigen Vortrag aus unserer Vortragsreihe, Link s. u. Interessant? Schicken Sie uns eine Mail mit 2 Terminwünschen und wir treffen uns virtuell in einer Beispiel-Schulung!

Eine neue Schulungsform bei GEDOPLAN – auch nach der Pandemie

Wir sind von der Schulungsform überzeugt, so dass wir diese Schulungsart auch nach Corona anbieten werden. Die Schulungsformen sind dabei vielseitig: Offenes Training mit bis zu 8 Teilnehmern, maßgeschneiderte Firmenschulung, DUO-Training für 2 Mitarbeiter aus einer Firma und Einzelcoaching.

Neue Kurse bei GEDOPLAN IT Training

Wir haben einige neue Kurse mit umfangreichem Schulungsmaterial für die Teilnehmer erstellt und haben neue Trainings von starken Partnern.

Microservices mit Quarkus – kompakt
Jakarta EE mit MicroProfile kombinieren, um Microservices und verteilte Systeme für Quarkus zu entwickeln

Microservices mit Quarkus – Grundlagen oder Aufbau
Microservices und verteilte Systeme schnell und einfach entwickeln mit Quarkus, Jakarta EE und MicroProfile

JEE Architektur für Entscheider
Moderne Architektur-Konzepte für JEE im Überblick

Domain-Driven Design (DDD) in der Praxis
Grundlagen und praktische Nutzung von Domain-Driven Design

Jakarta RESTful Web Services (Dauer: 3,5 Stunden)
Entwicklung von RESTful Web Services auf Basis von Jakarta EE

Java Vertiefung (Standard Edition)
Fortgeschrittene Anwendungsentwicklung mit Java SE 11

Java Testing und Code Quality
Testautomatisierung und Qualitätssicherung in Java-Projekten

Java Testing für Entscheider
Testautomatisierung für Java-Projekte im Überblick

Testing mit JUnit und Mocking
Automatisiertes Testen in Java mit JUnit und Mocking-Frameworks

Python 3 Einführung
Grundlagenkurs Programmierung mit Python

Data Science mit Python
Effiziente Datenanalyse mit der Python-Toolchain

Refactoring
Wie man Code schreibt: wartbar, lesbar und verständlich

Clean Code
Code lesbar, wartbar und verständlich schreiben

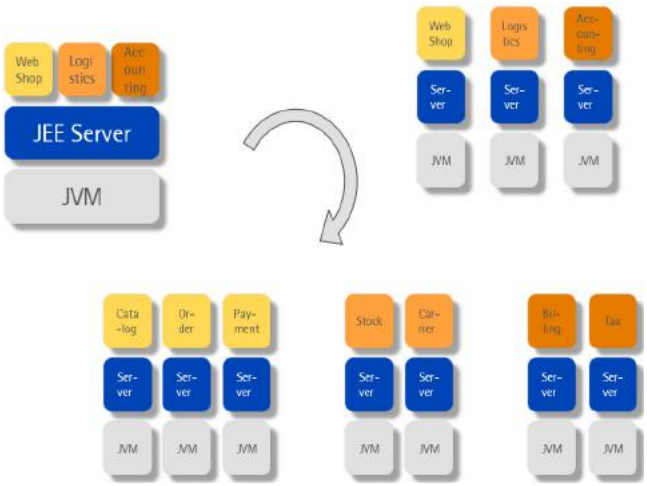
Scala Einführung
Einführung in die funktionale und objektorientierte Programmiersprache

Quarkus – leichtgewichtiges JEE mit Spaß!

Java-Anwendungen sind zumeist serverbasiert. Als Standard bietet Jakarta EE die notwendigen Zutaten wie Persistenz, Komponenteninjektion, Transaktionen, Webservices etc. an. Der als J2EE auf die Welt gekommene Standard blickt mittlerweile auf 22 Jahre Entwicklung mit Höhen und Tiefen zurück. Kritiker behaupten standhaft, JEE sei zu schwergewichtig und für neue Anwendungen nicht geeignet. Mit Quarkus liegt der Gegenbeweis auf dem Tisch – oder in der Entwicklungsumgebung!

Von Dirk Weil

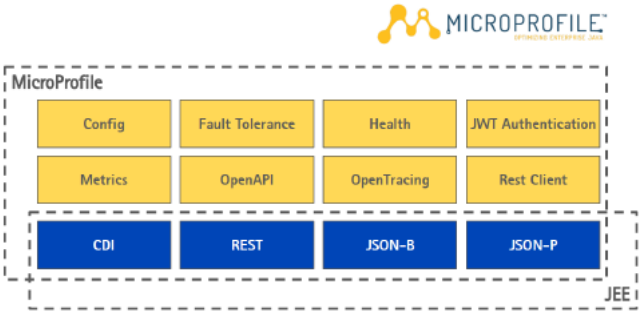
Die ursprüngliche Idee von J2EE war sicher, dass Application Server wie WebLogic oder WebSphere betrieben werden und jeweils mehrere JEE-Anwendungen beherbergen – wie eine Art Betriebssystem für Java-Server-Anwendungen. Organisatorisch und aus Gründen der Isolation der Anwendungen untereinander finden wir aber eher 1-zu-1-Beziehungen: Eine Anwendung läuft auf einem Server, der meist sogar auf einem dedizierten (virtuellen) Rechner betrieben wird. In Zeiten der Aufspaltung von Monolithen zu Modulithen (modularisierte Monolithen) oder Microservices ist eine Inflation der Serverinstallationen die Folge.



Es stellt sich somit die Frage, ob die separaten Server nötig sind oder ob man sie nicht in die Anwendungen integrieren kann. Diese Idee ist nicht neu: Spring Boot, Dropwizard, Micronaut u. a. m. implementieren dieses Modell. Und nun auch Quarkus. Brauchen wir noch ein weiteres Framework? Was ist das Besondere an Quarkus?

JEE und MicroProfile – oder der Wert von Standards

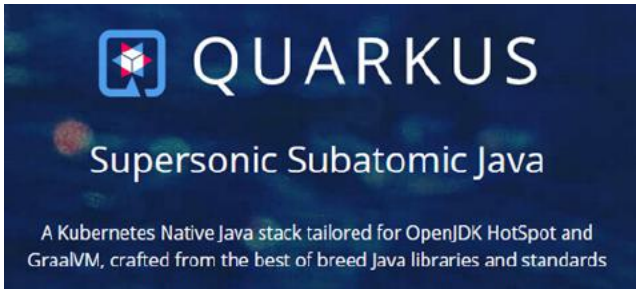
JEE hat eine bewegte Geschichte: 1998 als J2EE veröffentlicht, wurde der Standard 2006 in Java EE umgetauft. In den Jahren 2016 und 2017 war nicht klar, ob Oracle als Rechteinhaber überhaupt noch Interesse an Java EE hatte – Befürchtungen wurden laut, dass Java EE „eingestampft" würde. 2017 übertrug Oracle das Projekt an die Eclipse Foundation, wo der Standard unter dem Namen Jakarta EE als Open Source weiterentwickelt wird. Die neuerliche Umbenennung war notwendig, weil Oracle die Rechte am Namen Java EE behalten hat – für viele in der Community ein unverständliches Verhalten.
Aller Kritik zum Trotz stellt JEE einen großen Vorteil dar: Kompatibilität. Anwendungen können nahezu unverändert auf verschiedenen Serverprodukten betrieben werden. Soll's ein kostengünstiger Open-Source-Server sein oder lieber ein „großes" Produkt mit Support? Kein Problem: Der Kunde/das Team hat die Wahl – und auch ein späterer Wechsel ist möglich. Zudem hat es in den 22 Jahren J2EE/Java EE/Jakarta EE kaum Breaking Changes gegeben. Ältere Anwendungen können somit meist ohne größere Änderungen auch auf aktuellen Servern (mit aktuellen JVMs und aktuellen Security Patches) weiter betrieben werden.
Diese Abwärtskompatibilität ist nicht leicht zu erhalten und stellt sicher einen der Gründe dafür dar, dass die Releasezyklen von 3 - 4 Jahren recht groß waren und sich der Standard für viele zu langsam weiterentwickelte. Hier kommt MicroProfile ins Spiel – ebenfalls ein Open-Source-Projekt der Eclipse Foundation. MicroProfile optimiert Enterprise Java nach eigener Aussage für Microservice-Architekturen. Es ergänzt JEE um diverse Bausteine, die für verteilte Architekturen – insbesondere in der Cloud – nötig sind.



Die Kombination von JEE und MP erscheint recht sinnvoll: Während der „Tanker" JEE zuverlässig im Standard-Fahrwasser – nun mit Releases ungefähr im Jahresabstand – vorankommt, setzt MP „Schnellboote" ins Wasser und unterstützt mit Minor Releases alle paar Monate aktuelle Bedarfe in den Bereichen Microservices, Cloud und Serverless.
Bemerkenswert ist, dass alle aktuellen JEE-Produkte auch MP unterstützen: u. a. WildFly, Payara, Open Liberty – und auch Quarkus.

Supersonic Subatomic Java

Quarkus ist nicht gerade bescheiden in der Beschreibung auf quarkus.io



Überschallgeschwindigkeit, winzige Größe und Java – wir werden im Folgenden sehen, wie das zusammenpasst. Quarkus kommt wie WildFly aus dem Hause Red Hat. Das Open-Source-Projekt ist in gewisser

Weise der Nachfolger von Thorntail, dessen Weiterentwicklung mit Ablauf des Jahres 2020 endet. Quarkus ist optimiert für extrem kurze Startzeiten und läuft sowohl auf einer klassischen JVM wie auch nativ mit Hilfe der GraalVM.
Dabei ist Quarkus sowohl Framework als auch Toolset: Der Project Setup erfolgt mit Hilfe eines Maven-Plugins (Gradle geht auch). Dabei werden die erforderlichen Framework-Bestandteile wie Lego-Bausteine zu einer Menge von Dependencies kombiniert, auf deren Basis der Anwendungscode ergänzt werden kann. So erstellt der folgende Befehl bspw. eine Anwendung, die den klassischen Stack für Microservices inkl. Health Checking enthält:

```
mvn io.quarkus:quarkus-maven-plugin:create \
  -DprojectGroupId=de.gedoplan.showcase \
  -DprojectArtifactId=quarkus-rest-cdi-jpa \
                -Dextensions="resteasy-jsonb,hibernate-orm,jdbc-
h2,smallrye-health"
```

Das Quarkus-Plugin wird auch im Build des Projektes genutzt. Es erstellt ein ausführbares Jar-File mit dem Anwendungscode sowie ein Verzeichnis, in dem alle benötigten Libraries liegen. Diese Aufteilung – Thin-Jar + Dependencies – ist vorteilhaft bspw. für Docker-Images, weil bei einem erneuten Build der Layer mit den Dependencies nicht erneut gebaut werden muss. Das Jar-File kann einfach über java -jar ... gestartet werden.

Hot Reload im Development Mode

So praktisch die Integration des Servers in die Anwendung für den Produktivmodus ist – für die Entwicklungszeit muss die Anwendung nach Anpassungen allerdings gestoppt und neu gestartet werden, da ja kein separater Server existiert, in dem die Anwendung einfach neu deployed werden könnte. Nicht so bei Quarkus: Hier kann die Anwendung im Development Mode gestartet werden: mvn quarkus:dev. Quarkus überwacht dann den Quellcode der Anwendung und führt einen automatischen Reload durch, wenn nach Änderungen z. B. wieder auf das Rest API der Anwendung zugegriffen wird. Das geschieht in relativ kurzer Zeit, bei einem vollständigen kleinen Microservice bspw. innerhalb von 1 bis 2 Sekunden. Man kann somit bei laufender Anwendung weiter entwickeln und hat bei jedem Aufruf ohne große Verzögerung den aktuellen Entwicklungsstand im Zugriff.

Einfache (Auto-) Konfiguration

Quarkus-Anwendungen lassen sich mit einer zentralen Konfigurationsdatei im Properties- oder Yaml-Format konfigurieren – ganz ähnlich wie dies in Spring Boot gehandhabt wird. Und die verschiedenen Framework-Teile bringen eine funktionsfähige Grundkonfiguration mit. So wäre in o. a. Anwendung durch die Integration des O/R-Mappers Hibernate und der H2-Datenbank automatisch eine Datasource (auf eine In-Memory-DB) und eine Persistence Unit für die Nutzung in JPA konfiguriert. Die Einstiegshürde ist also bewusst niedrig gelegt: Man kommt schnell zu lauffähigen Ergebnissen und passt die Konfiguration einfach nach Bedarf an – Hot-Reload-fähig natürlich! Für bestimmte Ausführungssituationen lässt sich die Konfiguration über sog. Configuration Profiles modifizieren. Und auch dafür gibt es teilweise schlau vorbesetzte Werte. So ist z. B. der Standard-Port für Web-Zugriffe 8080. Im Test ist dagegen 8081 voreingestellt, sodass während der Entwicklung im Development Mode auch Tests ausgeführt werden können ohne dass Port-Konflikte entstehen.

Testen

Multi-Unit-Tests und Integrationstests sind im Enterprise-Bereich normalerweise keine leichte Kost. Für klassische JEE-Server wird hier i. d. R. Arquillian eingesetzt, ein Testframework mit umfangreichen Möglichkeiten, das aber aufgrund seiner Flexibilität recht schwierig aufzusetzen ist. Hier geht Quarkus wieder einen anderen, einfacheren Weg. JUnit-5-Testklassen lassen sich über die Annotation @QuarkusTest zu Integrationstests machen: Beim Ablauf fährt die Anwendung automatisch hoch und kann über externe Schnittstellen sofort „Black Box"-getestet werden. Da in der Testklasse zudem sämtliche von CDI gewohnte Injektionsmöglichkeiten zur Verfügung stehen, sind auch White Box Tests unproblematisch. Auch hier ist die Einstiegsschwelle so niedrig, dass Tests schnell und einfach integriert werden können und Testen wieder (?) Spaß macht.

Native Mode

Quarkus ist im normalen JVM-Betrieb schon hinsichtlich der Startzeiten und Ressourceverbräuche optimiert. So benötigt bspw. ein kleiner Service mit einem Rest API und Datenbank-Anbindung über JPA – also ein vollständiger Service, nicht etwa ein belangloser Hello-World-Service – mit Quarkus und OpenJDK unter einer Sekunde bis zum ersten Response, während bei einem klassischen Stack wie Spring Boot gut 4 Sekunden zu verbuchen wären.

Dabei werden etwa 74 MB Memory belegt. Diese von quarkus.io übernommenen Werte lassen sich durchaus mit eigenen Demos belegen. Schauen Sie sich bei Interesse mal unsere Demo unter github.com/GEDOPLAN/micro-comparison an.
Wenn's jetzt noch kleiner und noch schneller sein soll, was z. B. bei Betrieb in der Cloud von Interesse ist, oder wenn schnell nach Bedarf skaliert werden soll, dann kann eine Quarkus-Anwendung mit Hilfe der GraalVM in Maschinencode kompiliert werden. Das dauert im Build zwar einige Zeit (für die Demo-Anwendung sind's etwa 3 Minuten), erzeugt im Betrieb aber beeindruckende Ergebnisse: Die Speicherbelegung sinkt nochmals erheblich und die Startzeiten liegen nun im Millisekunden-Bereich.

Und was ist mit den Standards?

Über die beschriebenen Dinge hinaus bietet Quarkus einen fast ganz normalen Stack aus JEE und MicroProfile an: Rest-Endpoints, JPA-Entities, CDI-Services etc. lassen sich im Programmcode nicht vom klassischen JEE-Stack unterscheiden – Quarkus ist hier eben kein proprietäres Framework, sondern stützt sich vollständig auf dem Standard ab. Das gilt ebenso für die jüngeren Bausteine aus MicroProfile: Health Checking, Monitoring, Resilience etc. funktionieren hier mit Code, der identisch auch auf einem WildFly oder Open Liberty eingesetzt werden könnte. Für uns JEE-Experten hat das zwei Vorteile: Wir können unser Wissen weiter einsetzen, ohne umlernen zu müssen. Und die Entscheidung für das Runtime-Framework ist nicht unumstößlich: Stellen wir nach einiger Zeit fest, dass – aus welchen Gründen auch immer – ein Betrieb in einem klassischen Server besser wäre, bedeutet das keine Neuentwicklung, sondern nur eine überschaubare Migration.
Eine Einschränkung muss allerdings erwähnt werden: Ein Teil der Optimierungen wird dadurch erreicht, dass einige Initialisierungen, die ein normaler CDI-Container beim Deployment mit Hilfe von Reflection ausführen würde, bei Quarkus bereits zur Build-Zeit erfolgen. Dieser Augmentation genannte Vorgang setzt einen entsprechend angepassten CDI-Container namens ArC voraus. Er implementiert den CDI-Standard nicht vollständig. Die Einschränkungen sind allerdings praxisorientiert und relativ gut zu akzeptieren. Für den Native Mode gibt es weitere Einschränkungen, die sich daraus ergeben, dass in einem Maschinencode nicht die gleichen dynamischen Möglichkeiten existieren wie in einer JVM, die Bytecode ausführt.

Wie geht's weiter?

Mit dieser Frage kann zunächst einmal das Quarkus-Ökosystem gemeint sein. Und da tut sich recht viel. Die Releases des Frameworks erscheinen in kurzer Folge (im ersten Quartal von 2020 alleine fünf Final-Releases) und für die Anbindung von Drittsystemen erscheinen in atemberaubender Geschwindigkeit Quarkus-Extensions, z. B. für Kafka, MongoDB, Liquibase, Camel, Scheduling, Mail u. a. m. Auch Spring-Komponenten lassen sich mittlerweile integrieren.
Zum anderen ist nun natürlich die Frage, wie Sie nun weitermachen. Schauen Sie sich doch einmal unsere kostenlosen Expertenkreis-Vorträge zu den Themen MicroProfile und Quarkus an (gedoplan-it-consulting.de/termine/expertenkreis-java-archiv/). Und dann freuen wir uns natürlich, wenn wir Sie in einem unserer Seminare zum Thema begrüßen dürfen (gedoplan-it-training.de/suche/?q=quarkus).

Dirk Weil[dirk.weil@gedoplan.de]
Geschäftsführer der GEDOPLAN GmbH
Fachbuchautor, schreibt Artikel für Fachmagazine, hält Vorträge und leitet Seminare und Workshops zu diversen Java-SE-/EE-Themen.

Angular Materials Theming

Angular als Framework für die Entwicklung von anspruchsvollen Webanwendungen bringt alles mit, was der Entwickler braucht. Alles? Nicht ganz. Ähnlich wie bei JSF werden wir in aller Regel zusätzliche Komponenten integrieren, die z. B. eine Kalenderauswahl möglich macht oder eine Tabelle mit Filter- und Sortier-Funktion anbietet. Neben PrimenNG (Primetek, Primefaces) kommt aus der Feder von Google selbst @angular/material.

Von Dominik Mathmann

Angular Material ist dabei nicht einfach nur eine Komponenten-Bibliothek für Angular, sondern basiert bei der Gestaltung und technischen Umsetzung auf Googles Designsprache: „Material Design". Erstmals 2014 vorgestellt, handelt es sich dabei um eine Art Styleguide für die Entwicklung von Anwendungen. Neben grundsätzlichen Regeln zur Anordnung, Verwendung von Schatten, Formen und Animationen ist natürlich auch die Typographie und Farbgebung ein wichtiges Thema. Eine detaillierte Einführung in die Thematik bietet dafür die offizielle Seite: https://material.io/ .

Angular Material Komponenten

Bevor wir uns aber im Detail um die Bereiche Typographie und Farbgebung à la Angular-Material kümmern, schauen wir uns zuerst einmal Angular Material an. Angular Material ist wie bereits erwähnt eine Komponenten-Bibliothek, die allerhand fertige Komponenten bietet. Um diese verwenden zu können, müssen wir die entsprechenden Abhängigkeiten zu unserer Anwendung hinzufügen und installieren. Bei der Verwendung der Angular-CLI erledigt folgender Aufruf die Installation und eine erste Konfiguration:

```
ng add @angular/material
```

Führt unter anderem folgende Schritte durch:

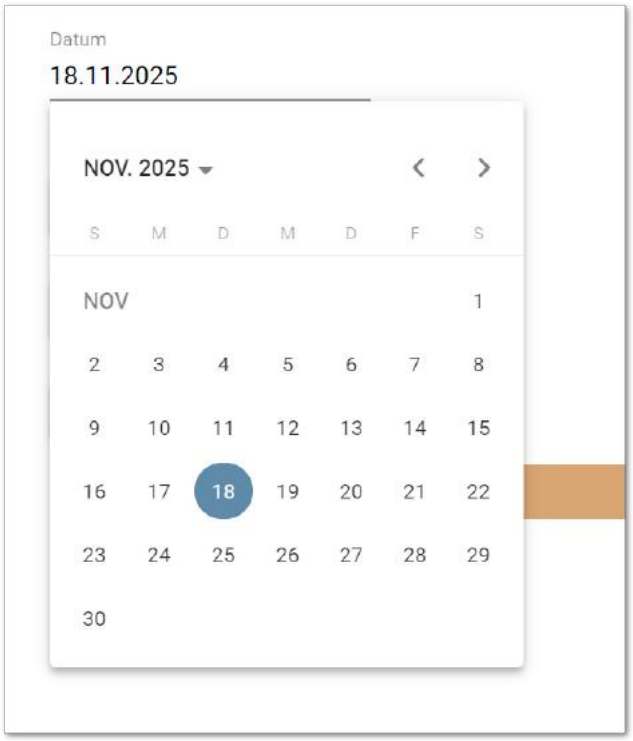
- fügt der package.json die benötigten Abhängigkeiten hinzu
- fügt der angular.json die Referenz auf Theme-CSS-Datei hinzu (falls eines der default-Themes gewählt wurde)
- fügt der index.html Links auf Default-Font und Material-Icon-Font hinzu.

Die zur Verfügung stehenden Komponenten sind nun aus Performance-Gründen in einzelne Module aufgeteilt, die in unserer eigenen Anwendung zur Benutzung importiert werden müssen. Als Beispiel nehmen wir hier den oft verwendeten „DatePicker":

```
import {MatDatepickerModule} from '@angular/material';

@NgModule({
  declarations: [AppComponent],
  imports: [
     BrowserModule,
     MatDatepickerModule,
     ...
  ]
```

Die so importierte Komponente kann dann wie gewohnt innerhalb unserer Templates verwendet und mit benötigten Property- und Event-Bindings angereichert werden:

```
<input matInput [matDatepicker]="datePicker"
     (focus)="datePicker.open()">
<mat-datepicker #datePicker></mat-datepicker>
```



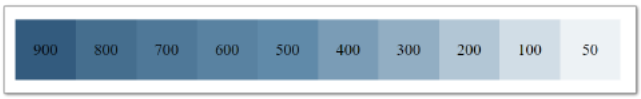
Wie wir sehen, ist die Verwendung solcher Komponenten recht einfach. In aller Regel passen diese Komponenten allerding farblich nicht zum Look&Feel unserer Anwendung und müssen entsprechend angepasst werden. Da den Komponenten natürlich CSS-Regeln zugrunde liegen, ließe sich eine solche Anpassung durchaus durch das Überschreiben der entsprechenden Styling-Regel bewerkstelligen. Dank der Developer-Konsole des Browsers lässt sich sehr einfach erkennen, welche Regeln hier Anwendung finden. So ließe sich mit folgender globalen CSS-Regel die Hintergrundfarbe des ausgewählten Datums anpassen:

```
.mat-calendar-body-selected {
  background-color: green;
}
```

Ein gangbarer Weg, jedoch verwenden wir in aller Regel eine Vielzahl von Materials-Komponenten, deren Klassen alle überschrieben werden müssten und die dann bei späterer Anpassung alle erneut Änderungen nach sich ziehen. In Bezug auf Schrift und Farbe begegnet uns nun das Thema „Themes", das solche Anpassungen global möglich macht. Angular Materials basiert auch in diesen Bereichen auf dem Material Design, also machen wir uns zuerst mit der Grundidee vertraut.

Farb- und Typografie-System

Das Farbsystem von Material Design ist mit Absicht sehr einfach gehalten. Die Grundlage bilden sogenannte Farbpaletten. Eine Farbpalette besteht immer aus einer Farbe, die in verschiedenen Helligkeitsabstufungen zur Verfügung steht.



Innerhalb der Angular-Anwendung werden in aller Regel mindestens 3 solcher Paletten verwendet: Primär-Palette, Sekundär-Palette und eine Palette für Warnungen (dazu später mehr), die jeweils in einem hellen und einem dunklen Theme integriert werden.
Auch die Richtlinien zur Schriftsetzung ist übersichtlich. Ausgehend von einer zentralen Standard-Schriftart werden im Grunde lediglich einige Bereiche benannt (input, button, headline, title, body…), deren Verwendung erläutert und technisch mit folgenden CSS-Regeln versehen wird:

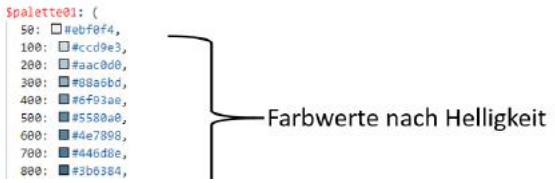
- font-size
- line-height
- font-weight
- font-family (optional abweichend von Standard)

Angular Material Themes

Angular Materials bietet 4 Standard-Themes, die bereits beim Hinzufügen der Abhängigkeiten über die Angular-CLI ausgewählt werden können. Die Auswahl führt wie bereits beschrieben zu einer entsprechenden Referenz in der Datei: angular.json auf die CSS-Datei des Themes. Alternativ zu diesem Vorgehen kann das Material Theme auch via Import eingebunden werden, indem eine der folgenden Importe im eigenen Root-Style deklariert wird:

```
@import
   '@angular/material/prebuilt-themes/deeppurple-amber.css';
@import '@angular/material/prebuilt-themes/indigo-pink.css';
@import '@angular/material/prebuilt-themes/pink-bluegrey.css';
@import '@angular/material/prebuilt-themes/purple-green.css';
```

Der Clou: alle Komponenten von Angular Materials bedienen sich der Farbpalette plus Abstufungen und auch der Deklarationen der Schriftarten aus den verschiedenen Bereichen. So lassen sich global alle Komponenten einheitlich und konsistent in Farbgebung und Schriftarten anpassen: ganz einfach durch die Aktivierung des Themes.

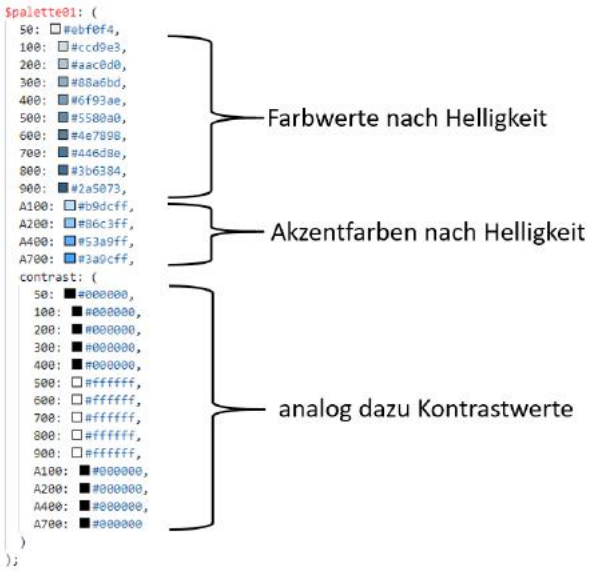


Custom Angular Theme

Wenn die vorgefertigten Themes für unsere Anwendung nicht in Frage kommen, reicht die Verwendung von CSS nicht mehr aus. Angular Material setzt dabei auf Sass als sogenannten CSS-Präprozessor, der viele interessante Features mitbringt, die auch (unabhängig von Angular Material) in der eigenen Anwendung verwendet werden können. Ähnlich wie TypeScript in Bezug auf JavaScript werden die scss-Dateien in css-Dateien umgewandelt, erlauben es aber z. B. Funktionen zu deklarieren, Berechnungen durchzuführen und Variablen für das Styling zu verwenden. Für Details s. h*ttps://sass-lang.com/* .
Ein eigenes Theme erlaubt es nun, selber Farbpaletten und Schriftarten zu setzen, die dann genauso von den Komponenten aufgegriffen werden. Neben den einfachen Farbpaletten, die bereits in der Einleitung gezeigt wurden, ergänzt Angular diese Idee mit zwei weiteren Konzepten:

- Kontrast-Farben: zu jedem Helligkeits-Wert der Farbpalette existiert ein Kontrast-Wert, um z. B. Schriften gut lesbar auf den farbigen Flächen darstellen zu können
- Akzent-Farben

Dazu werden im Folgenden sogenannte Sass-Mixins verwendet, die Angular genau für diesen Zweck definiert hat und die wie Methoden für CSS-Dateien funktionieren, indem sie Parameter entgegen nehmen und einen dynamischen Wert zurückliefern.
Die Definition einer solcher Palette ist relativ einfach. Technisch gesehen handelt es sich dabei um eine Map, welche die Farbewerte definiert, die für die verschiedenen Helligkeitsstufen verwendet werden:



Für ein vollständiges Theme werden 3 solcher Paletten benötigt: Primär-Palette, Sekundär-Palette und eine Palette für Warnungen. Bei der Erstellung eines eigenen Themes müssen diese Paletten selber geschrieben werden. Im Netz existieren jedoch eine ganze Reihe von Webseiten, die das Erstellen solcher Paletten vereinfachen (zum Beispiel: http://mcg.mbitson.com). Über ein entsprechendes Mixing, welches wir über den unten gezeigten Import bekommen, wird aus diesen 3 Paletten dann ein Theme (light oder dark), das für die Angular Materials Komponenten registriert werden kann:

```
@include mat-core();

$theme: mat-light-theme(
  mat-palette($palette01),
  mat-palette($palette02),
  mat-palette($palette03)
);
@include angular-material-theme($theme;
```

Das Vorgehen bei den Schriftarten basiert auf derselben Idee. Pro Kategorie (vollständige Liste unter: https://material.angular.io/guide/typography) definieren wir size, line-height, weight und optional eine abweichende font-family. Diese registrieren wir ebenfalls mit einem entsprechenden Mixin:

```
$custom-typography: mat-typography-config(
  $font-family: 'Roboto, "Helvetica Neue", sans-serif',
  $headline: mat-typography-level(24px, 32px, 400),
  $button: mat-typography-level(14px, 14px, 500),
  $input: mat-typography-level(inherit, 1.125, 400)
);

@include angular-material-typography($custom-typography);
```

Eigene Komponenten stylen

Haben wir die Angular Materials-Komponenten erst einmal aufgehübscht, stoßen wir schnell auf den Bedarf, die so registrierten Farbwerte auch in den eigenen Komponenten zu verwenden. Auch dazu bietet Angular entsprechende Mixin-Funktionen an, die wir nutzen können, um bestimmte Werte aus den übergebenen Paletten aus zu lesen (über import von ‚~@angular/material/theming '). Um dies möglichst transparent und wartbar zu halten, bietet es sich an, die Definition von Farben, Schriften und Paletten und die Registrierung zu trennen. Folgende Aufteilung hat sich in unseren Projekten mit Sass bewährt:

theme.scss

- Definition von Farben
- Definition von Paletten
- Definition von Theme

variables.scss

- import theme.scss und ~@angular/material/theming
- Definition allgemein gültiger SASS Variablen (z.B. Abstände)
- Definition häufig verwendeter Farb- und Typographie-Werte

```
@import '~@angular/material/theming';
@import 'theme.scss';

$primary-color: mat-color($palette01, 400);
$accent-color: mat-color($palette02, 400);
$warn-color: mat-color($palette03, 400);

$primary-contrast-color: mat-color($palette01, contrast(400));
$accent-contrast-color: mat-color($palette02, contrast(400));
$warn-contrast-color: mat-color($palette03, contrast(400));

$font-family: mat-font-family($custom-typography);
$font-title-size: mat-font-size($custom-typography, title);
```

style.scss (root – Style)

- import der theme.css
- include mat-core()
- Registrierung des Themes (wie oben zu sehen)

Die Style-Datei der einzelnen Komponente benötigt dann lediglich einen einzelnen Import und benötigt keinerlei direkte Abhängigkeiten zu den Sass-Funktionen:

```
@import 'variables.scss';

header {
  border-bottom: 2px solid $primary-color;
}
```

Anmerkung: der hier gezeigte absolute Import der variables.scss funktioniert nur durch einen zusätzlichen Eintrag in der angular.json unter architect/build/options

```
"styles": ...,
"stylePreprocessorOptions": {
        "includePaths": ["src"]
  },
```

Damit sparen wir uns die Speicherortabhängigen relativen Referenzen auf unsere grundlegenden scss Dateien, die unter „src" zu finden sind.

Eigene Library-Komponenten stylen

Wie oben zu sehen, importieren wir innerhalb unserer Komponenten die anwendungsspezifische variables.scss, die wiederum auf das anwendungsspezifisch definierte Theme zugreift. Bei Komponenten, die im eigenen Projekt liegen, mag das noch funktionieren, aber spätestens, wenn wir unsere eigenen Komponenten in ein separates Library-Projekt auslagern, um sie in verschiedene Projekte zu importieren, funktioniert dieser einfache Ansatz nicht mehr. Unsere eigene Komponenten-Bibliothek muss also analog zum Vorgehen bei Angular Materials durch entsprechende Mixin-Funktionen mit dem aktuellen Theme versorgt werden. Um das zu tun, bedarf es einiger Vorarbeit innerhalb unserer Komponenten.
Zusätzlich zu der üblichen Komponenten-Style-Datei implementieren wir eine theme.scss für unsere Bibliothek. Diese Datei enthält alle Regeln mit Theme-spezifischen Eigenschaften für die in der Bibliothek enthaltenden Komponenten. Hier implementieren wir dann die benötigten Mixin-Funktionen:
@import '~@angular/material/theming';

```
@mixin ng-lib-theme($theme) {
  $primary: map-get($theme, primary);
  $accent: map-get($theme, accent);

  lib-panel div.body {
    border-top-color: mat-color($primary, 400);
    border-bottom-color: mat-color($accent, 400);
  }
}

@mixin ng-lib-typography($config) {
  lib-panel div.body {
    font-size: mat-font-size($config, title);
  }
}
```

Wie zu sehen, definieren wir jeweils eine eigene Mixin-Funktion für das Theme und Typografie, die mit einem entsprechenden Parameter versorgt werden (Theme-Objekt bzw. Typografie-Konfig). Aus dem Theme können wir dann z. B. mittels „map-get" die einzelnen Paletten extrahieren (primary, accent, warning), die dann wiederum dazu genutzt werden können, um konkrete Farbwerte zu ermitteln. Diese werden dann wie bereits gesehen in einzelnen CSS-Regeln verwen-

det, um Theme spezifische Ergänzungen zu den Komponente-Styles zu liefern.
Innerhalb der konkreten Anwendung ähnelt die Registrierung dann erschreckend der von Angular Materials:

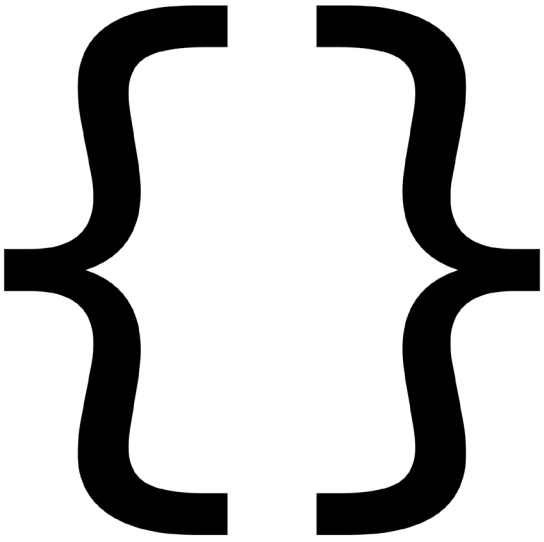
```
@include ng-lib-theme($theme02);
@include ng-lib-typography($custom-typography);
```

Zugegeben, mit „ein bisschen" CSS ist es hier nicht getan. Auch um die Verwendung von Sass als CSS-Präprozessor kommen wir an dieser Stelle nicht mehr herum. Der ist aber ohnehin zu empfehlen, auch wenn man im eigenen Projekt die Möglichkeiten von Sass vielleicht gar nicht ausschöpft oder sogar nur Kernfunktionen wie die Definition von Variablen nutzt. Ob die Anbindung der eigenen Komponenten an das Material Theming, wie zuletzt gesehen, gewünscht ist, hängt sicher von der Bibliothek selbst ab. Insbesondere bei internen Komponenten-Bibliotheken, die keinerlei dynamischen Theming-Funktionalitäten benötigen, erfüllt eine zentrale Library mit festgelegtem Look&Feel (in unserem Beispiel wäre das die theme.scss und variables.scss) sicherlich auch seinen Zweck und macht die Entwicklung / Styling der eigenen Komponenten etwas einfacher.

Fazit

Unterm Strich bietet uns das Theming von Angular Material ein mächtiges Instrument, um global auf das Aussehen der Komponenten einzuwirken. Darüber hinaus liefert das System genügend Anknüpfpunkte, um auch für unsere eigenen Komponenten eine solide Basis für Farben und Schriften zu liefern.

Dominik Mathmann[dominik.mathmann@gedoplan.de]
Berater, Entwickler und Trainer bei der GEDOPLAN GmbH. Auf Basis seiner langjährigen Erfahrung in der Implementierung von Java-EE-Anwendungen leitet er Seminare, hält Vorträge und unterstützt Kunden bei der Konzeption und Realisierung von Webanwendungen vom Backend bis zum Frontend.



News from the Blog

GEDOPLAN betreibt seit Jahren einen Blog, in dem wir über Neuigkeiten und Tipps rund um Java EE berichten. Regelmäßig informieren unsere Mitarbeiter von ihren Erfahrungen in aktuellen Projekten, geben Tipps, wie kleine Probleme mit Java EE gelöst werden können oder regen Diskussionen um unterschiedliche Lösungsmöglichkeiten an.
Einige der wichtigsten Beiträge möchten wir Ihnen in unserer Rubrik „News from the Blog" in unregelmäßigen Abständen kurz vorstellen. Vielleicht wecken wir Ihr Interesse und Sie besuchen unseren Blog unter javaeeblog.wordpress.com, um sich zu informieren und den gesamten Beitrag zu lesen. Oder, was uns noch mehr freuen würde, um sich an den Diskussionen zu beteiligen.

Angular

Sie wollen weitere Informationen rund um das Thema Angular? Unser Autor Dominik Mathmann befasst sich in unserem Blog mit Themen in direktem Zusammenhang mit Angular.

Angular Theming, Library-Komponenten

Dieser Blogeintrag beantwortet die Frage, wie man Komponenten einbindet, die in einem seperaten Projekt liegen. Und "zugegeben, mit einem bisschen css ist es hier nicht getan".

Angular – im Griff – Compodoc

Komponenten, Services, Module... Angular-Anwendungen können sehr schnell sehr groß werden. Dieser Artikel stellt Ihnen zwei Hilfsmittel vor, um den Überblick zu bewahren: eine gute Struktur und ein kleines Tool.

maven vulnerability check

Wir stellen Ihnen einen Weg vor, wie Sie mit regelmäßigen Kontrollen der Versionen Ihrer Bibliotheken die Sicherheit in Ihren Projekten erhöhen.

MicroProfile Fault Tolerance

Was passiert, wenn in verteilten Anwendungen einzelne Module nicht erreichbar sind? Dirk Weil stellt Ihnen mit *MicroProfile Fault Tolerance* ein Werkzeug vor, das verhindert, dass nicht wie in einem Dominoeffekt die gesamte Anwendung von diesem Ausfall betroffen ist.

GEDOPLAN Java EE Blog
Hier finden sich Neuigkeiten, Ideen und Kommentare rund um Java EE

javaeeblog.wordpress.com

GEDOPLAN

IT Training & IT Consulting

GEDOPLAN IT Training Seit 1998 schulen wir Java, viele Seminare auch in englischer Sprache. Unsere Schulungsleiter sind Java-Experten aus der Praxis, die ihr Wissen in Java-Projekten selbst unter Beweis gestellt haben. Unsere Java-Schulungen beinhalten neben den theoretisch notwendigen Grundlagen auch einen entsprechend hohen Praxisanteil. In unseren Seminaren gehen wir auf aktuelle Problemstellungen des Alltags ein. Damit Sie den bestmöglichen Erfolg erzielen, bieten wir zwei Formen an: In offenen Kursen vermitteln unsere Java-Experten ihr Know-how zu unterschiedlichen, definierten Schwerpunkten. Benötigen Sie Expertenwissen für sehr spezielle Java Fragen oder Projekte, führen wir individuelle Gruppen- und Firmenschulungen durch, die wir auf Ihre konkreten Bedürfnisse abstimmen.

GEDOPLAN IT Consulting steht seit vielen Jahren für hochwertiges Consulting in den Java-Technologien. Wir setzen auf offene Standards und Open Source Produkte. Die Java EE Plattform ist unsere Basis für die Entwicklung betrieblicher Anwendungen. Plattformen wie WildFly und Liferay führen schnell und sicher zum Ziel.
Ob Neuentwicklung, Migration nach Java EE oder Codereviews: Wir entwickeln IT-Systeme als Komplettpakete, unterstützen unsere Kunden aber auch gerne vor Ort.

GEDOPLAN
Unternehmensberatung und
EDV-Organisation GmbH
Stieghorster Straße 60
33605 Bielefeld
Fon: + 49 521 / 2 08 89 10
Fax: +49 521 / 2 08 89 45
info@gedoplan.de

Geschäftsstelle Berlin:
GEDOPLAN GmbH
UPPER WEST
Kantstraße 164
10623 Berlin

Postadresse Berlin:
GEDOPLAN GmbH
Kurfürstendamm 11
10719 Berlin

+49 30 / 755 49 188
it-training@gedoplan.de.

Herausgeber: GEDOPLAN Unternehmensberatung und EDV-Organisation GmbH | Stieghorster Straße 60 | 33605 Bielefeld
Verantwortlich: Dipl.-Inform. Dirk Weil | Redaktionsleitung: Ulrich Hake | Fon: 0521 2088910 | E-Mail: info@gedoplan.de
Produktion: networker Medienfabrik GmbH | Nachdruck, auch auszugsweise, nur nach Rücksprache mit der Redaktion.

www.gedoplan.de