Curriculum für

CPSA Certified Professional for Software Architecture®

– Advanced Level –

Modul:
FLEX

Flexible Architekturmodelle - Microservices und Self-Contained Systems

Version 1.1 (November 2015)

Inhaltsverzeichnis

0 Einleitung: Allgemeines zum iSAQB-Advanced-Level

0.1 Was vermittelt ein Advanced-Level-Modul?

- Der iSAQB-Advanced-Level bietet eine modulare Ausbildung in drei Kompetenzbereichen mit flexibel gestaltbaren Ausbildungswegen. Er berücksichtigt individuelle Neigungen und Schwerpunkte.
- Die Zertifizierung erfolgt als Hausarbeit. Die Bewertung und mündliche Prüfung wird durch vom iSAQB benannte Experten vorgenommen.

0.2 Was können Absolventen des Advanced Level (CPSA-A)?

CPSA-A-Absolventen können:

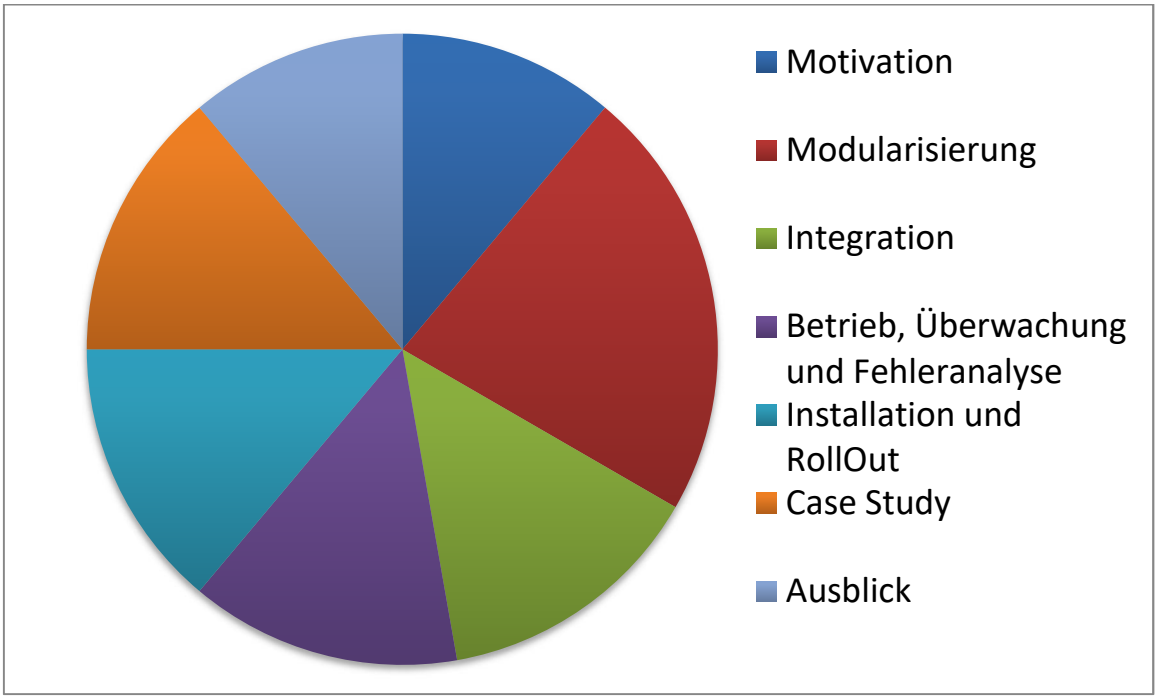
- Eigenständig und methodisch fundiert mittlere bis große IT-Systeme entwerfen.
- In IT-Systemen mittlerer bis hoher Kritikalität technische und inhaltliche Verantwortung übernehmen.
- Maßnahmen zur Erreichung nichtfunktionaler Anforderungen konzeptionieren, entwerfen und dokumentieren. Entwicklungsteams bei der Umsetzung dieser Maßnahmen begleiten.
- Architekturrelevante Kommunikation in mittleren bis großen Entwicklungsteams steuern und durchführen.

0.3 Voraussetzungen zur CPSA-A-Zertifizierung

- Eine erfolgreiche Ausbildung und Zertifizierung zum CPSA-F Certified Professional for Software Architecture Foundation Level®
- Mindestens drei Jahre Vollzeit-Berufserfahrung in der IT-Branche, dabei Mitarbeit an Entwurf und Entwicklung von mindestens zwei unterschiedlichen IT-Systemen
    - Ausnahmen auf Antrag zulässig (etwa: Mitarbeit in OpenSource-Projekten)
- Aus- und Weiterbildung im Rahmen von iSAQB-Advanced-Level-Schulungen im Umfang von mindestens 70 Credit Points aus allen drei unterschiedlichen Kompetenzbereichen (detailliert geregelt in Abschnitt 1.2).
    - Bestehende Zertifizierungen können ggfs. auf diese Credit Points angerechnet werden. Die Liste der aktuellen Zertifikate, für die Credit Points angerechnet werden, ist auf der iSAQB-Homepage zu finden.
- Erfolgreiche Bearbeitung der CPSA-A-Zertifizierungsprüfung.

1 Grundlegendes zum Modul FLEX

1.1 Gliederung des Lehrplans für FLEX und empfohlene zeitliche Aufteilung



- Motivation 02:00
- Modularisierung 04:00
- Integration 02:30
- Betrieb, Überwachung und Fehleranalyse 02:30
- Installation und Roll Out 02:30
- Case Study 02:30
- Ausblick 02:00

1.2 Dauer, Didaktik und weitere Details

Die unten genannten Zeiten sind Empfehlungen. Die Dauer einer Schulung zum FLEX sollte mindestens 3 Tage betragen, kann aber länger sein. Anbieter können sich durch Dauer, Didaktik, Art- und Aufbau der Übungen sowie der detaillierten Kursgliederung voneinander unterscheiden. Insbesondere die Art der Beispiele und Übungen lässt der Lehrplan komplett offen.

Lizensierte Schulungen zu FLEX tragen zur Zulassung zur abschließenden Advanced-Level-Zertifizierungsprüfung folgende Punkte (Credit Points) bei:

Methodische Kompetenz: 10 Punkte
Technische Kompetenz: 20 Punkte
Kommunikative Kompetenz: 00 Punkte

1.3 Voraussetzungen für das Modul FLEX

Teilnehmer **sollten** folgende Kenntnisse und/oder Erfahrung mitbringen:

- Grundlagen der Beschreibung von Architekturen mit Hilfe verschiedener Sichten, übergreifender Konzepte, Entwurfsentscheidungen, Randbedingungen etc., wie es im CPSA-F (Foundation Level) vermittelt wird.
- Erfahrung mit der Implementierung und Architektur in agilen Projekten.
- Erfahrungen aus der Entwicklung und Architektur klassischer Systeme mit den typischen Herausforderungen.

Hilfreich für das Verständnis einiger Konzepte sind darüber hinaus:

- Verteilte Systeme
  - Probleme und Herausforderungen bei der Implementierung verteilter Systeme
  - Typische verteilte Algorithmen
  - Internet-Protokolle
- Kenntnisse über Modularisierungen
  - Fachliche Modularisierung
  - Technische Umsetzungen wie Pakete oder Bibiliotheken
- Klassische Betriebs- und Deployment-Prozesse

1.4 Gliederung des Lehrplans für FLEX

Die einzelnen Abschnitte des Lehrplans sind gemäß folgender Gliederung beschrieben:

- **Begriffe/Konzepte**: Wesentliche Kernbegriffe dieses Themas.
- **Unterrichts-/Übungszeit**: Legt die Unterrichts- und Übungszeit fest, die für dieses Thema bzw. dessen Übung in einer akkreditierten Schulung mindestens aufgewendet werden muss.
- **Lernziele**: Beschreibt die zu vermittelnden Inhalte inklusive ihrer Kernbegriffe und -konzepte.

Dieser Abschnitt skizziert damit auch die zu erwerbenden Kenntnisse in entsprechenden Schulungen. Die Lernziele werden differenziert in folgende Kategorien bzw. Unterkapitel:

- Was sollen die Teilnehmer **können**? Diese Inhalte sollen die Teilnehmer nach der Schulung selbständig anwenden können. Innerhalb der Schulung werden diese Inhalte durch Übungen abgedeckt und sind Bestandteil der Modulprüfung FLEX und/oder der Abschlussprüfung des iSAQB-Advanced-Levels.
- Was sollen die Teilnehmer **verstehen**? Diese Inhalte können in der Modulprüfung FLEX geprüft werden.
- Was sollen die Teilnehmer **kennen**? Diese Inhalte (Begriffe, Konzepte, Methoden, Praktiken oder Ähnliches) können das Verständnis unterstützen oder das Thema motivieren. Diese Inhalte sind nicht Bestandteil der Prüfungen, werden in Schulungen thematisiert, aber nicht notwendigerweise ausführlich unterrichtet.

1.5 Ergänzende Informationen, Begriffe, Übersetzungen

Soweit für das Verständnis des Lehrplans erforderlich, haben wir Fachbegriffe ins iSAQB-Glossar aufgenommen, definiert und bei Bedarf durch die Übersetzungen der Originalliteratur ergänzt.

2 Motivation

Dauer: 120 Min	Übungszeit: Keine

2.1 Begriffe und Konzepte

Verfügbarkeit, Zuverlässigkeit, Time-to-Market, Flexibilität, Vorhersagbarkeit, Reproduzierbarkeit, Homogenisierung der Stages, Internet/Web-Scale, verteilte Systeme, Parallelisierbarkeit der Feature-Entwicklung, Evolution der Architektur (Build for Replacement), Heterogenität, Automatisierbarkeit.

2.2 Lernziele

2.2.1 Was sollen die Teilnehmer können?

- Architekturen können auf unterschiedliche Qualitätsziele hin optimiert werden. In diesem Modul lernen die Teilnehmer, wie sie flexible Architekturen erstellen, die schnelles Deployment und damit schnelles Feedback aus der Anwendung des Systems erlauben.
- Sie haben die Treiber für die Architektur-Typen verstanden, die in diesem Lehrplanmodul vermittelt werden, welche Konsequenzen die Treiber für die Architekturen haben und wie die Wechselwirkung der Architekturen mit Organisation, Prozessen und Technologien ist.
- Sie haben die Tradeoffs der vorgestellten Architektur-Typen (mindestens Microservices, Self Contained Systems und Deployment Monolithen) verstanden und können diese sowohl vermitteln als auch im Rahmen konkreter Projekte/Systementwicklungen anwenden, um angemessene Architekturentscheidungen zu treffen.

2.2.2 Was sollen die Teilnehmer verstehen?

- Auf die Fähigkeit, neue Features schnell in Produktion bringen zu können, hat die Architektur entscheidenden Einfluss.
- Abhängigkeiten zwischen Komponenten, die von unterschiedlichen Entwicklungsteams verantwortet werden, beeinflussen die Dauer, bis Software in Produktion gebracht werden kann, weil sie die Kommunikationsaufwände erhöhen und sich Verzögerungen fortpflanzen.
- Die Automatisierung von Aktivitäten (wie z. B. Test- und Deployment-Prozesse) erhöht die Reproduzierbarkeit, Vorhersagbarkeit und Ergebnisqualität dieser Prozesse. Das kann zu einer Verbesserung des gesamten Entwicklungsprozesses führen.
- Die Vereinheitlichung der verschiedenen Umgebungen (z. B. Entwicklung, Test, QA, Produktion) reduziert das Risiko von spät entdeckten und (in anderen Umgebungen) nicht reproduzierbaren Fehlern aufgrund unterschiedlicher Konfigurationen.
- Die Vereinheitlichung und Automatisierung sind wesentliche Aspekte von Continuous Delivery.
- Continuous Integration ist eine Voraussetzung für Continuous Delivery.
- Eine geeignete Architektur ist die Voraussetzung für eine Parallelisierbarkeit der Entwicklung sowie die unabhängige Inbetriebnahme von eigenständigen Bausteinen. Das können, müssen aber nicht „Services“ sein.
- Einige Änderungsszenarien lassen sich leichter in monolithischen Architekturen umsetzen. Andere Änderungsszenarien lassen sich leichter in verteilten Service-Architekturen umsetzen. Beide Ansätze können kombiniert werden.
- Es gibt unterschiedliche Arten der Isolation mit unterschiedlichen Vorteilen. Beispielsweise kann der Ausfall auf eine Komponente begrenzt werden oder Änderungen können auf eine Komponente begrenzt werden.
- Bestimmte Arten der Isolation sind zwischen Prozessen mit Remote-Kommunikation deutlich einfacher umzusetzen.
- Remote-Kommunikation hat aber Nachteile – z. B. viele neue Fehlerquellen.

2.2.3 Was sollen die Teilnehmer kennen?

- Gesetz von Conway
- Partitionierbarkeit als Qualitätsmerkmal
- Durchlaufzeiten durch die IT-Wertschöpfungskette als Wettbewerbsfaktor
- Aufbau einer Continuous-Delivery-Pipeline
- Die verschiedenen Test-Phasen in einer Continuous-Delivery-Pipeline

2.3 Referenzen

- Jez Humble, David Farley: Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Addison-Wesley, 2010, ISBN 978-0-32160-191-9
- Eberhard Wolff: Continuous Delivery: Continuous Delivery: Der pragmatische Einstieg, dpunkt, 2014, ISBN 978-3-86490-208-6
- Jez Humble, Barry O'Reilly, Joanne Molesky: Lean Enterprise: Adopting Continuous Delivery, DevOps, and Lean Startup at Scale, O'Reilly 2014, ISBN 978-1-44936-842-5

3 Modularisierung

Dauer: 120 Min	Übungszeit: 30 Min

3.1 Begriffe und Konzepte

- Motivation für die Dekomposition in kleinere Systeme
- Unterschiedliche Arten von Modularisierung, Kopplung
- Systemgrenzen als Mittel für Isolation
- Hierarchische Struktur
- Anwendung, Applikation, Self-Contained System, Microservice
- Domain-Driven Design-Konzepte und „Strategic Design", Bounded Contexts

3.2 Lernziele

3.2.1 Was sollen die Teilnehmer können?

- Teilnehmer können für eine gegebene Aufgabenstellung eine Zerlegung in einzelne Bausteine entwerfen.
- Die Teilnehmer sollen die Kommunikationsstruktur der Organisation beim Festlegen der Modulgrenzen berücksichtigen (Gesetz von Conway).
- Die Teilnehmer sollen technische Modularisierungskonzepte projektspezifisch bewerten und auswählen können.
- Die Teilnehmer sollen die Beziehungen zwischen Modulen sowie zwischen Modulen und Subdomänen veranschaulichen und analysieren können (Context Mapping).
- Die Teilnehmer können die Konsequenzen verschiedener Modularisierungsstrategien bewerten und den mit der Modularisierung verbundenen Aufwand dem zu erwartenden Nutzen gegenüberstellen.
- Die Teilnehmer können die Auswirkungen der Modularisierungsstrategie auf die Autonomie von Bausteinen zur Entwicklungszeit und zur Laufzeit beurteilen.
- Die Teilnehmer können einen Plan zur Aufteilung eines Deployment-Monolithen in kleine Services aufstellen.
- Die Teilnehmer können ein Konzept erarbeiten, um ein System aus Services aufzubauen.
- Die Teilnehmer können eine geeignete Modularisierung und eine geeignete Granularität der Modularisierung wählen – abhängig von der Organisation und den Qualitätszielen.

3.2.2 Was sollen die Teilnehmer verstehen?

- Teilnehmer verstehen, dass jede Art von Bausteinen neben einer griffigen Bezeichnung eine Beschreibung benötigt,
  - was Bausteine dieser Art ausmacht,
  - wie ein solcher Baustein zur Laufzeit integriert wird,
  - wie ein solcher Baustein (im Sinne des Build-Systems) gebaut wird,
  - wie ein solcher Baustein deployt wird,
  - wie ein solcher Baustein getestet wird,
  - wie ein solcher Baustein skaliert wird.
- Teilnehmer verstehen, dass eine Integrationsstrategie darüber entscheidet, ob eine Abhängigkeit
  - erst zur Laufzeit entsteht,
  - zur Entwicklungszeit entsteht, oder
  - beim Deployment entsteht.

- Modularisierung hilft Ziele wie Parallelisierung der Entwicklung, unabhängiges Deployment/Austauschbarkeit zur Laufzeit, Rebuild/Reuse von Modulen und leichtere Verständlichkeit des Gesamtsystems zu erreichen.
- Daher ist beispielsweise Continuous Delivery und die Automatisierung von Test und Deployment ein wichtiger Einfluss auf die Modularisierung.
- Modularisierung bezeichnet Dekomposition eines Systems in kleinere Teile. Diese Teile nach der Dekomposition wieder zu integrieren, verursacht organisatorische und technische Aufwände. Diese Aufwände müssen durch die Vorteile, die durch die Modularisierung erreicht werden, mehr als ausgeglichen werden.
- Teilnehmer verstehen, dass zur Erreichung von höherer Autonomie der Entwicklungsteams ein Komponentenschnitt besser entlang fachlicher Grenzen anstatt entlang technischer Grenzen erfolgt.
- Je nach gewählter Modularisierungstechnologie besteht Kopplung auf unterschiedlichen Ebenen:
    - Sourcecode (Modularisierung mit Dateien, Klassen, Packages, Namensräumen etc.)
    - Kompilat (Modularisierung mit JARs, Bibliotheken, DLLs etc.)
    - Laufzeitumgebung (Betriebssystem, virtuelle Maschine oder Container)
    - Netzwerkprotokoll (Verteilung auf verschiedene Prozesse)
- Eine Kopplung auf Ebene des Sourcecodes erfordert sehr enge Kooperation sowie gemeinsames SCM. Eine Kopplung auf Ebene des Kompilats bedeutet, dass die Bausteine in der Regel gemeinsam deployt werden müssen. Nur eine Verteilung auf unterschiedliche Anwendungen/Prozesse ist praktikabel im Hinblick auf ein unabhängiges Deployment.
- Teilnehmer verstehen, dass eine vollständige Isolation zwischen Bausteinen nur durch eine Trennung in allen Phasen (Entwicklung, Deployment und Laufzeit) gewährleistet werden kann. Ist das nicht der Fall, können unerwünschte Abhängigkeiten nicht ausgeschlossen werden. Gleichzeitig verstehen die Teilnehmer auch, dass es sinnvoll sein kann, aus Gründen wie effizienter Ressourcennutzung oder Komplexitätsreduktion auf eine vollständige Isolation zu verzichten.
- Teilnehmer verstehen, dass bei der Verteilung auf unterschiedliche Prozesse manche Abhängigkeiten nicht mehr in der Implementierung existieren, sondern erst zur Laufzeit entstehen. Dadurch steigen die Anforderungen an die Überwachung dieser Schnittstellen.
- Microservices sind unabhängige Deployment-Einheiten und damit auch unabhängige Prozesse, die ihre Funktionen über leichtgewichtige Protokolle exponieren, aber auch ein UI haben können. Für die Implementierung jedes einzelnen Microservices können unterschiedliche Technologieentscheidungen getroffen werden.
- Ein Self-Contained System (SCS) stellt ein fachlich eigenständiges System dar. Es beinhaltet üblicherweise UI und Persistenz. Es kann aus mehreren Microservices bestehen. Fachlich deckt ein SCS meist einen Bounded Context ab.
- Der Modulschnitt kann entlang fachlicher oder technischer Grenzen erfolgen. In den meisten Fällen empfiehlt sich ein fachlicher Schnitt, da sich so fachliche Anforderungen klarer einem Modul zuordnen lassen und somit nicht mehrere Module für die Umsetzung einer fachlichen Anforderung angepasst werden müssen. Dabei kann jedes Modul sein eigenes Domänenmodell im Sinne eines Bounded Context und damit unterschiedliche Sichten auf ein Geschäftsobjekt mit eigenen Daten haben.
- Transaktionale Konsistenz lässt sich über Prozessgrenzen hinweg nur über zusätzliche Mechanismen erreichen. Wird ein System in mehrere Prozesse aufgeteilt, so stellt die Modulgrenze daher häufig auch die Grenze für transaktionale Konsistenz dar. Daher muss ein DDD-Aggregat in einem Modul verwaltet werden.
- Teilnehmer verstehen, welche Modularisierungskonzepte nicht nur für Transaktions-, sondern auch für Batch- und Datenfluss-orientierte Systeme genutzt werden können.

- Unterschiedliche Grade an Vorgaben für die Entwicklung eines Bausteins können sinnvoll sein. Einige Vorgaben sollten besser übergeordnet allgemein für die Integration mit anderen Bausteinen dieser Art gelten. Die übergreifenden Entscheidungen, die alle Systeme beeinflussen, können eine Makro-Architektur bilden - dazu zählen beispielsweise die Kommunikationsprotokolle oder Betriebsstandards. Mikro-Architektur kann die Architektur eines einzelnen Systems sein. Es ist weitgehend unabhängig von anderen Systemen. Zu große Einschränkungen auf Ebene der Makro-Architektur führen dazu, dass die Architektur insgesamt auf weniger Probleme angewendet werden kann.

3.2.3 Was sollen die Teilnehmer kennen?

- Die Teilnehmer sollen verschiedene technische Modularisierungsmöglichkeiten kennen: z. B. Dateien, JARs, OSGI Bundles, Prozesse, Microservices, SCS.
- Die Teilnehmer sollen verschiedene technische Modularisierungsmöglichkeiten kennen: Sourcecode-Dateien, Bibliotheken, Frameworks, Plugins, Anwendungen, Prozesse, Microservices, Self-Contained Systems.
- Die Teilnehmer sollen folgende Begriffe aus dem Domain-Driven Design kennen: Aggregate Root, Context Mapping, Bounded Contexts und Beziehungen dazwischen (z. B. Anti-Corruption Layer).
- Die Teilnehmer sollen "The Twelve-Factor App" kennen.
- Die Teilnehmer sollen das Gesetz von Conway kennen.

3.3 Referenzen

- Eric Evans: Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley Professional, 2003
- http://12factor.net/
- Sam Newmann: Building Microservices: Designung Fine-Grained Systems, O'Reilly Media, 2015
- Eberhard Wolff: Microservices - Grundlagen flexibler Software Architekturen, dpunkt, 2015
- http://martinfowler.com/articles/microservices.html

4 Integration

Dauer: 90 Min	Übungszeit: 30 Min

4.1 Begriffe und Konzepte

Frontend Integration, Legacy Systeme, Authentifizierung, Autorisierung, (lose) Kopplung, Skalierbarkeit, Messaging Patterns, Domain Events, dezentrale Datenhaltung.

4.2 Lernziele

4.2.1 Was sollen die Teilnehmer können?

- Die Teilnehmer sollen eine Integrationsstrategie wählen, die auf das jeweilige Problem am besten passt. Das kann beispielsweise eine Frontend-Integration, eine Integration über RPC-Mechanismen, mit Message-orientierter Middleware, mit REST oder über die Replikation von Daten sein.
- Die Teilnehmer sollen einen geeigneten Ansatz für das Umsetzen von Sicherheit (Autorisierung/Authentifizierung) in einem verteilten System konzeptionieren können.
- Anhand dieser Ansätze sollen die Teilnehmer eine Makroarchitektur entwerfen können, die zumindest Kommunikation und Sicherheit abdeckt.
- Für die Integration von Legacy-Systemen muss definiert werden, wie mit alten Datenmodellen umgegangen wird. Dazu kann der Ansatz des Strategic Designs mit wesentlichen Patterns wie beispielsweise Anti-Corruption Layer genutzt werden.
- Die Teilnehmer können abhängig von den Qualitätszielen und dem Wissen des Teams eine geeignete Integration vorschlagen.

4.2.2 Was sollen die Teilnehmer verstehen?

- Die Teilnehmer sollen die Vor- und Nachteile verschiedener Integrationsmechanismen kennen. Dazu zählen Frontend-Integration mit Mash Ups, Integration auf dem Middle Tier und Integration über Datenbanken oder Datenbank-Replikation.
- Die Teilnehmer sollen die Konsequenzen und Einschränkungen verstehen, die sich aus der Integration von Systemen über verschiedenen Technologien und Integrationsmuster z. B. in Bezug auf Sicherheit, Antwortzeit oder Latenz ergeben.
- Die Teilnehmer sollen ein grundlegendes Verständnis für die Umsetzung von Integrationen mit Hilfe von Strategic Design aus dem Domain Driven Design und wesentliche Pattern kennen.
- RPC bezeichnet Mechanismen, um Funktionalität in einem anderen Prozess über Rechnergrenzen hinweg synchron aufzurufen. Dadurch entsteht Kopplung in vielerlei Hinsicht (zeitlich, Datenformat, API). Diese Kopplung hat negative Auswirkungen auf Verfügbarkeit und Antwortzeiten des Systems. REST macht Vorgaben, die diese Kopplung reduzieren können (Hypermedia, standardisierte API). Die zeitliche Kopplung bleibt jedoch grundsätzlich bestehen.
- Bei der Integration mittels Messaging kommunizieren Systeme durch den asynchronen Austausch von Nachrichten. Die Systeme werden somit zeitlich entkoppelt. Technisch wird dies mittels Indirektion über eine Middleware erreicht. Nachrichten können optional persistiert, gefiltert, transformiert etc. werden. Es gibt verschiedene Messaging Patterns wie Request/Reply, Publish/Subscribe oder Broadcast.
- Eine Integration über Daten ermöglicht hohe Autonomie, die allerdings über die Notwendigkeit zur redundanten Datenhaltung und die damit notwendige Synchronisation erkauft wird. Es darf nicht angenommen werden, dass andere Systeme dieselben Schemata nutzen, weil das eine unabhängige Weiterentwicklung der Schemata verhindert. Daher muss für die Integration eine angemessene Transformation vorgesehen werden.

- Bei einer Event-Driven Architecture (EDA) wird RPC vermieden oder reduziert, indem Domain Events publiziert werden. Domain Events beschreiben Zustandsänderungen. Interessierte Systeme können diese Nachrichten verarbeiten (Publish/Subscribe). Dieses Vorgehen hat Auswirkungen darauf, wie der Zustand gespeichert wird. Während bei einer RPC-basierten Integration der Server die Daten speichern muss, liegt diese Verantwortung bei EDA beim Subscriber. Somit entstehen Replikate (dezentrale Datenhaltung). Die Subscriber fungieren dadurch als Cache für den Publisher. Es können weitere Subscriber hinzugefügt werden, ohne den Publisher zu beeinflussen (außer durch Polling). Monitoring der Event-Flüsse ist wichtig.
- Domain Events können via Messaging publiziert werden. Dabei pusht der Publisher die Nachrichten in ein Messaging System. Alternativ können die Nachrichten auch beim Publisher gepollt werden (bspw. Atom/RSS). Beim Einsatz eines Messaging Systems können Subscriber die Nachrichten per Push oder Pull erhalten. Dies hat Auswirkungen auf den Umgang mit Backpressure.

4.2.3 Was sollen die Teilnehmer kennen?

- Typische verteilte Sicherheitsmechanismen wie OAuth oder Kerberos
- Ansätze für die Front-End-Integration
- Techniken für die Integration von Services: REST, RPC, Message-orientierte Middleware
- Herausforderungen bei der Nutzung gemeinsamer Daten
- Datenbank-Replikationsmechanismen mit ETL-Tools oder anderen Ansätzen
- Messaging Patterns (Request/Reply, Publish/Subscribe etc.)
- Messaging Systeme (RabbitMQ, Kafka etc.), Protokolle (AMQP, MQTT, STOMP etc.) und APIs (JMS)

4.3 Referenzen

- Eric Evans: Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, 2003, ISBN 978-0-32112-521-7
- http://oauth.net/
- Gregor Hohpe, Bobby Woolf: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley, 2003, ISBN 978-0-32120-068-6

5 Installation und Roll Out

Dauer: 90 Min	Übungszeit: 30 Min

5.1 Begriffe und Konzepte

Moderner Betrieb, DevOps, Infrastructure as Code, Configuration Management.

5.2 Lernziele

5.2.1 Was sollen die Teilnehmer können?

- Die Teilnehmer sollen ein Konzept grob skizzieren und verstehen können, wie ein System möglichst einfach automatisiert deployt wird. Sie müssen zwischen den verschiedenen technologischen Ansätzen abwägen können.
- Die Teilnehmer sollen Deployment-Automatisierung konzeptionieren und bewerten können. Beispielsweise müssen sie die Qualität und das Testen dieser Ansätze bewerten können. Sie müssen einen geeigneten Ansatz für ein Projekt-Szenario auswählen können.
- Die Teilnehmer sollen auf Basis des DevOps-Organisationsmodells einen Team-Aufbau skizzieren können.

5.2.2 Was sollen die Teilnehmer verstehen?

- Basis für die Automatisierung eines Deployments sind Virtualisierung oder Cloud mit Infrastructure as a Service (IaaS). Eine leichtgewichtige Alternative sind Linux-Container, wie sie Docker umsetzt.
- Ohne eine Deployment-Automatisierung ist das Deployment einer großen Anzahl von Servern und Services praktisch nicht möglich.
- Moderne Deployment-Werkzeuge ermöglichen es, auf Rechnern Software automatisiert zu installieren. Neben der Anwendung selbst kann dabei auch die vollständige Infrastruktur automatisiert aufgebaut werden. Dabei sind die Installationen idempotent, d. h. sie führen unabhängig vom Ausgangszustand des Systems immer zum gleichen Ergebnis.
- Immutable Server werden grundsätzlich nie geändert. Muss eine neue Version der Software in Betrieb genommen werden, wird der Server komplett neu aufgebaut. Das kann einfacher und zuverlässiger sein, als sich auf idempotente Werkzeuge zu verlassen.
- PaaS (Platform as a Service) stellt eine vollständige Plattform bereit, in die Anwendungen deployt werden können. Da in diesem Fall die Infrastruktur nicht selber aufgebaut wird, ist der Ansatz einfacher, aber auch weniger flexibel.
- Konzepte wie Tests oder Code Reviews sind auch für die Deployment-Automatisierung unabdingbar. Infrastruktur wird zu Code, der denselben Anforderungen wie produktiver Code gerecht werden muss (Infrastructure as Code).
- Um das Deployment zu unterstützen, kann das Ergebnis eines Build-Prozesses Packages für ein Betriebssystem oder gar Images für virtuelle Maschinen sein.
- Die Umgebungen eines Entwicklers sollten idealerweise mit den Umgebungen in Produktion übereinstimmen. Mit modernen Werkzeugen ist es möglich, eine solche Umgebung auf Knopfdruck zu erzeugen und aktuell zu halten.
- Die Komplexität des Deployments wird zu einem weiteren Qualitätsmerkmal des Systems und beeinflusst Architektur-Werkzeuge.
- Durch DevOps wird der Aufbau der Teams anders. Es müssen neben Entwicklung auch Betrieb stärker betrachtet werden. Neben dem Provisioning hat das auch Auswirkungen auf Continuous Delivery (siehe Kapitel „Continuous Delivery“).

5.2.3 Was sollen die Teilnehmer kennen?

- Grundlegende Konzept von moderner Infrastruktur wie IaaS, PaaS und Virtualisierung

- Konzepte von Deployment-Werkzeuge wie Chef, Puppet, Ansible oder Salt
- Organisationsformen für DevOps
- Konzept von Deployments mit Package-Managern oder Linux Containern
- Verschiedene PaaS-Plattformen und ihre Konzepte

5.3 Referenzen

- Gottfried Vossen, Till Haselmann, Thomas Hoeren: Cloud-Computing für Unternehmen: Technische, wirtschaftliche, rechtliche und organisatorische Aspekte, dpunkt, 2012, ISBN 978-3-89864-808-0
- Eberhard Wolff, Stephan Müller, Bernhard Löwenstein: PaaS - Die wichtigsten Java Clouds auf einen Blick, entwickler.press, 2013
- Jez Humble, David Farley: Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Addison-Wesley, 2010, ISBN 978-0-32160-191-9
- Eberhard Wolff: Continuous Delivery: Continuous Delivery: Der pragmatische Einstieg, dpunkt, 2014, ISBN 978-3-86490-208-6

6 Betrieb, Überwachung und Fehleranalyse

Dauer: 90 Min	Übungszeit: 30 Min

6.1 Begriffe und Konzepte

Monitoring, Operations, Logging, Tracing, Metrics.

6.2 Lernziele

6.2.1 Was sollen die Teilnehmer können?

- Die Teilnehmer sollen ein Konzept grob skizzieren und verstehen können, auf dessen Basis ein System überwacht werden kann d. h. den aktuellen Status zu beurteilen, Fehler und Abweichungen vom normalen Betrieb möglichst zu vermeiden oder zumindest so früh wie möglich zu erkennen zu behandeln.
- Dabei können sie abhängig vom konkreten Projekt-Szenario den Fokus im Konzept auf Logging, Monitoring und die dazu notwendigen Daten legen.
- Die Teilnehmer sollen Architekturvorgaben so treffen können, dass der Einsatz geeigneter Werkzeuge bestmöglich unterstützt wird, dabei jedoch angemessen mit Systemressourcen umgegangen wird.

6.2.2 Was sollen die Teilnehmer verstehen?

- Logging und Monitoring kann sowohl fachliche als auch technische Daten enthalten.
- Die richtige Auswahl von Daten ist zentral für ein zuverlässiges und sinnvolles Monitoring und Logging.
- Damit Systeme, insbesondere solche, die sich aus vielen einzelnen Teilsystemen zusammensetzen, betreibbar sind, muss die Unterstützung des Betriebs mit hoher Priorität Bestandteil der Architekturkonzepte sein.
- Damit eine möglichst hohe Transparenz erreicht wird, müssen sehr viele Daten erfasst, aber auch zielgruppengerecht voraggregiert und auswertbar gemacht werden.
- Die Teilnehmer sollen verstehen, welche Informationen sie aus Log-Daten und welche sie (besser) durch Instrumentierung des Codes mit Metrik-Sonden beziehen können.
- Die Teilnehmer sollen verstehen, wie eine typische zentralistische Logdaten-Verwaltung aufgebaut ist und welche Auswirkungen sie auf die Architektur hat.
- Die Teilnehmer sollen verstehen, wie eine typische zentralistische Metriken-Pipeline aufgebaut ist (Erfassen, Sammeln & Samplen, Persistieren, Abfragen, Visualisieren) und welche Auswirkungen sie auf die Architektur hat (Performance-Overhead, Speicherverbrauch,...).
- Die Teilnehmer sollen die unterschiedlichen Möglichkeiten von Logging, Monitoring und einer Operations DB (siehe M. Nygard, Release IT!) verstehen, was man wofür einsetzt und wie man diese Werkzeuge sinnvoll kombiniert.

6.2.3 Was sollen die Teilnehmer kennen?

- Werkzeuge für zentralistische Logdaten-Verwaltung
- Werkzeuge für zentralistische Metriken-Verarbeitung
- Unterscheidung zwischen Geschäfts-, Anwendungs- und Systemmetriken
- Bedeutung wichtiger, werkzeugunabhängiger System- und Anwendungsmetriken

6.3 Referenzen

- Eberhard Wolff: Continuous Delivery: Continuous Delivery: Der pragmatische Einstieg, dpunkt, 2014, ISBN 978-3-86490-208-6

- Michael Nygard: Release It!: Design and Deploy Production-Ready Software, Pragmatic Programmers, 2007, ISBN 978-0-97873-921-8

7 Case Study

Dauer: 90 Min	Übungszeit: 60 Min

Im Rahmen einer Lehrplan-konformen Schulung muss eine Fallstudie die Konzepte praktisch erläutern.

7.1 Begriffe und Konzepte

Die Case Study führt keine neuen Begriffe und Konzepte sein.

7.2 Lernziele

Die Case Study soll keine neuen Lernziele vermitteln, sondern die Themen durch praktische Übungen vertiefen und die Praxis verdeutlichen.

7.3 Referenzen

Keine. Schulungsanbieter sind für die Auswahl und Beschreibung von Beispielen verantwortlich.

8 Ausblick

Dauer: 120 Min	Übungszeit: 0 Min

Der Ausblick stellt fortgeschrittene Themen dar, in die sich Teilnehmer vertiefen können. So erreichen sie ein tieferes Verständnis für die Herausforderungen bei der Umsetzung flexibler Systeme. Außerdem lernen sie weitere Einflussfaktoren auf die Auswahl von Technologien kennen.

8.1 Begriffe und Konzepte

- Konsistenzmodelle: ACID, BASE, Partitionierung, CAP
- Resilience: Resilient Software Design, Stabilität, Verfügbarkeit, Graceful Degradation Circuit Breaker, Bulkhead

8.2 Lernziele

8.2.1 Was sollen die Teilnehmer können?

- Die Teilnehmer sollen verschiedene Konsistenzmodelle kennen. Die Tradeoffs der verschiedenen Konsistenzmodelle sollten sie grundlegend kennen.
- Abhängig von den Anforderungen und Rahmenbedingungen sollen sie entscheiden können, ob traditionelle Stabilitätsansätze hinreichend sind oder ob Resilient Software Design erforderlich ist.

8.2.2 Was sollen die Teilnehmer verstehen?

- Konsistenzmodelle
  - Die Notwendigkeit für ACID Transaktionen ist wesentlich geringer, als häufig angenommen wird.
  - Unterschiedliche Skalierungs-, Verteilungs- und Verfügbarkeitsanforderungen erfordern unterschiedliche Konsistenzmodelle.
  - Das CAP-Theorem beschreibt ein Spektrum, in dem man abhängig von den gegebenen Anforderungen sehr feingranular ein geeignetes Konsistenzmodell wählen kann.
  - BASE-Transaktionen garantieren Konsistenz, sie sind nur nicht unbedingt atomar und isoliert wie ACID-Transaktionen, weshalb vorrübergehend Inkonsistenzen sichtbar werden können.
- Resilience
  - Traditionelle Stabilitätsansätze (Fehlervermeidungsstrategien) auf Infrastrukturebene sind für heutige verteilte, hochvernetzte Systemlandschaften in der Regel nicht mehr hinreichend.
  - Es gibt keine Silver Bullet für Resilient Software Design, d. h. die relevanten Maßnahmen und eingesetzten Muster und Prinzipien hängen von den Anforderungen, den Rahmenbedingungen und den beteiligten Personen ab.

8.2.3 Was sollen die Teilnehmer kennen?

- Konsistenzmodelle
  - Eigenschaften von und Unterschiede zwischen ACID- und BASE-Transaktionen
  - Einige Produktbeispiele aus unterschiedlichen Kategorien (z. B. NoSQL, Konfigurationswerkzeuge, Service Discovery)
  - CAP zur Beschreibung und Erklärung von Konsistenzmodellen
- Resilience
  - Die Formel für Verfügbarkeit und die unterschiedlichen Ansätze, die Verfügbarkeit zu maximieren (Maximierung von MTTF, Minimierung von MTTR)

- Isolation und Latenzüberwachung als sinnvolle Einstiegsprinzipien von Resilient Software Design
- Grundlegende Resilience-Muster wie Bulkhead, Circuit Breaker, Redundanz, Failover

8.3 Referenzen

- Andrew Tanenbaum, Marten van Steen, Distributed Systems – Principles and Paradigms, Prentice Hall, 2nd Edition, 2006
- Leslie Lamport, The Part-Time Parliament, ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169
- Eric Brewer, Towards Robust Distributed Systems, PODC Keynote, July-19-2000
- Mikito Takada, Distributed Systems for Fun and Profit, http://book.mixu.net/distsys/ (Guter Einstieg und Überblick)
- Michael T. Nygard, Release It!, Pragmatic Bookshelf, 2007
- Robert S. Hanmer, Patterns for Fault Tolerant Software, Wiley, 2007
- James Hamilton, On Designing and Deploying Internet-Scale Services, 21st LISA Conference 2007

9 Quellen und Referenzen zu Informationssysteme für agile Umgebungen

Dieser Abschnitt enthält Quellenangaben, die ganz oder teilweise im Curriculum referenziert werden.

B

Eric Brewer, Towards Robust Distributed Systems, PODC Keynote, July-19-2000

E

Eric Evans: Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley Professional, 2003

F

Martin Fowler: Microservices, http://martinfowler.com/articles/microservices.html

H

James Hamilton, On Designing and Deploying Internet-Scale Services, 21st LISA Conference 2007

Robert S. Hanmer, Patterns for Fault Tolerant Software, Wiley, 2007

Gregor Hohpe, Bobby Woolf: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley, 2003, ISBN 978-0-32120-068-6

Jez Humble, David Farley: Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Addison-Wesley, 2010, ISBN 978-0-32160-191-9

Jez Humble, Barry O'Reilly, Joanne Molesky: Lean Enterprise: Adopting Continuous Delivery, DevOps, and Lean Startup at Scale, O'Reilly 2014, ISBN 978-1-44936-842-5

L

Leslie Lamport, The Part-Time Parliament, ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169

N

Sam Newmann: Building Microservices: Designung Fine-Grained Systems, O'Reilly Media, 2015

Michael T. Nygard, Release It!, Pragmatic Bookshelf, 2007

O

OAuth: http://oauth.net/

T

Mikito Takada, Distributed Systems for Fun and Profit, http://book.mixu.net/distsys/ (Guter Einstieg und Überblick)

Andrew Tanenbaum, Marten van Steen, Distributed Systems – Principles and Paradigms, Prentice Hall, 2nd Edition, 2006

V

Gottfried Vossen, Till Haselmann, Thomas Hoeren: Cloud-Computing für Unternehmen: Technische, wirtschaftliche, rechtliche und organisatorische Aspekte, dpunkt, 2012, ISBN 978-3-89864-808-0

W

Eberhard Wolff: Microservices - Grundlagen flexibler Software Architekturen, dpunkt, 2015

Eberhard Wolff: Continuous Delivery: Continuous Delivery: Der pragmatische Einstieg, dpunkt, 2014, ISBN 978-3-86490-208-6

Eberhard Wolff, Stephan Müller, Bernhard Löwenstein: PaaS - Die wichtigsten Java Clouds auf einen Blick, entwickler.press, 2013