Kapitel 7 Case Study Flexinale

albion.eu www.tectrain.ch www.accso.de

< Kapitel 6 Betrieb, Ueberwachung und Fehleranalyse Kapitel 8 Ausblick >

Kapitel 7 Case Study "Flexinale"

FLEX Lehrplan

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.

Inhalte

(A) Was ist die Flexinale? Überblick und Anforderungen

1. Flexinale in a Nutshell

Flexinale ist eine Beispiel-Anwendung für Organisation, Durchführung und Ticketverkauf eines Filmfestivals. Flexinale orientiert sich an einem echten, realen Kundenproblem (mit anderer Fachlichkeit, aber ähnlichen Nutzergruppen, -szenarien und sehr großen Mengengerüsten von >300k Nutzern und >2 Mio Anzahl Haupt-Entitäten pro Jahr, bei teils sehr großen Lastspitzen verursacht durch Algorithmik und Nutzer-Requests).

Der Name ist ein Kunstwort aus dem "FLEX"-Advanced Level Modul und dem größten deutschen Filmfestival, der jährlich stattfindenden Berlinale.

Nachstehende Abbildung gibt einen Überblick über die Flexinale.

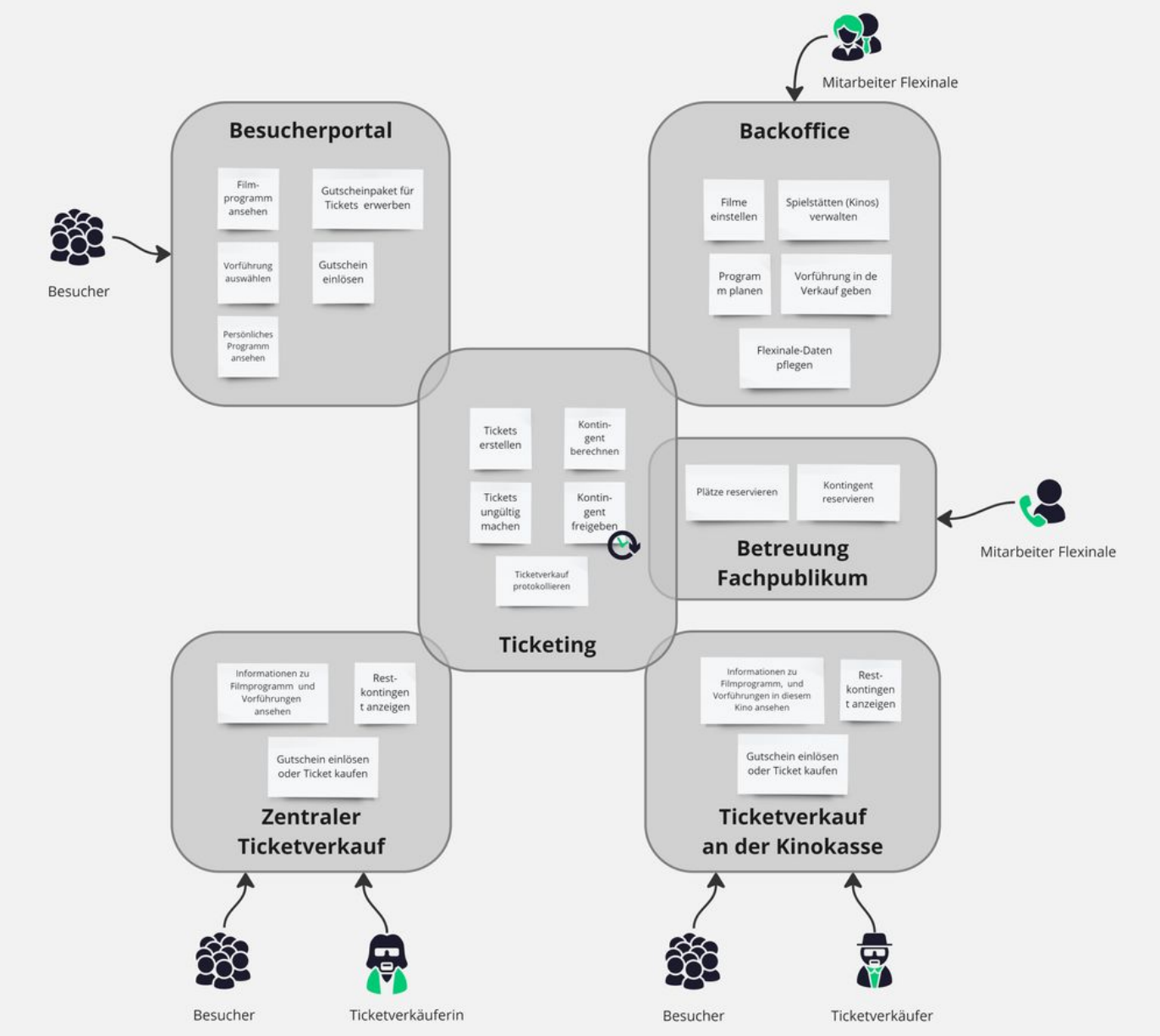


Abbildung: Überblick über die Flexinale
(Quelle: selbst erstellt)

Die Funktionalität wird im Detail ab Abschnitt 3 beschrieben.

Die vier Flexinale-Varianten implementieren nicht alle Funktionalitäten vollständig. Die folgenden Abbildungen zeigen, was in der jeweilige Variante implementiert ist.

Die Flexinale-Anwendung ist in vier Varianten implementiert:

- als Monolith
- als Modulith 1 mit Onion Architektur
- als Modulith 2 mit Fachlichen Komponenten
- als verteilte, service-orientierte und event-basierte Anwendungen (Besucherportal, Backoffice, Ticketing), im Folgenden: Distributed

Monolith: (Funktional identisch mit Modulith-1 und Modulith-2)

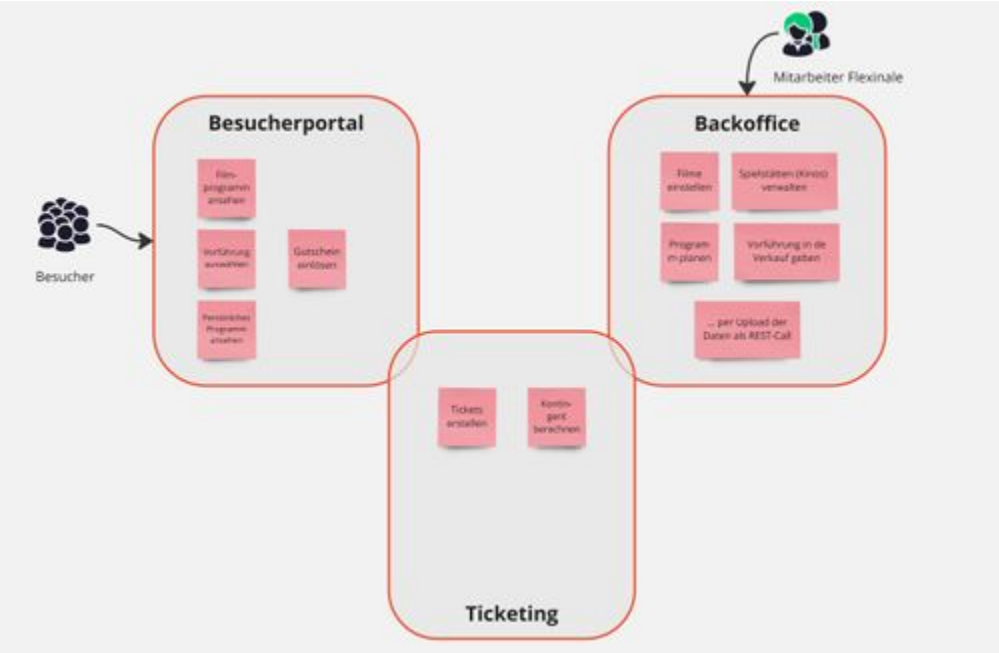


Abbildung: In der Variante "Monolith" implementierte Funktionalität
(Quelle: selbst erstellt)

Modulith-1: (Funktional identisch mit Monolith und Modulith-2)

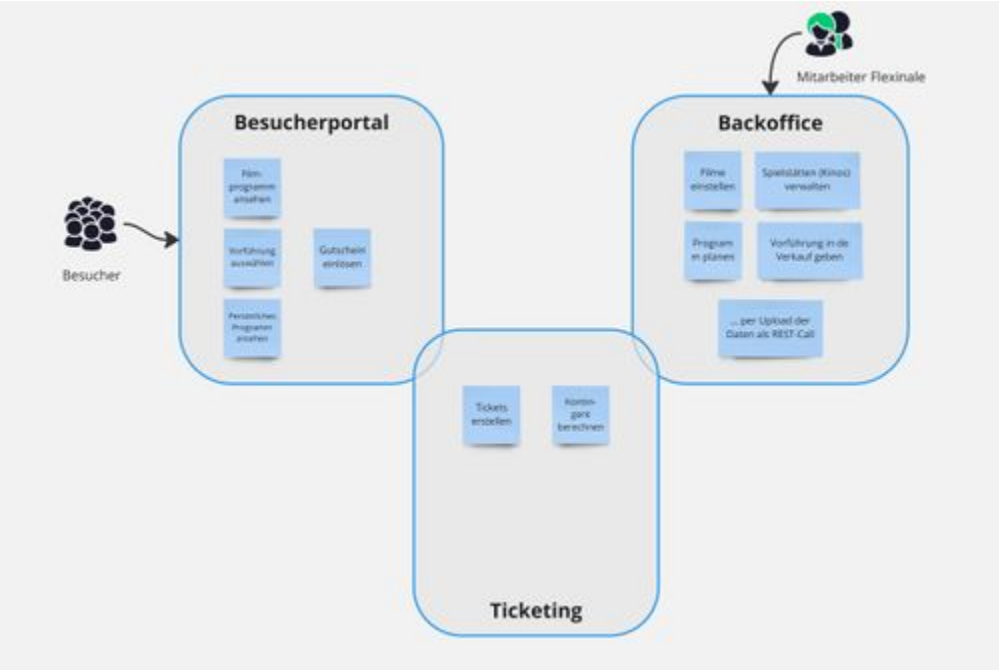


Abbildung: In der Variante "Modulith-1" implementierte Funktionalität
(Quelle: selbst erstellt)

Modulith-2: (Funktional identisch mit Monolith und Modulith-1)

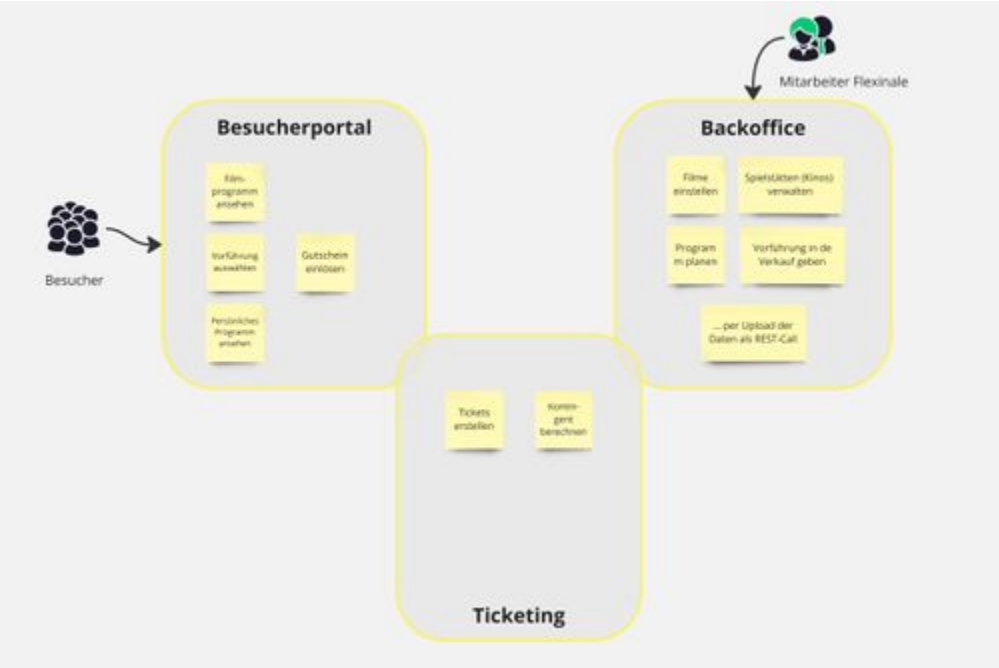


Abbildung: In der Variante "Modulith-2" implementierte Funktionalität
(Quelle: selbst erstellt)

Distributed: (Zusätzliche Funktionalität ggü. Monolith, Modulith-1 und Modulith-2: "Ticket ungültig machen")

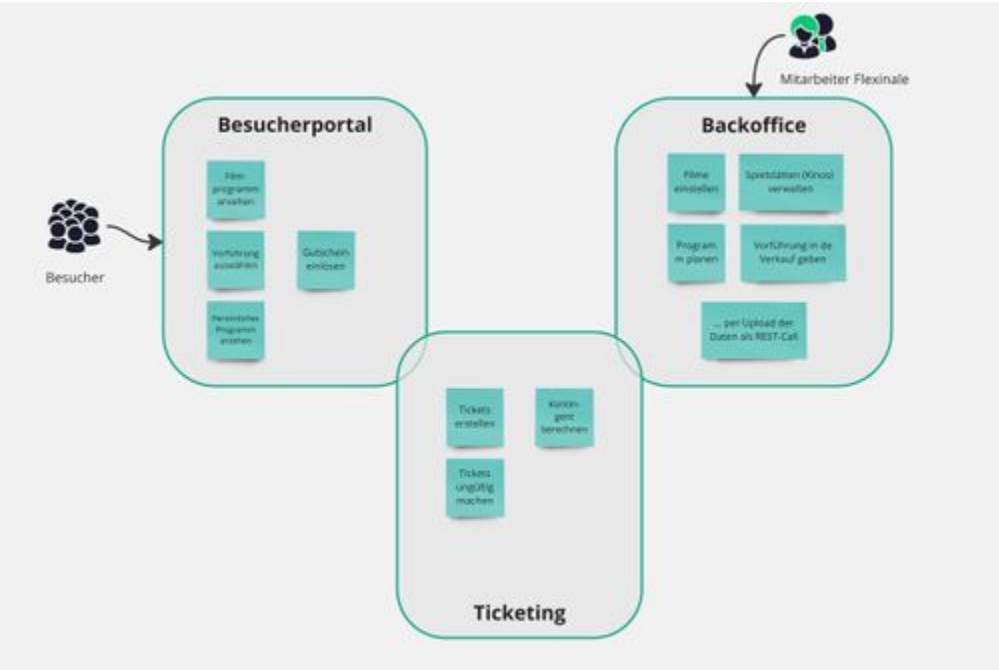


Abbildung: In der Variante "Distributed" implementierte Funktionalität
(Quelle: selbst erstellt)

2. Mengengerüste und Qualitätsanforderungen

Die Qualitätsanforderungen an die Flexinale orientieren sich an realen Mengengerüsten und Nutzungsszenarien.

Mengengerüste
 	
orientieren sich am Berlinale-Filmfest
	Reale Zahlen aus der Berlinale:
	
	1. Mehr als 400 Filme sollen auf der Berlinale 2023 gezeigt werden.
	2. Die Filme werden häufig mehrfach gezeigt, so dass von weit über 1.000 Vorführungen auszugehen ist.
	3. Es gibt 20 - 30 Kinos / Spielstätten mit 1 - 10 Sälen
	4. Aus der Bilanz der Berlinale 2020
	   a. 330.000 verkaufte Tickets
	   b. Schätzungsweise ca. 30.000 "Named User" im Portal (Grundlage der Schätzung: Jeder Nutzer kauft im Schnitt ca. 10 Tickets)
	   c. bis zu 500.000 Vorführungsbesuche (auch Fachpublikum).
	   d. 22.000 akkreditierte Fachbesucher

Last	Lastspitzen gibt es v.a. kurz vor und während der Flexinale beim Abruf von Informationen zu Filmen und Vorführungen, sowie beim Ticketkauf.
Antwortzeiten	Die Anwendung soll auch unter hoher Last noch responsiv sein. 1. Informationsseiten (zu Filmen, Spielstätten) sollen im Sekundenbereich antworten. 2. Beim Ticketkauf dürfen die Antwortzeiten länger sein.

Sicherheit	Jeder Zugriff auf die Anwendung von außen muss geschützt sein (User, Password).
	
	1. Für das Besucherportal, insbesondere zum Einlösen von Gutscheinen, sowie für Aktionen im Backoffice (hier: Datenpflege) ist eine Authentifizierung erforderlich (user, password).
	2. Es gibt ein Rollenkonzept:
	   a. Besucher benötigen die Rolle BESUCHER.
	   b. Die Datenpflege im Backoffice benötigt die Rolle ADMIN

Nachvollziehbarkeit	Alle Ticket-Käufe müssen über Transaktionen nachvollziehbar sein: 1. Kauf, d.h. hier das Einlösen von Gutscheinen. 2. Fehlgeschlagener Kauf (wenn z.B. das Kontingent ausgeschöpft ist) 3. Ungültig gewordene Tickets (nur verteilte Anwendung)
Mandantenfähigkeit	Die Flexinale bedient verschiedene Filmfeste. Aktuell ist in der Software keine Mandanten-ID vorgesehen, Mandantenfähigkeit kann aber recht einfach durch getrennte Installationen erreicht werden.
Internationalisierung	Die Flexinale bedient Filmfeste mit großem internationalem Publikum. Daher muss die Flexinable internationalisierbar sein: Mindestens deutsch- und eine englischsprachige Version muss es geben.

3. Besucherportal

		Hauptentitäten, Aggregate	Implementiert?
Login	Besucher:innen können das Besucherportal nur mit einem Login benutzen.	Besucher	✅
Filmprogramm ansehen	Besucher:innen erhalten Informationen, welche Filme es gibt, wann und wo sie gezeigt werden, sowie zu den jeweiligen Kinos (Spielstätten), in denen Filme gezeigt werden.	Nur lesend: Film, Kino, Vorführung	✅
Vorführung auswählen	Zu jedem Film können sich die Besucher:innen anschauen welche Vorführungen es gibt und eine auswählen, die sie besuchen möchten.		✅
Gutscheinpaket für Tickets erwerben	Um Tickets zu erwerben, muss ein Besucher zunächst ein Gutscheinpaket für Tickets erwerben. Die Gutscheine löst er dann gegen Tickets ein. (Dieses Vorgehen wurde für diese Case Study gewählt, um keinen Bezahlprozess implementieren zu müssen. Der Gutscheinkauf selber ist daher ebenfalls nicht implementiert.)	Gutscheinpaket	
Ticketkauf - Gutscheine online einlösen	Besucher:innen können für eine ausgewählte Vorführung Gutscheine einlösen. Die Zahl der pro Vorführung einlösbaren Gutscheine ist begrenzt. Um Besuchern die Planung zu erleichtern und insbesondere damit sie nicht versehentlich Tickets für zwei zeitgleich stattfindende Vorführungen erwerben, gibt es folgende Komfort-Funktionen: Ein/e Besucher:in kann nur Gutscheine für Vorführungen einlösen ... 1. für die sie noch keine Tickets hat, 2. die zeitlich nicht mit einer anderen Vorstellung überlappt, für die sie bereits Tickets hat. Bei der zeitlichen Überlappung ist außerdem ein Puffer eingerechnet für eine Mindestzeit zwischen zwei Vorführungen (um z.B. die Spielstätte zu wechseln). Das Aggregate "Ticketbundle" kennt alle Tickets des Besuchers und ist daher notwendig, um diese Komfortfunktionen bereitzustellen.	Gutschein Aggregate: Ticketbundle - alle Tickets eines Besuchers	✅
Persönliches Programm ansehen	Die Vorführungen, für die Tickets erworben wurden, sowie die Anzahl der für die Vorführung erworbenen Tickets werden angezeigt. **Nur in Distributed:** Tickets, die ihre Gültigkeit verloren haben, werden ebenfalls angezeigt. Gründe, dass ein Ticket seine Gültigkeit verliert, können z.B. sein, dass eine Vorführung abgesagt wird, oder allgemein Änderungen an Daten Film, Kino / Kinosaal / Vorführung (s.u. unter Backoffice, "Änderungen an den Daten Film,..."		✅

4. Backoffice

		Hauptentitäten, Aggregate	Implementiert?
Flexinale-Daten pflegen	Daten zum Festival wie Zeitraum, Ort	Metadaten zur Instanz des Festivals (z.B. Zeiten, Tage)	
Filme einstellen	Die Filme, die auf der Flexinale gezeigt werden, erstellen und pflegen: Name, IMBD-Link, Dauer (in Stunden)	Film	✅
Ausstrahlungsbedingungen verwalten	Verwalten der (recthlichen) Bedingungen, unter denen ein Film ausgestrahlt werden kann: 1. Wann und wie lange darf ein Film ausgestrahlt werden? 2. FSK-Bedingungen 3. Wie groß dürfen die Spielstätten (Anzahl der Plätze) maximal sein?		
Spielstätten (Kinos) verwalten	Die Spielstätten der Flexinale pflegen: Die Kinos mit ihren Adressen und Kontaktdaten, sowie ihren für die Flexinale zur Verfügung stehenden Kinosäle und ihr Platzangebot.	Kino Kinosaal	✅
Programm planen	Die Vorführungen der Filme planen, also planen, welcher Fiilm wann und in welchem Kino gezeigt wird.	Vorführung	✅
UI für die Datenpflege im Backoffice	Die Datenpflege (Film, Kino / Kinosaal, Vorführung) ist in der Flexinale als REST-Controller implementiert. Per HTTP-POST-Request wird ein Excel-Sheet der Anwendung übergeben, welches die Daten enthält.		✅
CRUD-Funktionen für die Datenpflege im Backoffice	1. Im Monolith / Modulith 1 / Modulith 2 können nur Daten hinzugefügt werden. 2. In Distributed können darüber hinaus Daten aktualisiert werden (sind versioniert). Das Löschen von Daten ist bewusst nicht möglich.		✅
Änderungen an den Daten Film, Kino, Kinosaal oder Vorführung und Tickets	Wenn die Daten Film, Kino, Kinosaal oder Vorführung geändert werden, werden die Tickets aller Vorführungen, die von einer der Änderungen betroffen sind, ungültig. Das heißt: 1. Änderungen an einem Film: Tickets für alle Vorführungen, die diesen Film zeigen, werden ungültig. 2. Änderungen an einem Kino oder Kinosaal: Tickets für alle Vorführungen, die in diesem Kino oder Kinosaal stattfinden, werden ungültig. 3. Änderungen an einer Vorführung: Tickets für diese Vorführung werden ungültig.		✅ (nur Distributed)

5. Betreuung Fachpublikum

Diese Funktionalität ist nicht implementiert.

		Hauptentitäten, Aggregate	Implementiert?
Plätze reservieren	In jeder Vorführung können Plätze für die Filmcrew und andere Mitwirkende reserviert werden.	Reservierung	
Kontingent reservieren	In jeder Vorführung an ein Kontingent an Tickets für ein Fachpublikum, Presse, Fernsehen o.ä. reserviert werden. Die nicht in Anspruch genommenen Tickets gehen kurz vor Beginn der Vorführung in den Verkauf in der Spielstätte.	Kontingent	

6. Ticketing

		Hauptentitäten, Aggregate	Implementiert?
Tickets erstellen	Wenn ein/e Besucher:in einen oder mehrere Gutscheine für eine Vorführung einlöst, werden entsprechend Tickets für sie erstellt. Dazu wird zunächst geprüft, ob das Restkontingent für die Vorführung und für den Verkaufskanal (Online, zentraler Verkauf, an der Kinokasse) noch ausreicht. Nur wenn das vorhandene Kontingent ausreicht für die gewünschte Anzahl an Tickets, werden diese erstellt.	Ticket Kontingent	✅ (nur für Verkaufskanal online)
Kontingent berechnen	Das Kontingent zu einer Vorführung ist die Gesamtzahl der Tickets, die für diese Vorführung verkauft werden können. Es teilt sich auf in 1. Kontingent für den Online-Verkauf 2. Kontingent für den zentraler Verkauf 3. Kontingent für den Verkauf an der Kinokasse der Spielstätte Die Aufteilung zwischen den drei Verkaufsarten ist durch eine (anwendungsweit) konfigurierbare Quote für den Online-Verkauf festgelegt. Das Kontingent für den zentraler Verkauf und den Verkauf an der Kinokasse der Spielstätte ist jeweils die Hälfte des verbliebenen Restes nach Abzug des Online-Kontingentes. Das Kontingent ist die Anzahl der Plätze eines Kinosaals (eigentlich abzüglich der Reservierungen, s.u. Diese sind jedoch nicht implementiert). Das Kontingent wird (neu) berechnet, 1. wenn eine Vorführung in den Verkauf gegeben wird. 2. wenn Tickets für diese Vorführung verkauft werden. 3. wenn eine Vorführung geändert wird.	Kontingent	✅
Kontingent freigeben	s. Betreuung Fachpublikum - Kontingent reservieren: Tickets aus dem für ein Fachpublikum reservierten Kontingent, die nicht in Anspruch genommen wurden, werden automatisch zeitgesteuert 30 Minuten vor einer Vorführung für den freien Verkauf direkt in der Spielstätte freigegeben.	Kontingent	
Ticket ungültig machen	Wenn Daten von Film, Kino, Kinosaal oder Vorführung geändert werden, werden die Tickets für alle Vorführungen, die von einer der Änderungen betroffen sind, ungültig. Das heißt: 1. Änderungen an einem Film: Tickets für alle Vorführungen, die diesen Film zeigen, werden ungültig. 2. Änderungen an einem Kino oder Kinosaal: Tickets für alle Vorführungen, die in diesem Kino oder Kinosaal stattfinden, werden ungültig. 3. Änderungen an einer Vorführung: Tickets für diese Vorführung werden ungültig. Das Kontingent für die betroffene Vorführung wird neu berechnet.	Ticket Kontingent	✅ (nur Distributed)
Ticketverkauf protokollieren	Aus Gründen der Nachvollziehbarkeit wird protokolliert, welche Tickets verkauft wurden oder ungültig gemacht wurden.		

7. Zentrale Verkaufsstelle

Diese Funktionalität ist nicht implementiert.

		Hauptentitäten, Aggregate	Implementiert
Information	Verkäufer und Besucher können Informationen zu Filmprogramm, Spielstätten (Kinos / Kinosälen) und Vorführungen ansehen	Nur lesend: Film, Kino, Vorführung	
Restkontingent anzeigen	Der Verkäufer sieht, wieviele Plätze im zentralen Verkauf für eine Vorführung noch zur Verfügung stehen. Besucher sehen nur eine grobe Zahl in Form einer Ampel (grün - noch reichlich Plätze frei, gelb - nur noch wenige Plätze frei, rot - ausverkauft)	Nur lesend: Kontingent zentral	
Gutschein einlösen oder Ticket kaufen	Besucher können Tickets erwerben, indem sie Gutscheine einlösen.		

8. Spielstätte

Diese Funktionalität ist nicht implementiert.

		Hauptentitäten, Aggregate	Implementiert
Information	Informationen zu Filmprogramm und Vorführungen in dieser Spielstätte	Nur lesend: Film, Kino, Vorführung	
Restkontingent anzeigen	Der Verkäufer sieht, wieviele Plätze im Verkauf an der Kinokasse für eine Vorführung noch zur Verfügung stehen. Besucher sehen nur eine grobe Zahl in Form einer Ampel (grün - noch reichlich Plätze frei, gelb - nur noch wenige Plätze frei, rot - ausverkauft)	Nur lesend: Kontingent Spielstätte	
Gutschein einlösen oder Ticket kaufen	1. Anzeige der noch vorhandenen Tickets für eine Vorführung (des Kontingentes für den Verkauf an der Kinokasse) 2. Tickets verkaufen		

9. Buchhaltung

Diese Funktionalität ist nicht implementiert

		Hauptentitäten, Aggregate	Implementiert
Reporting	Informationen zu Anzahl Filme, Anzahl Vorführungen, Anzahl verkaufter Tickets	Lesend alle Entitäten	
Finanzbuchhaltung	Informationen zu Einnahmen und Ausgaben	Lesend Tickets	

10. Bounded Contexts

Die Abbildung unten zeigt eine Möglichkeit, Bounded Contexts für die Flexinale zu schneiden. Das sind die Treiber für den Schnitt der Bounded Contexts:

Akteure und Nutzungsszenarien	1. Das Besucherportal wird von allen Flexinale-Besuchern genutzt, das Backoffice dagegen nur von Flexinale-Mitarbeitern. 2. Der Ticketverkauf dagegen wird von außen, über verschiedene Verkaufskanäle, angestoßen.
Datenhoheit	Die Hoheit über Entitäten oder Aggregates sollte vollständig innerhalb eines Bounded Contexts liegen
Qualitätsteigenschaften: Security	1. Das Besucherportal steht im Internet, muss daher gut gegen entsprechende Angriffe abgesichert sein. 2. Die Verwaltung der Filme dagegen kann eine interne Anwendung sein. 3. Auch der Ticketverkauf muss nicht direkt im Internet stehen, er wird nur angestoßen bspw. durch Aktionen aus dem Besucherportal
Qualitätsteigenschaften: Last	Unterschiedliche Bounded Contexts müssen unterschiedlich skalieren. 1. Der Ansturm auf Tickets ist in den Tagen direkt vor oder zu Beginn einer Flexinale besonders hoch, wenn die Vorführungen für den Verkauf freigeschaltet werden. Dann muss der Ticketverkauf skalieren. 2. Die Vorbereitungen - Einstellen der Filme, Spielstätten, Programmplanung - dagegen werden über einen längeren Zeitraum und nur von einer sehr begrenzten Zahl von Anwendern durchgeführt.

Die Bounded Contexts "Finanzen" und "Reporting" sind hier gestrichelt markiert, weil sie außerhalb des eigentlichen fachlichens Scopes der Flexinale liegen. Der Bounded Context "Finanzen" wird in einem SAP-System umgesetzt, "Reporting" ist eine Data Warehouse-Anwendung. Beide benötigen jedoch Daten aus der Flexinale, die sie nur lesend verwenden.

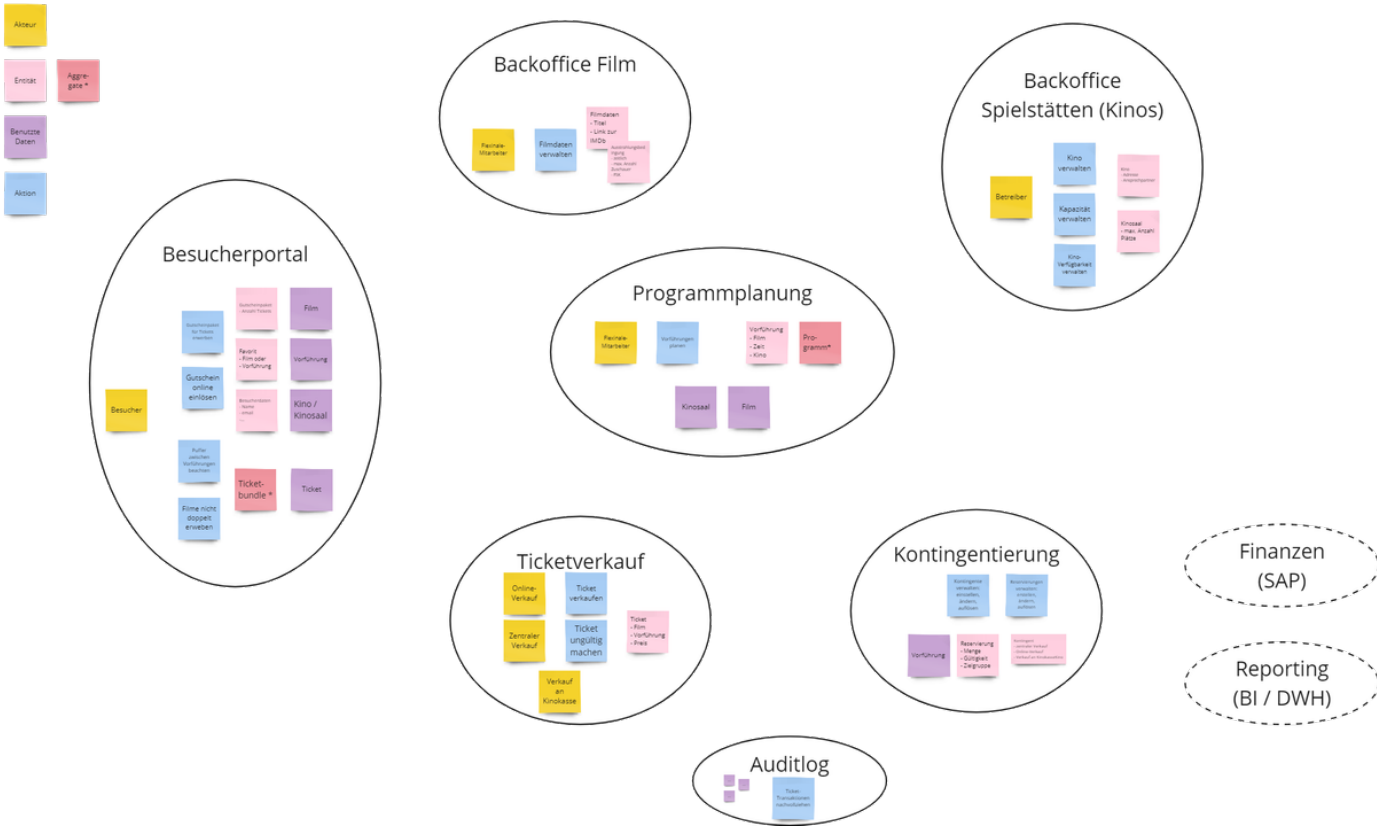


Abbildung: Bounded Contexts der Flexinale
(Quelle: selbst erstellt)

11. Context Map

Die Context Map veranschaulicht die Abhängigkeiten zwischen den verschiedenen Bounded Contexts. Die Pfeile zeigen den Datenfluss in der Richtung Upstream Downstream. Das heißt besipielsweise, dass der Bounded Context "Besucherportal" Daten aus dem "Backoffice Film" bekommt, also von diesem Bounded Context abhängt.

Die Bounded Contexts "Finanzen" und "Reporting" benötigen Daten aus fast allen Bounded Contexts.

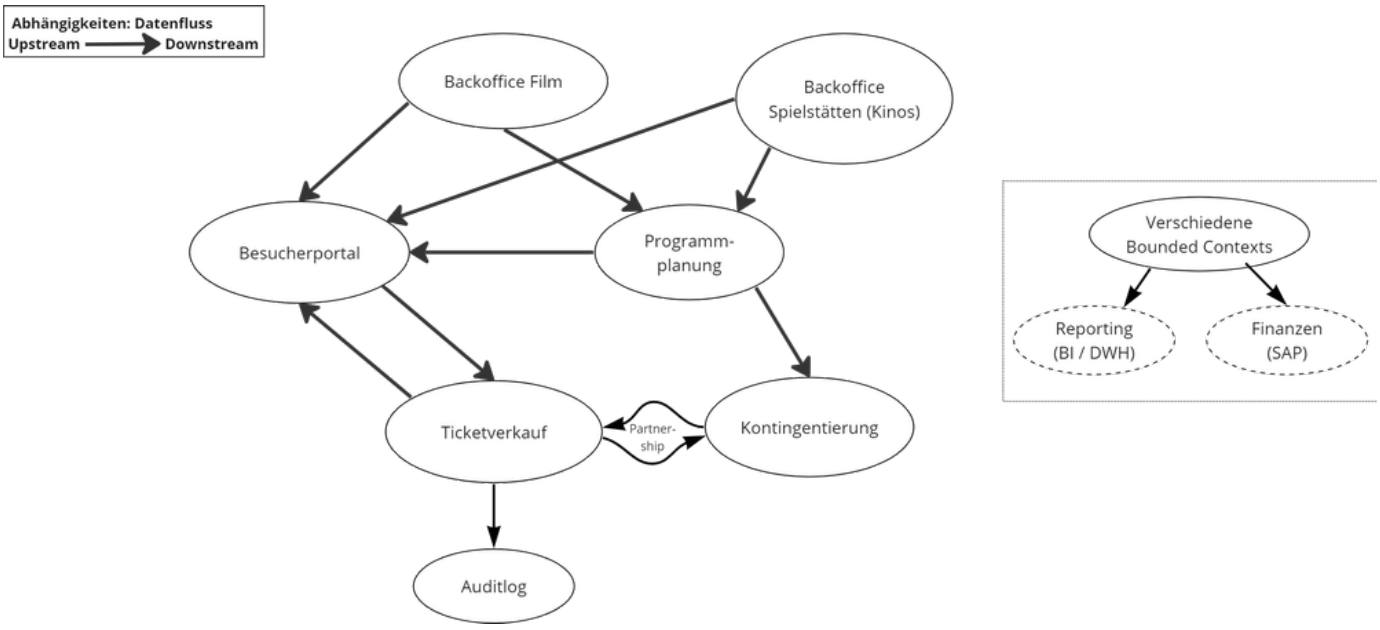


Abbildung: Context Map der Flexinale
(Quelle: selbst erstellt)

(B) How-To: Wie mache ich was in der Flexinale?

1. Wie richte ich die Anwendungen ein und bringe sie erstmalig zum Laufen?

Die Einrichtung der Infrastrukur, von IntelliJ, das erstmalige Bauen und Starten der Anwendung ist beschrieben in Installationsanleitung: Java, Spring-Boot, Maven, Sourcecode für die Case Study "Flexinale"

2. Wo finde ich den Code der vier verschiedenen Flexinale-Anwendungen?

Die vier Anwendungen sind in IntelliJ als ein Projekt mit vier Modulen angelegt. Sie sind direkt unterhalb des obersten Verzeichnisses flex-training-filmfestival zu finden:

1. flexinale-distributed
2. flexinale-modulith-1-onion
3. flexinale-modulith-2-components
4. flexinale-monolith

Auf oberster Ebene liegt außerdem das Verzeichnis *infrastructure*. Es enthält Skripte zum Starten der Datenbank und Kafka mit docker oder podman.

Das Modul *flexinale-distributed* ist in weitere Module zerlegt.

Alle Module folgen der Standard-Struktur von Maven-Projekte: *src/main/java, src/test/java*, etc.

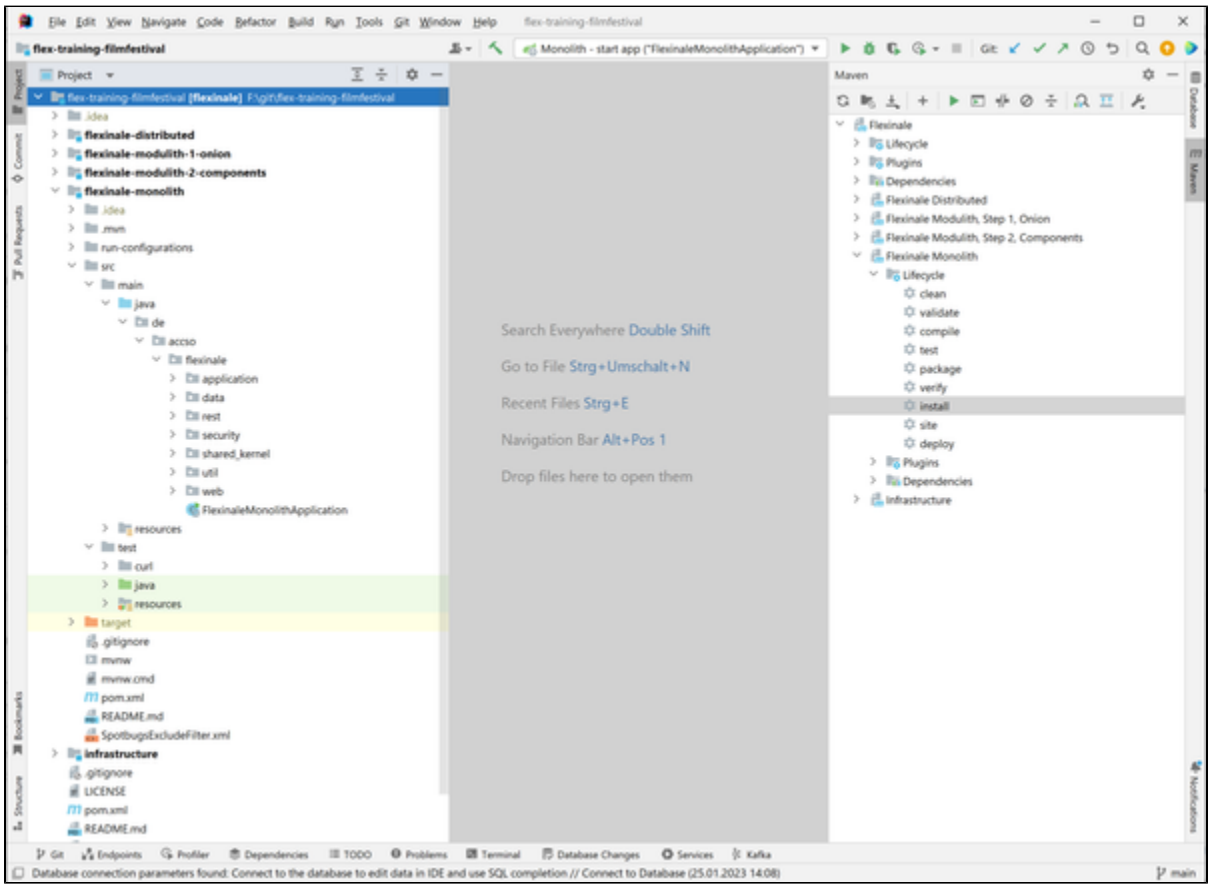


Abbildung: Flexinale-Varianten in IntelliJ
(Quelle: selbst erstellt)

3. Wie baue ich die Flexinale-Anwendungen?

Die Anwendungen werden mit Maven gebaut. Wie das genau geht, ist in 7 Case Study "Flexinale" (C) HowTo: Setup des Flexinale-Beispielcodes für jede der Anwendungen in den Abschnitten 8.a) / 9.a) / 10.a)/ 11.a) ausgeführt.

Der Maven-Build kann auf allen Ebenen der Modul-Hierarchie ausgeführt werden: Auf oberster Ebene für alle vier Anwendungen zusammen, für jede Anwendung getrennt, und getrennt für die Module in flexinale-distributed.

4. Wie starte ich die Infrastruktur (Datenbank und Kafka)?

- Alle vier Flexinale-Anwendungen, ebenso die Tests, benötigen eine laufende Datenbank.
- Nur die Flexinale distributed benötigt ein laufendes Kafka.

Die Datenbank und Kafka werden per docker oder podman gestartet.

1. Wie man die Datenbank erstmalig einrichtet startet, ist in 7 Case Study "Flexinale" (C) HowTo: Setup des Flexinale-Beispielcodes, 5.a) beschrieben.
2. Wie man die Datenbank nach der ersten Einrichtung startet und stoppt ist in 7 Case Study "Flexinale" (C) HowTo: Setup des Flexinale-Beispielcodes, 5.b) beschrieben.
3. Wie man Kafka startet und stoppt ist in 7 Case Study "Flexinale" (C) HowTo: Setup des Flexinale-Beispielcodes, 6. beschrieben.

5. Wo gibt es Testdaten?

Für jede Implementierungsvariante wird eine eigene Postgres-Datenbank benutzt. Wie diese eingerichtet werden, ist in 7 Case Study "Flexinale" (C) HowTo: Setup des Flexinale-Beispielcodes, 5.a) beschrieben.

Zur Befüllung steht jeweils ein Satz an Testdaten in Form eines Excel-Sheets zur Verfügung. Sie sind abgelegt unter

- im Monolith und beiden Modulithen: *src/test/resources/testdata/TestData.xlsx*
- in Distributed im Modul *flexinale-distributed-testdata*, dort ebenfalls unter *src/test/resources/testdata/TestData.xlsx*

Die Excel-Sheets haben verschiedene Tabellenblätter für die verschiedenen Datenarten.

Wer die Testdaten ergänzen oder verändern will, kann in diesen Excelsheets ändern.

6. Wie kann ich die Datenbank mit Testdaten füllen?

Die Testdaten können auf zwei verschiedene Arten in die Datenbank geladen werden:

1. Über per JUnit startbare Testdaten-Loader
2. oder über HTTP-Requests.

Option 2. ist im Abschnitt Backoffice -Datenpflege weiter unten beschrieben.

Hier beschreiben wir 1. über **per JUnit startbare Testdaten-Loader.**

Für das Laden per JUnit-Testdaten-Loader gibt es run configurations in IntelliJ, gruppiert nach der jeweiligen Anwendung (s. im Run/Debug-Configurations-Fenster oder im Servcies-Fenster von IntelliJ)

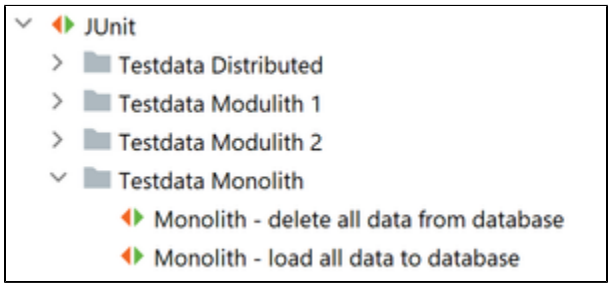


Abbildung: JUnit Test in IntelliJ
(Quelle: selbst erstellt)

Hier gibt es jeweils run configurations, um

1. Testdaten aus der Datenbank zu löschen (erstellt insbesondere eine Datenbank mit allen Schemata)
2. Testdaten in die Datenbank zu laden.

Im Monolith und Modulith gibt es nur jeweils eine run configuration für alle Datentypen gemeinsam, im Modulith 2 und in Distributed gibt es zusätzlich run configurationen, um Benutzer- und Daten zu Film, Kino, Kinosälen und Vorführungen getrennt zu laden.

Anlegen der Datenbank-Schemata
Die Datenbank-Schemata müssen nicht explizit angelegt werden.

Sie werden automatisch von Spring / Hibernate beim Starten der run-configuration *delete all data from database* gelöscht neu erzeugt.

Dafür sorgt die property `spring.jpa.hibernate.ddl-auto=create`.

Die JUnit-Klassen, die sich um Löschen und Laden der Testdaten kümmern, liegen in der jeweiligen Anwendung unter *src/test/java/testdata*, bzw. in der flexinale-distributed im Modul *flexinale-distributed-testdata* unter *src/test/java/testdata*.

Die **Testdaten** liegen in der jeweiligen Anwendung unter src/test/resources/testdata, bzw. in der flexinale-distributed im Modul flexinale-distributed-testdata unter src/test/resources/testdata, dort in einem oder mehreren Excelsheets.

Die Testdaten-Loader implementieren folgendes Verhalten:

1. Wenn ein Datensatz mit der gleichen ID noch nicht in der Datenbank vorhanden ist, wird der Datensatz aus den Excelsheet hinzugefügt.
2. Wenn ein Datensatz mit der gleichen ID bereits vorhanden ist, wird er mit dem Datensatz aus den Excelsheet überschrieben.
3. Datensätze in der Datenbank werden nie gelöscht.

7. Wo finde ich Testdaten?

Die Testdaten liegen in der jeweiligen Anwendung unter *src/test/resources/testdata*, bzw. in der flexinale-distributed im Modul *flexinale-distributed-testdata* unter *src/test/resources/testdata*, dort in einem oder mehreren Excelsheets.

8. Wie starte ich die Flexinale-Anwendungen?

Die Flexinale-Anwendungen starten wir aus IntelliJ heraus. Wie das genau geht, ist in 7 Case Study "Flexinale" (C) HowTo: Setup des Flexinale-Beispielcodes für jede der Anwendungen in den Abschnitten 8.c) / 9.c) / 10.c)/ 11.c) ausgeführt.

9. Gibt es statische Code Analyse für die Flexinale und wo finde ich sie?

Für die statische Code Analyse in der Flexinale wird spotbugs genutzt, sowie findsecbugs für Security-Bugs. Für beide ist das jeweilige Maven-plugin konfiguriert. Sie laufen in der "Verify"-Phase von Maven.

Ausnahmen, d.h. Warnungen oder Fehler, die spotbugs oder findswecbugs finden und die bewusst ignoriert werden sollen, sind (wo vorhanden) in der Datei *SpotbugsExcludeFilter.xml* konfiguriert. Sie liegt jeweils auf oberster Ebene eines Projektes.

Hinweis: Sporbugs und findsecbugs analysieren den Java byte code. Außerdem benutzen sie die SpotbugsExcludeFilter.xml, die im target-Verzeichnis eines Projektes liegt. Bei einer Analyse von geändertem Code oder einer Änderung an der SpotbugsExcludeFilter.xml ist also ein neuer maven build erforderlich.

10. Wo finde ich Architekturtests für die Flexinale?

Architekturtests gibt es für die Modulithen und Distributed. Für die Tests verwenden wir ArchUnit. Diese Tests liegen unter

Modulith-1	flexinale-modulith-1-onion/src/test/java/architecturetests
Modulith-2	flexinale-modulith-2-components/src/test/java/architecturetests
Distributed	in einem eigenen Maven-Projekt: flexinale-distributed/flexinale-distributed-test-architecture/src/test/java/architecturetests

Die ArchUnit-Tests sind als JUnit-Tests geschrieben. Damit

- können sie wie diese in IntelliJ direkt gestartet werden
- laufen sie in der test phase in Maven mit

11. Wo finde und wie starte ich Tests?

Tests für die Anwenung sind als JUnit-Tests geschrieben. Sie liegen jeweils unter

Monolith	flexinale-monolith/src/test/java/de/accso/flexinale
Modulith-1	flexinale-modulith-1-onion/src/test/java/de/accso/flexinale

Modulith-2	flexinale-modulith-2-components/src/test/java/de/accso/flexinale
Distributed	in zwei eigenen Maven-Projekten: 1. flexinale-distributed/flexinale-distributed-test-integrated/src/test/java - funktionale Tests. Hierbei werden die Anwendung integriert in einem einzigen Prozess gestartet. Zum Messaging wird ein In-Memory-Bus verwendet. 2. flexinale-distributed/flexinale-distributed-test-distributed/src/test/java - Verteilungs-Aspekte werden hier getestet.

Voraussetzungen:

1. Die Datenbank muss laufen
2. Für distributed muss für die Tests der Verteileung (Tests in flexinale-distributed-test-distributed) auch Kafka laufen.

Tests können gestartet werden

1. In IntelliJ, Rechtsklick auf die Testklasse und "Run..." / "Debug..."
2. Per Maven, test-Phase

12. Wie starte ich die Anwendung?

Die Anwendungen werden als Spring Boot Application gestartet. Für den Start der Anwendungen gibt es in der Gruppe Spring Boot jeweils eine Gruppe für die Run Configuration(s) in IntelliJ.

Infrastruktur für die Anwendungen
Damit die Anwendungen starten können:

- Muss PostgreSQL laufen.
- Müssen die Datenbankschemata angelegt sein.
  - Dazu reicht es, einmal die run configuration zum Löschen aller Daten aus der Datenbank zu starten (s. Abschnitt "Datenbank mit Testdaten füllen").
- Nur Distributed: Muss Kafka laufen.

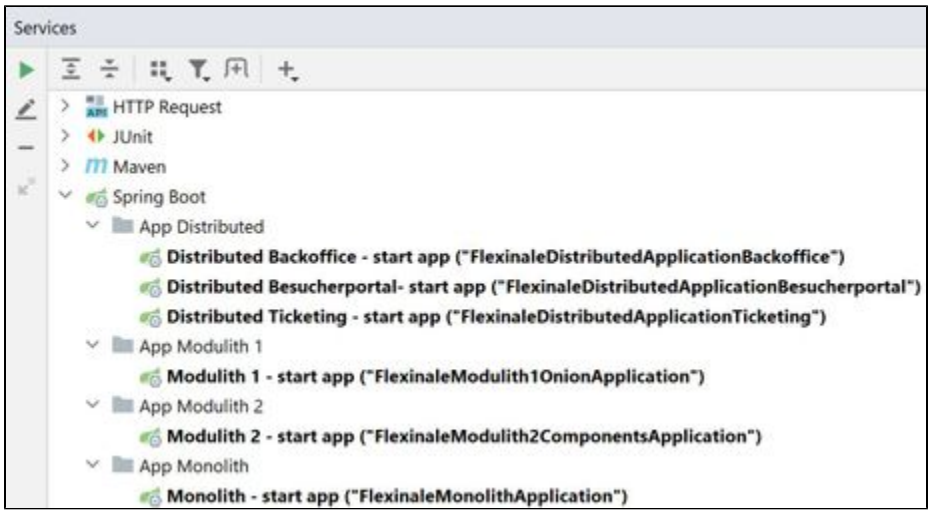


Abbildung: Services zum Start in IntelliJ
(Quelle: selbst erstellt)

Der Monolith und die beiden Modulithen bestehen jeweils aus genau einer Spring Boot Application, die gestartet werden muss.

Distributed besteht aus drei Spring Boot Applications. Diese müssen in der richtigen Reihenfolge gestartet werden.

Startreihenfolge für **Distributed**

1. Besucherportal (warten auf die Info "Started FlexinaleDistributedApplicationBesucherportal" in der Konsole)
2. Ticketing (warten auf die Info "Started FlexinaleDistributedApplicationTicketing" in der Konsole)
3. Backoffice

13. Wie greife ich auf die verschiedenen Flexinale-Anwendungen zu und wie logge ich mich ein?

Alle Anwendungen (in Distrubuted: Das Besucherportal) laufen unter **http://localhost:8080/**. Im Default horchen damit alle Anwendungen auf den gleichen Port, es kann also nur eine Anwendung zu einem Zeitpunkt gestartet werden.

Login

Die Anwendung kann nur mit einem eingeloggten Benutzer benutzt werden. Benutzer und Passwörter finden sich im Testdaten-Excelsheet, im Tabellenblatt *Benutzer*. Sie müssen die Rolle Besucher haben (ROLE_BESUCHER).

Vor der ersten Benutzung
Die Benutzer, die sich einloggen können, müssen in der Datenbank stehen.

Daher vor der ersten Benutzung einmal mindestens die Benutzer aus dem Testdatensatz in die Datenbank laden,
z.B. über die run configuration zum Laden aller Daten in die Datenbank (s. Abschnitt "Datenbank mit Testdaten füllen").

Beispielsweise kann man sich nach dem Laden der Benutzer-Testdaten mit dem Benutzer *man* mit Passwort *mann1* einloggen.

Hinweis: Die Passwörter der einzelnen Benutzer stehen nur im Excel-Sheet im Klartext. In der Datenbank sind sie verschlüsselt. Sie sind aber immer nach diesem Muster gebildet:

- Passwort = <login>1
- Beispiel: login: mann, Passwort: mann1

Anwendungen anhand der Header-Farbe unterscheiden

Zur besseren Orientierung hat jede Anwendung eine andere Header-Farbe:

Monolith	
Modulith 1 (Onion)	
Modulith 2 (Components)	
Distributed	

14. Wie pflege ich die Daten zu Filmen, Kinos, Kinosälen, etc.?

Die UI für die Datenpflege im Backoffice ist eine REST-Schnittstelle, über die ein Excel Sheet mit den gewünschten Daten geladen werden kann. Die REST-Controller akzwptieren dazu Excel-Sheets mit Daten in dem gleichen Format wie auch die Testdaten abgelegt sind. Zum Test der REST-Schnittstelle können also die Excelsheets mit den Testdaten (s.o.) verwendet werden.

Vor dem Laden der Daten
Damit die Daten überhaupt geladen werden können:

1. Muss die Anwendung laufen.
2. Muss es einen Benutzer in der Benutzer-Tabelle geben, der die Rolle Admin hat (ROLE_ADMIN). In den Testdaten gibt es dafür den User admin (Passwort: admin1).

Die benötigten HTTP-Requests kann man direkt aus IntelliJ heraus absetzen oder per curl von einer Kommandozeile, dazu s.u.

In IntelliJ gibt es run configurations für die HTTP-POST-Requests zur Datenpflege (in diesen ist der User admin mit Passwort admin1 hinterlegt):

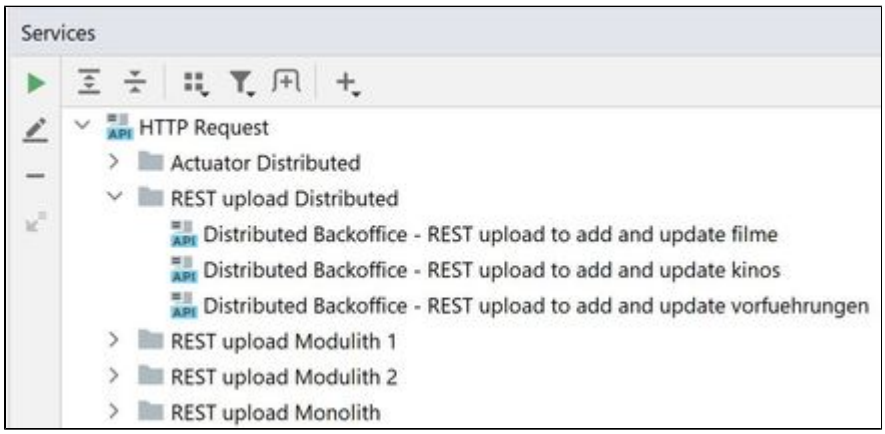


Abbildung: Services zum Absetzen von REST-Post-Calls
(Quelle: selbst erstellt)

Per HTTP GET kann man außerdem die Daten zu Film, Kino, Kinosälen und Vorführungenüber die REST-Controller auslesen. Dazu braucht man die Rolle Besucher (ROLE_BESUCHER). Für die GET-Requests sind keine run-configurations hinterlegt.

Einfügen und aktualisieren von Daten zu Film, Kino, Kinosälen, Vorführungen
Mit dem hier beschriebenen Mechanismus kann man

- Neue Daten hinzufügen.
- Vorhandenen Daten ändern.
- Benutzerdaten können nicht über die Datenpflege des Backoffice geladen werden. Sie müssen über die Testdatenloader eingespielt werden.

Das Löschen vorhandener Daten ist nicht möglich.

(S.a. Beschreibung der Flexinale)

HTTP-Requests per Curl

Curl und WSL2

⚠️ Curl funktioniert nicht von WSL2 auf die Flexinale-Anwendungen, die unter WIndows (z.B. in IntelliJ) laufen, da der Zugriff auf localhost nicht möglich ist (siehe hier). Auch Änderungen der wslconfig (siehe hier) funktioniert leider nicht.

Curl muss man daher von der Windows-CMD oder der Windows-Powershell nutzen.

Für CLI-Benutzer (Windows CMD oder Powershell): GET-Requests für die Daten zu Film, Kino, Kinosälen und Vorführungen

(Hier wird ein Benutzer mit Besucher-Rechten benötigt (ROLE_BESUCHER)

Curl für GET-Requests auf Rest-Controller: Daten auslesen

```
# Port bei Monolith und Modulith1/2 ist 8080, bei Distributed läuft die
Backoffice-App auf 8081

# Filme
curl -v -X GET -u USER:PASSWORT http://localhost:PORT/rest/filme

# Kinos (und -saele)
curl -v -X GET -u USER:PASSWORT http://localhost:PORT/rest/kinos

# Vorfuehrungen
curl -v -X GET -u USER:PASSWORT http://localhost:PORT/rest/vorfuehrungen
```

Für CLI-Benutzer (Windows CMD oder Powershell): POST-Requests zum Einspielen von Daten zu Film, Kino, Kinosälen und Vorführungen

(Hier wird ein Benutzer mit Admin-Rechten benötigt (ROLE_ADMIN)

Curl für POST-Requests auf Rest-Controller: Daten neu einspielen oder updaten

```
# Wechsel in Verzeichnis, wo XLSX-Datei liegt
cd ...\src\test\resources\testdata

# Curl

# Port bei Monolith und Modulith1/2 ist 8080, bei Distributed läuft die
Backoffice-App auf 8081

# Filme
curl -v -X POST -u USER:PASSWORD -F
file=@TestdataFilmKinoKinoSaalVorfuehrung.xlsx http://localhost:PORT
/rest/filme

# Kinos (und -saele)
curl -v -X POST -u USER:PASSWORD -F
file=@TestdataFilmKinoKinoSaalVorfuehrung.xlsx http://localhost:PORT
/rest/kinos

# Vorfuehrungen
curl -v -X POST -u USER:PASSWORD -F
file=@TestdataFilmKinoKinoSaalVorfuehrung.xlsx http://localhost:PORT
```

```
/rest/vorfuehrungen

# Benutzer
# ... können absichtlich NICHT per Curl geladen werden. Sie müssen in
der Datenbank vorhanden sein, sonst ist die Authentifizierung nicht
möglich (401)
```

15. Monitoring

Monitoring-Endpunkte gibt es nur in der Variante Distributed.

Für das Monitoring nutzt die Flexinale Spring Boot Actuator. Hierfür gibt es Endpunkte in den jeweiligen Applikationen flexinale-distributed-besucherportal, bzw. flexinale-distributed-backoffice, bzw. flexinale-distributed-ticketing, dort im infrastructure-Package.

Am Beispiel Besucherportal:

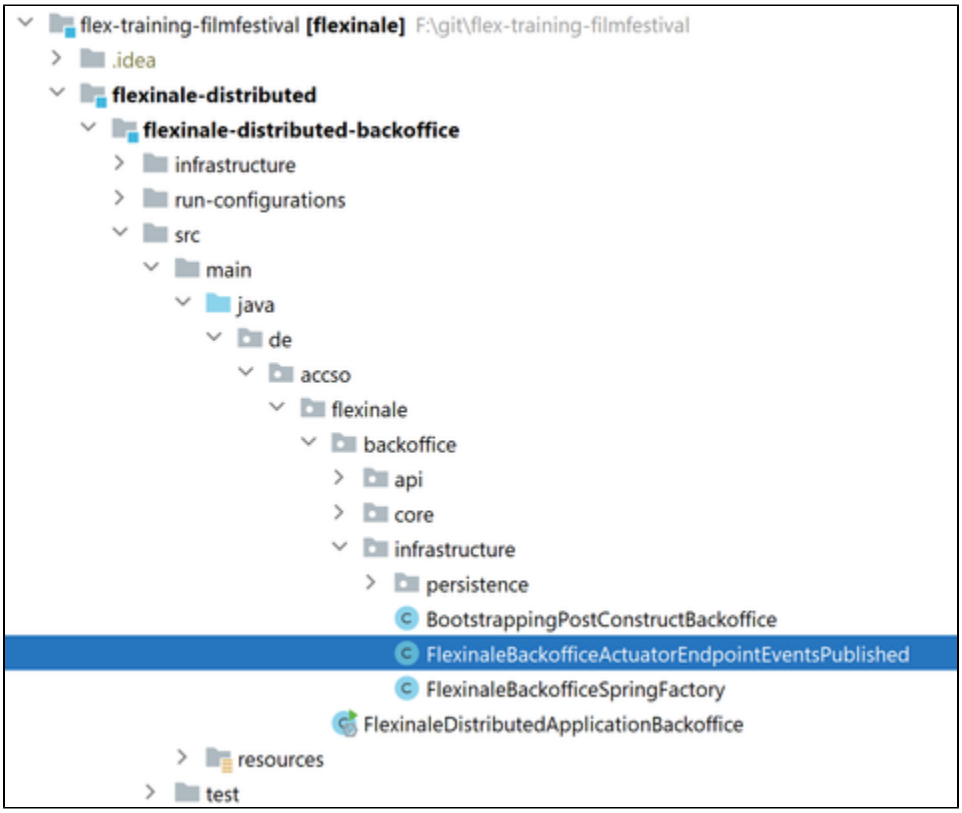
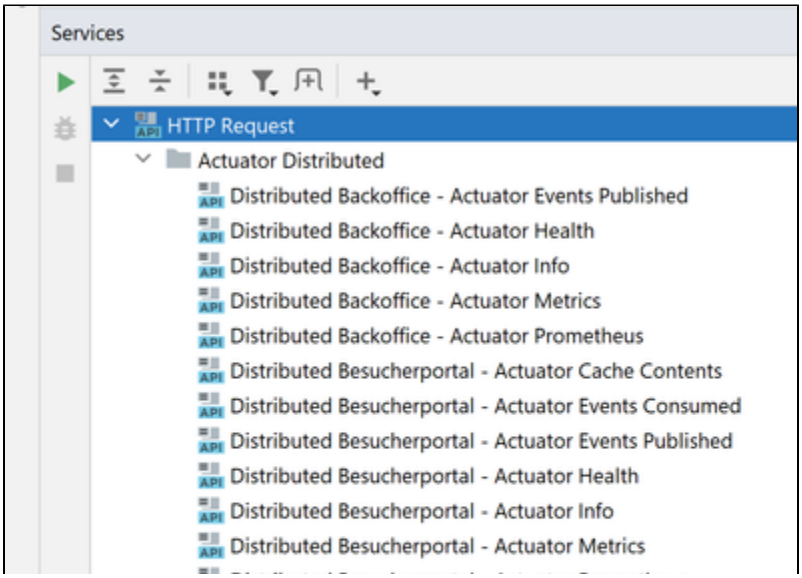


Abbildung: Actuator-Endpoint in flexinale-distributed, backoffice (Quelle: selbst erstellt)

Die Actuator-Endpoints können per HTTP-Request aufgerufen werden.

In IntelliJ gibt es run configurations für die HTTP-GET-Requests zum Aufrufen der Actuator-Endpoints (in diesen ist der User admin mit Passwort admin1 hinterlegt):



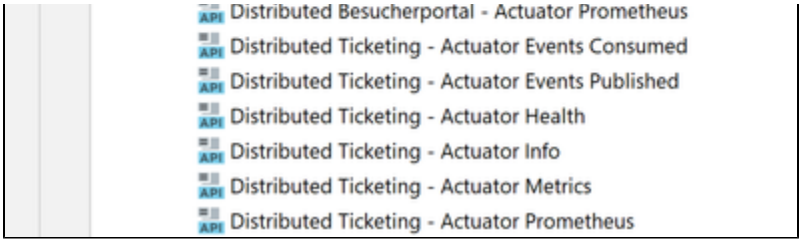


Abbildung: Services zum Absetzen von HTTP-Requests für Actuator-Endpoints (Quelle: selbst erstellt)

HTTP-Requests per Curl

Curl und WSL2

⚠ Curl funktioniert nicht von WSL2 auf die Flexinale-Anwendungen, die unter WIndows (z.B. in IntelliJ) laufen, da der Zugriff auf localhost nicht möglich ist (siehe hier). Auch Änderungen der wslconfig (siehe hier) funktioniert leider nicht.

Curl muss man daher von der Windows-CMD oder der Windows-Powershell nutzen.

Für CLI-Benutzer (Windows CMD oder Powershell): GET-Requests für Actuator

Hier wird ein Benutzer mit Admin-Rechten benötigt (ROLE_ADMIN)

Beispiel:

Curl für GET-Requests auf Rest-Controller: Daten auslesen

```
# Beispiel: Aufruf des info-Acutator-Endpunkts in Ticketing. Die Die
Ticketing-Application läuft in Distributed auf Port 8082

curl -v -u USER:PASSWORD http://localhost:8082/actuator/info
```

16. Wie baue ich Docker-Images aus den Flexinale-Anwendungen?

Docker-Images der Flexinale-Anwendungen benötigen wir nicht für die Schulung. Es reicht dafür völlig aus, die Anwendungen in IntelliJ zu starten.

Im Projekt für die jeweilige Applikation gibt es im Verzeichnis infrastructure jeweils Skripte für Docker, bzw. Podman. In Monolith, Modulith-1 und Modulith-2 liegen die Skripte zum Bauen und Starten der Images im gleichen Verzeichnis

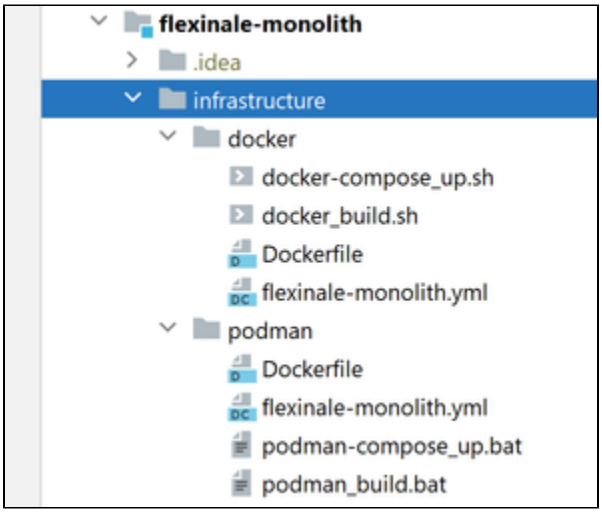


Abbildung: Monolith:Verzeichnis infrastructure mit Docker-/Podman-Skripten (Struktur in Modulith-1 und Modulith-2 analog)
(Quelle: selbst erstellt)

In Distributed gibt es infrastructure-Verzeichnisse in den jeweiligen Anwendungen (zum Bauen der Images), sowie eines auf oberster Ebene zum Starten der gesamten Anwendung.

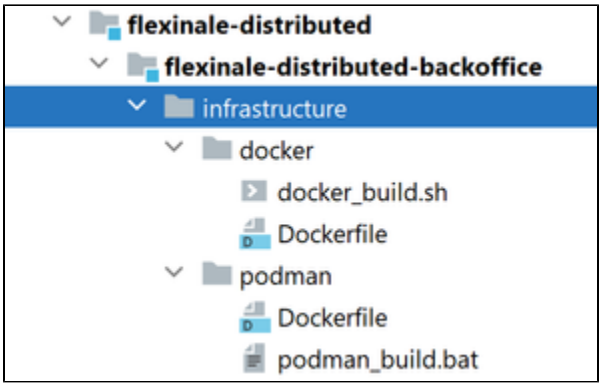


Abbildung: Distributed: Verzeichnis infrastructure in backoffice mit Docker-/Podman-Skripten zum Bauen (Struktur in besucherportal und ticketing analog)
(Quelle: selbst erstellt)

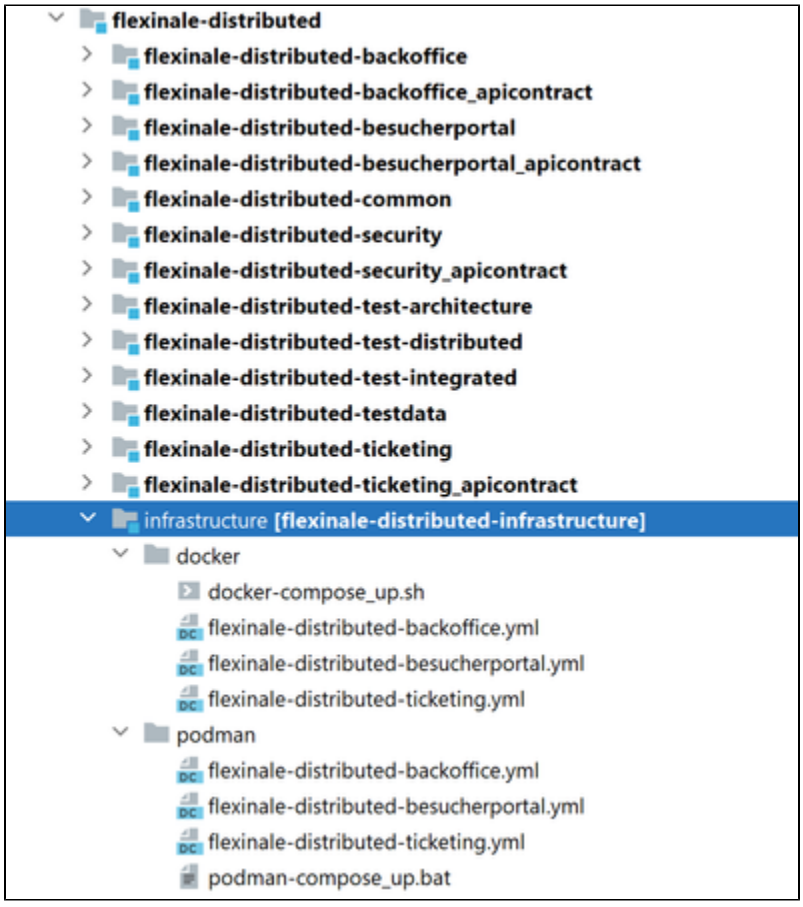


Abbildung: Distributed: Verzeichnis infrastructure in backoffice mit Docker-/Podman-Skripten zum Starten mit Compose
(Quelle: selbst erstellt)

Zum Bauen der Images gibt es Build-Skripte:

- Für Docker: In einer Shell (z.B. WSL2) in das passende Unteverzeichnis wechseln, dort `docker_build.sh` aufrufen.
- Für Podman: In einer CLI (Windows CMD oder Powershell) in das passende Unteverzeichnis wechseln, dort `podman_build.bat` aufrufen.

Zum Starten der Anwendung inkl. Infrastruktur (Monolith / Modulith-1 und -2: inkl. Postgres-Datenbank; Distributed: zusätzlich Kafka) gibt es compose-Skripte, Verzeichnisse s.o..

- Für Docker: In einer Shell (z.B. WSL2) in das passende Unteverzeichnis wechseln, dort `docker-compose_up.sh` aufrufen.
- Für Podman: In einer CLI (Windows CMD oder Powershell) in das passende Unteverzeichnis wechseln, dort `podman-compose_up.bat` aufrufen.

Compose in Distributed

⚠ Der Start der gesamten Anwendung mit den Applikationen als Docker-/ Podman-Images funktioniert derzeit in Distributed nicht. Hier können sich die Applikationen nicht mit Kafka connecten. Die Fehlerursache ist derzeit unklar.

(C) Entwicklungstagebuch: Monolith > Modulith 1 > Modulith 2 > Verteiltes System

	"ab" Anwendung	was (Thema)	warum	Referenz im Code (Link zu Github)
1	**Monolith**	RestController als "Backoffice"		
2	**Monolith**	TicketBundle - unser Aggregate		
3	**Monolith**	Rollen Besucher, Admin; Benutzer mit unterschiedlichen Rollen		
4	**Modulith-1**	Onion-Architektur		
5	**Modulith-1**	Trennung von domain und infrastructure: 1. domain model, Daos getrennt von Entitäten, Repositories 2. Keine Spring-Annotationen in Domäne, dafür DAos als Interfaces, Delegates, SpringFactory		
6	**Modulith-1**	Trennung von Besucher und Benutzer-Entität	Verschiedene Blickwinkel: Für A&A wird login, password, Rolle benötigt. Besucher ist fachlich, wird im Besucherportal benutzt. Wir ziehen das nicht so konsequent durch, insbesondere machen wir nichts wirklich mit dem Benutzer. Wir verwenden ihn nur für Login und die ID an den Tickets.	BenutzerEntity Besucher
7	**Modulith-1**	Equals by content	Equals-Check ohne Version	
8	**Modulith-1**	Version in den Entitäten (auch model, später TOs)	Implementierung von Optimistic Locking	
9	**Alle**	**Good Practice**: Architekturtests für Soll-Architektur vorab schreiben (nein, nicht alle, aber... vorab) (ATDD = Architecture Test Driven Design)	So überlegt man sich vorab, wie die Architektur aussehen soll und sieht regelmäßig, wo man steht	
10	**Modulith-2, Distributed**	Im besucherportal und ticketing nur noch besucherId anstelle des vollständigen Benutzers benutzen Vorführung hat kein Kontingent mehr, dafür hat Kontingent nun die VorführungsId. Ticket hat nur noch die IDs von Film, Vorführung, Besucher anstatt der vollständigen Entität.	Abhängigkeiten im Datenmodell lösen gemäß den fachlichen Komponenten: Abhängigkeiten nur noch über die ID (Fremdschlüssel), nicht mehr als Relation auf die vollständige Entität.	
11	**Alle**	Tests müssen die Architekturtests nicht bestehen.	Viele integrative Tests	Per selbstgeschriebener Annotation (in allen Projekten identisch): DoNotCheckInArchitectureTests Man kann in ArchUnit auch per Importer Testklassen ausschließen.
12	**Modulith-2, Distributed**	Security, onion haben keine Onionarchitektur und müssen auch keine haben	Primär, weil weitgehend Implementierung von SpringSecurity-Klassen, damit ist die Struktur vorgegeben.	z.B. in Modulith2: security
13	**Alle**	(Spring-)Context in alle Testklassen isolieren	Sonst gibt's durcheinander, auch auf der Datenbank	Mit Annotation `@DirtiesContext(classMode = DirtiesContext.ClassMode.BEFORE_CLASS)`
14	**Alle**	Bidirektionale Beziehung Kino KinoSaal: @JsonIgnore im Kino im KinoSaal (Alles, was serialisiert wird - TOs oder Domänenobjekte oder - im Monolith - Entitäten) Bidirektionale Beziehungen (Kino KinoSaal) sind echt anstrengend 😕	Sonst gibt es eine Endlosschleife. Wir haben uns für Kino im KinoSaal entschieden (da Kino die abhängige Entität ist)	
15	**Alle**	Tests und damit Testverzeichnisse aufgeteilt nach • Architekturtests • Testdaten-Cleaner und -Loader • "echten" Tests der Anwendung	Sind verschiedene Kategorien von Tests, das will man weit oben unterscheiden. Die TestDataLoader sind eigentlich nur aus Convenience als Tests geschrieben - so kann man sie ganz einfach starten.	
16	**Distributed**	(Fast) alle Tests nach distributed-test-integrated verschoben.	Sehr viele integrierte Tests. Es lohnt sich nicht, kleinteilig nach Tests in nur den einzelnen Anwendungen zu suchen.	

17	**Distributed**	EventBus und Events sind designed und implementiert: 1. Bus gibt es InMemory (synchron) und via Kafka (asynchron) 2. Bus ist event-typisiert, über einen Bus kommt immer nur ein Typ von Events (bei Kafka also in 1 Topic nur ein Event-Typ) 3. Event-Interface und AbstractEvent als Vorgaben, haben auch a. ID, Context mit Historie und Correlation-ID. b. Tests sichern ab, dass die Events die richtige Struktur haben (z.B. Version, Default-Konstruktor), damit auch Serialisierung per Jackson funktioniert. Subscriber und Publisher empfangen/senden Events.		
18	**Distributed**	Die Retriever aus Modulith-2 werden zu Consumern und der Kontrollfluss dreht sich von Pull nach Push.		
19	**Distributed**	InMemoryCache implementiert	Zur Benutzung überall, wo nicht-eigene Daten abgelegt werden sollen, z.B. im Besucherportal für alle Daten FKKsV.	
20	**Distributed**	KafkaConsumerAdapter und KafkaProducerAdapter als eigene Implementierung erstellt, nicht per Spring @Listener,... Annotation	Nur so können wir selber die Topic-Namen vergeben.	KafkaConsumerAdapter KafkaProducerAdapter
21	**Alle**	@Transactional an allen Rest- oder Web-Controllern gesetzt, wo es Änderungen gibt (POST).		
22	**Distributed**	VorfuehrungSubscriber idempotent machen, so dass dieser mit mehreren VorfuehrungCreatedEvents für die gleiche Vorfuehrung umgehen kann (Folge-Created-Events werden erkannt und ignoriert)		VorfuehrungSubscriber
23	**Distributed**	Error Handling in Kafka Consumers: DeadLetterTopic (DLT)		KafkaConsumerErrorHandler
24	**Distributed**	KinoId im VorfuehrungTO ergänzt.	Serialisierungsproblem, bzw. Problem der bidirektionalen Relation. Bei der Kino KinoSaal-Relation muss an einer Seite ein @JsonIgnore stehen, sonst gibt es eine Endlosschleife. Diese steht am Jino des KinoSaals.	VorfuehrungTO KinoSaalTO
25	**Distributed**	Behandlung Doppelzustellung eines GutscheinEinloesen-Events	Auch Gutschein einlösen muss idempotent sein	GutscheinEinloesenBeauftragtSubscriber
26	**Distributed**	@PostContructs für Backofficeund Ticketing: Publizieren ihren gesamten Datenbankinhalt (Besucherportal hat keine Daten, die andere interessieren)	Restart von Anwendungen; Kafka "Read from Beginning" ist nicht implementiert.	1. BootstrappingPostConstructBackoffice 2. BootstrappingPostConstructTicketing
27	**Monolith / Modulith-1 / Modulith-2**	Gutschein einlösen: Rückmeldung im Besucherportal, ob der Ticketkauf erfolgreich war		
28	**Distributed**	Gutschein einlösen: Meldung "Ticketkauf beauftragt". (Rückmeldung, wenn er fehlschlug wegen Kontingent ausgeschöpft, nicht implementiert)	... da asynchron.	
29	**Distributed, Modulith-2**	Film als vollwertiges TO im VorfuehrungTO (war vorher nur filmId)	Genauso behandeln wie KinoSaal.	VorfuehrungTO in Modulith-2 VorfuehrungTO in Distributed
30	**Alle**	Monolith / Modulith-1 / Modulith-2 / Distributed sind an Header-Farben unterscheidbar	Hilft beim Erkennen, wo man gerade ist.	
31	**Distributed**	Verschieden Profiles eingeführt: 1. test-integrated 2. test-distributed 3. testdata 4. configtest	1) Erzeugen des passenden Event-Busses (bzw. der -Factory): 1. Keines der Profile: KafkaAsyncEventBus 2. test-integrated: InMemorySyncEventBusSpy 3. testdata, configtest, configtest: NopeEventBus	FlexinaleCommonSpringFactory FlexinaleDistributedApplicationBesucherportal, FlexinaleDistributedApplicationBackoffice, FlexinaleDistributedApplicationTicketing versus FlexinaleDistributedApplicationTestIntegrated BootstrappingPostConstructBackoffice

		5. smoketest 6. local	2) Passende SpringBootApplication auswählen - Unterscheidung, ob die drei Applications separat laufen oder integriert als eine einzige Anwendung 3) PostConstruct wird nicht in den integrierten Tests aufgerufen und nicht beim Erstellen der Testdaten.	BootstrappingPostConstructTicketing (Besucherportal hat kein PostConstruct)
32	**Distributed**	Alle Spring Factories: Die creates mit Präfix für Application versehen	Um in den integrated-Tests alles zusammen als eine einzige Anwendung starten zu können, brauchen alle Spring-Beans (hier: mit @Bean annotierte Methoden) unterschiedliche Namen.	FlexinaleBesucherPortalSpringFactory FlexinaleBackofficeSpringFactory FlexinaleTicketingSpringFactory
33	**Distributed**	Update der Daten Film, Kino / Kinosaal, Vorfuehrung möglich: Wenn F/K/Ks/V bereits vorhanden, Update. VorfuehrungUpdatedEvents werden nicht nur für die direkt betroffenen Entitäten publiziert, sondern auch für die indirekt betroffenen Vorfuehrungen. NUR in Distributed umgesetzt! (in MMM weiterhin nur ADD möglich!)	Beispiel für Änderungen an DatenF/K/Ks/V und Umgang mit deren fachlich abhängigen Daten	Backoffice. Insbesondere: FilmRestController, Methode `publishFilmsAndVorfuehrungen` KinoKinoSaalRestController, Methode `publishKinosAndVorfuehrungen`
34	**Distributed**	Tickets ungültig machen, wenn sich eine Vorführung ändert (direkt oder indirekt - also auch, wenn sich am Film oder dem KinoSaal in der Vorführung etwas ändert). Geänderte Vorführung im Ticketing: 1. Tickets für diese Vorführung ungültig machen (+ publishen) 2. Online-Kontingent zurücksetzen für diese Vorführung. Ungültige Tickets im besucherportal handhaben: 1. Anzeige ungültiger Tickets 2. Nur gültige Tickets sind relevant beim Berechnen überlappender Vorführungen oder "Ticket bereits vorhanden".	Fachlich sehr einfacher  Umgang mit sich ändernden Vorführungen	Etliche Stellen im Besucherportal, Ticketing
35	**Modulith-1, Modulith-2, Distributed**	Unterscheide die unterschiedlichen Arten von api: 1. apicontract 2. eingehende api (web, rest, retriever) Siehe auch unten bei Nr. #45 (Trennung dort von `api_in` und `api_out`	Abhängigkeiten sortieren: Welche apiX darf welche apiY aufrufen.	Regeln s. Architekturtests in jeder Anwendung: Architekturtests Modulith 1 Architekturtests Modulith 2 Architekturtests distributed
36	**Distributed**	Actuator-Endpunkte erstellt	Fachliches und technisches Monitoring.	http-Request jeweils abgelegt in den drei Applications besucherportal backoffice ticketing
37	**Distributed**	PredecessorEventContext, insbes. Correlation ID, bei Eventketten weitergeben. 1. Events, die Teil einer Eventkette sind, haben ein PredecessorEventContext, die anderen nicht. 2. Der EventContext wird beim Aufrufen der Subscriber gesetzt und anschließend removed. 3. Beim Publish wird der Predecessor aus dem EventContext geholt und gesetzt (wenn vorhanden) 4. Implement ist der EventContextHolder über ThreadLocal	Damit wir Eventketten identifizieren können	1. TicketGekauftEvent mit PredecessorEventContext, GutscheinEinloesenBeauftragtEvent dagegen ohne, da Beginn einer Eventkette. 2. Befüllen des EventContext: 3. Auslesen und Setzen des Predecessors (insbes. Correlation ID) aus dem EventContextHolder: TicketPublisher, beide publish-Methoden. 4. Interface EventContextHolder, Implementierung
38	**Distributed**		Die (fachlichen) Tests der korrekten Behandlung	

		Korrekte Behandlung von PredecessorEventContext / Correlation ID für synchronen Event Bus: Dazu eigene Implementierung des EventContextHolder für den synchronen Fall. Zähler, wie oft er gesetzt wurde. Hinweis: Diese Implementierung funktioniert, wenn es zu einer Zeit nur maximal einen EventContext gibt. Für mehrere wäre ein Stack nötig. Der passende EventContextHolder (synchron / asynchron) wird im jeweiligen Eventbus gesetzt.	Correlation ID sollen synchron ausführbar sein.	Implementierung des Interface EventContextHolder durch InMemorySyncEventContextHolder. Eventbusse mit jeweils passendem EventContextHolder: 1. KafkaAsyncEventBus 2. InMemorySyncEventBus
39	**Distributed**	Erlaube Registrierung ungetypter "Generic" Eventsubscriber, um alle Events zu erhalten.	z.B. für ein Auditlog, das sämtliche Events loggen soll.	
40	**Distributed**	Kafka-Konfiguration: Consumer auf "earliest" umgestellt (aber siehe auch unten bei #43) Kein ID-Suffix mehr in Consumer Groups bei Subscription an Topics	Dadurch lesen 1. bei erstmaliger subscription die Consumer alle Events im Topic (es existiert noch kein Offset) 2. bei erneuter subscription die Events seit dem letzten Lesen. Die Anwendungen müssen nun nicht mehr in einer bestimmten Reihenfolge gestartet werden.	Änderung in KafkaConfiguration.java
41	**Alle**	Docker-Build-Files + Skripte für Flexinale erstellt	Die Flexinale selbst kann nun "dockerisiert" werden. Außerdem gibt es dazu build-Skripte und Compose-Skripte zum Starten. ⚠ In Distributed funktioniert leider das Compose nicht. Die Flexnale-Applikationen finden Kafka nicht. Problem unklar.	In den jeweiligen infrastructure-Verzeichnissen auf oberster Ebene der Varianten, bzw. in distributed in den Applikations-Projekten für Besucherportal / Backoffice / Ticketing

42	**Modulith-1, Modulith-2 und Distributed**	Typisierte Attribute, insbesondere für Version und Id	Attribute sind nun typisiert. Statt "Roh"-Typen wie Integer und String werden nun explizite Typen benutzt:
			
			1. Diese Typen sind Records und Teil der "umgebenden Klasse", also nicht in separaten Dateien.
			2. Id ist ein Record des Interfaces Identifiable
			3. Version ist ein Record des Interfaces Versionable
			
			Einschränkungen:
			
			1. In den Persistenz-Klassen wurden bewusst die Roh-Typen weiter benutzt, da man sonst das Hibernate-Mapping hätte erweitern müssen.
			2. Typisierte Attribute werden daher benutzt in: Domain-Klassen, TO-Klassen, Event-Klassen
			3. Daher wurden insbesondere umgestellt: Modulith-1, Modulith-2 und Distributed.
			
			Es gibt dazu nun Architekturtests, die auf die Typisierung der Attribute prüfen, z.B. "Wird in einer Domain-Klasse eine ID-Klasse benutzt anstelle eines Strings?"
				Beispiel: Domain-Klasse Film in Modulith-2
				
				`public class Film`
				`  implements Identifiable, Versionable, EqualsByContent {`
				
				`  public record Titel(String raw) implements RawWrapper<String> {}`
				`  public record ImdbUrl(String raw) implements RawWrapper<String> {}`
				`  public record DauerInMinuten(Integer raw) implements RawWrapper<Integer> {}`
				
				`  public final Id id;`
				`  public final Version version;`
				`  public Titel titel;`
				`  public ImdbUrl imdbUrl;`
				`  public DauerInMinuten dauerInMinuten;`
				
				Architekturtest in Modulith-2: Für Domain-Klassen, für Entity-Klassen, für TO-Klassen

43	**Distributed**	Kafka-Konfiguration: Consumer sind nun in Consumer-Gruppen und geben an, wie sie bei Start lesen wollen
		
		Löst diese Probleme:
		
		1. Consumergroup in Kafka hatte die technische Adapter-Klasse "KafkaConsumerAdapter" im Namen der ConsumerGroup
		2. Ohne UUID-Suffix in der Consumer-Group hiessen haben Subscriber in Besucherportal und Ticketing gleich nur einer erhält Stammdaten
		3. Mit UUID-Suffix werden nach Restart / Wiederanlauf (weil "earliest") alle Bewegdaten (z.B. GutscheinEinloesen-Aufträge) nochmal verarbeitet.
		
		Man muss also explizit unterscheiden, welchen Typ der Subscriber hat und wie er mit den Daten in einem Topic (als Stammdaten (i.d.R. immer alles neu einlesen) oder als
			Wird Kafka als verteilter, asynchroner Bus benutzt, so gibt nun jeder Subscriber an:
			
			1) In welcher Gruppe bin ich? Als Gruppen sind aktuell die Application-Namen ("besucherportal", "ticketing", ...) fest vorgegeben. Zum Beispiel sind alle Subscriber im Besucherportal in einer Gruppe.
			
			2) Wie soll das Startverhalten für die Subscription aussehen? Möglich sind eine der folgenden drei Optionen:
			
			`// (re)read all messages from the EventBus`
			`// from the very beginning (i.e. from offset 0)`
			`// In Kafka this is -`
			`// for a new ConsumerGroup - "earliest".`
			`START_READING_FROM_BEGINNING,`
			
			`// read all messages from now on (so don't`
			`// (re)read anything published before now)`
			`// In Kafka this is -`
			`// for a new ConsumerGroup - "latest".`
			`START_READING_FROM_NOW,`
			
			`// read all messages from the EventBus,`
			`// starting where we left off last time`
			`// (so last offset we had + 1)`
			`// (if we had not connected before at all: start`
			`// from beginning, i.e. from offset 0)`
			`// In Kafka this is - for a known ConsumerGroup -`
			`// the default.`
			`START_READING_FROM_LAST_TIME`
				EventSubscriptionAtStart
				
				Methode consumerConfig() in KafkaConfiguration
				
				InMemorySyncEventBus und InMemorySyncEventBusSpy
				
				Verteilmechanismus: Selection

		Bewegdaten (i.d.R. nur die neuesten Daten neu einlesen) umgehen soll. Wegen Problem 2. hatten wir zwischenzeitlich mal Änderung #40 (UUID-Suffix, earliest) reaktiviert, aber damit andere Probleme, siehe oben.	Die InMemory-Bus-Variante berücksichtigt ebenfalls das Startverhalten und die Consumergruppen. Anders als vorher bekommen also *nicht mehr* alle Subscriber auf einen Event-Typ ein Event als Broadcast zugeschickt, falls sie in der gleichen Gruppe sind (dabei sind unterschiedliche Verteilalgorithmen möglich, i.d.R. ist "RoundRobin" sinnvoll, sprich nur ein Subscriber in der Gruppe bekommt die Nachricht, danach der nächste etc.). Damit unterscheidet sich die InMemory-Variante von Kafka nur noch: 1. Kafka ist asynchron - InMemory ist synchron 2. Kafka hat Partitionen - InMemory nicht 3. Kafka macht automatische Verteilung der Consumer über Partitionen - InMemory macht das über den eingestellten Verteilmechanismus ("Selection")	
44	**Distributed**	Datenbank-Schemata aufgeteilt	Jeder Service hat nun sein eigenes Datenbank-Schema (weiterhin jedoch in der gleichen Infrastruktur: Es gibt weiterhin nur eine Postgres-DB): 1. besucherportal: distributed-besucherportal: Enthält die Benutzer-Tabelle (für A&A) 2. backoffice: distributed-backoffice: Enthällt die Benutzer-Tabelle (für A&A), sowie Film, Kino, Kinosaal, Vorfuehrung 3. ticketing: distributed-ticketing: Enthält die Tabellen Kontingent und Ticket. Für die integrierten tests (test-integrated) gibt es ein weiteres Schema, das alle tabellen enthält: distributed-test. Die Testdata-Loader wurden ebenfalls entsprechend angepasst	

45	**Modulith-1, Modulith-2 und Distributed**	Onion-Refactoring	Wir haben die Onion-Struktur in M1, M2 und D korrigiert und i.W. restriktiver gemacht:
			
			1. Ringe darf man nicht mehr überspringen
			   a. Insbesondere dürfen `domain.services` noch von `application` genutzt werden (nicht mehr aus `api`).
			   b. Das Domain-Model darf aber weiterhin auch von api rein lesend genutzt werden (daher dort nun alle Klassen immutable, dazu Änderungen an `Kontingent`).
			2. Die Transaktionskontrolle liegt ausschließlich in `application`.
			3. `api` nach "in" und "out" getrennt: In "in" sind Rest- und Web-Controller sowie die Event-Subscriber. In "out" sind die Event-Publisher.
			
			Dazu
			
			1. Architekturtests angepasst (bei der Gelegenheit auch geändert: Nur noch eine ausgehende Richtung der Abhängigkeiten sind modelliert / wird geprüft, bisher waren das redundant die ausgehenden *und* die eingehenden Abhängigkeiten).
			2. Neue Services in `application` erstellt. Diese sind nun mit `@Transactional`-annotiert.
			3. In Distributed: Event-Nutz/Metadaten werden (wie bisher) in `application` aufbereitet, dem Publisher übergeben (der aber nun ein Interface implementiert, das in `application` liegt ähnlich Dao-Interface und Implementierung in `persistence`.
			4. `apicontract` umbenannt in `api_contract` (analog zu `api_in` und `api_out`)
			5. `core` abgeschafft, `application` und `domain` liegen "eins höher"
			
			Bewusst nicht gemacht:
			
			1. Logging nur in Application erlauben. (Das wäre sehr aufwendig; außerdem ist es auch in API und ggf. in Domain mal sinnvoll zu loggen)	