WODSS

Event Planner

Autoren: Andreas Gassmann, Jonas Frehner, Lukas Schönbächler

n|w University of Applied Sciences and Arts Northwestern Switzerland

FHNW
Schweiz
März 27, 2017

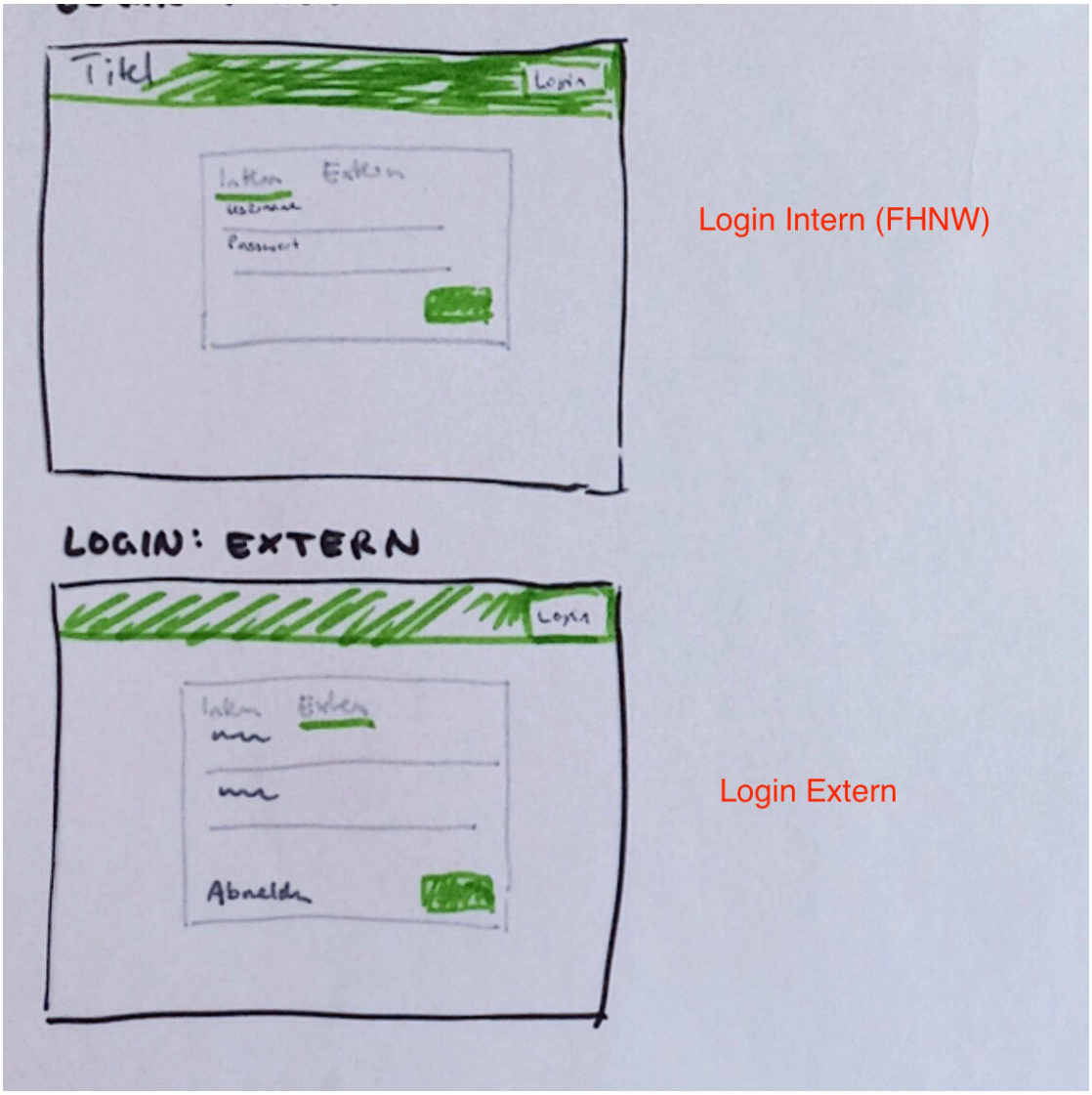
Abstract

Das vorliegende Projekt wurde im Rahmen des Moduls "wodss" der FHNW realisiert. Es wurde eine Plattform entwickelt, mit deren Hilfe auf einfache Art und Weise CS-Seminare und Anmeldungen zu den entsprechenden Anlässen verwaltet werden können. Die Lösung wurde mit Hilfe von Angular2 und Ionic im Frontend und SpringBoot im Backend realisiert. Architektonisch basiert das Projekt auf der Serverseite auf so genannten Microservices.

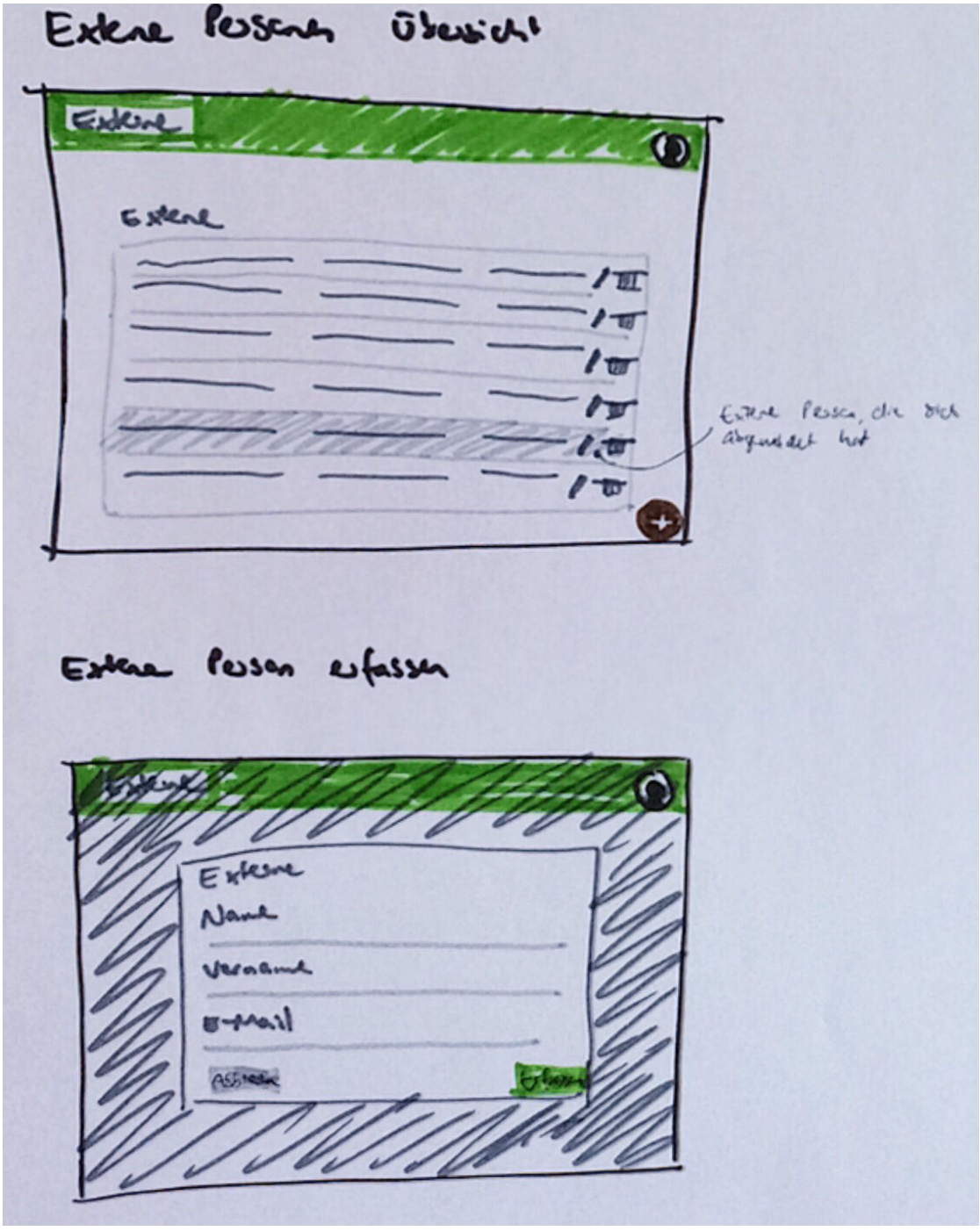
Contents

1 Mockups

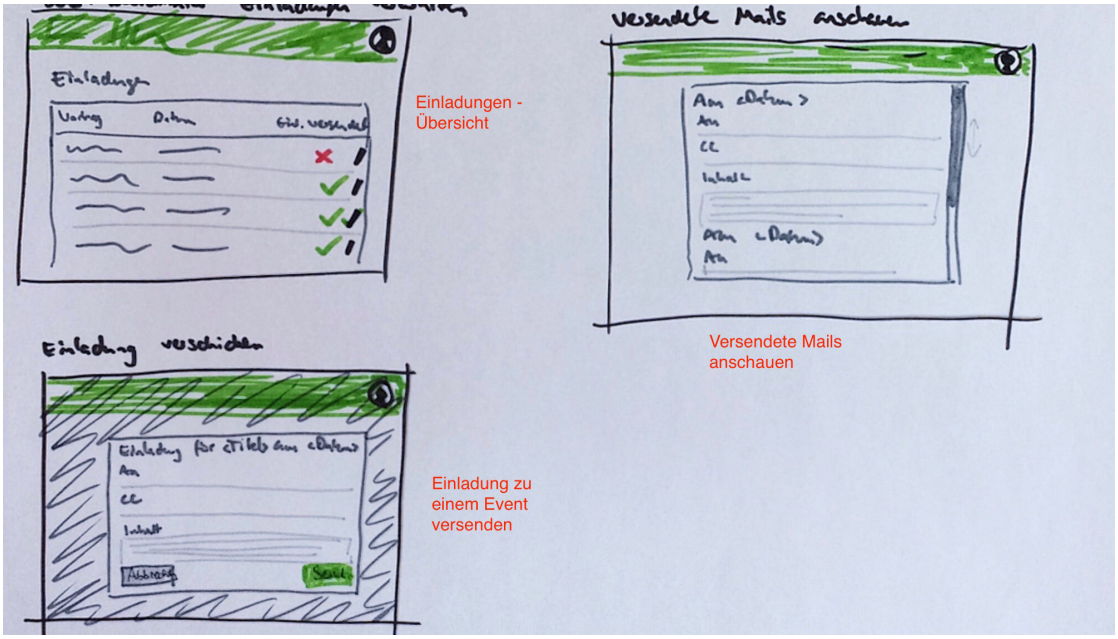
1.1 Login



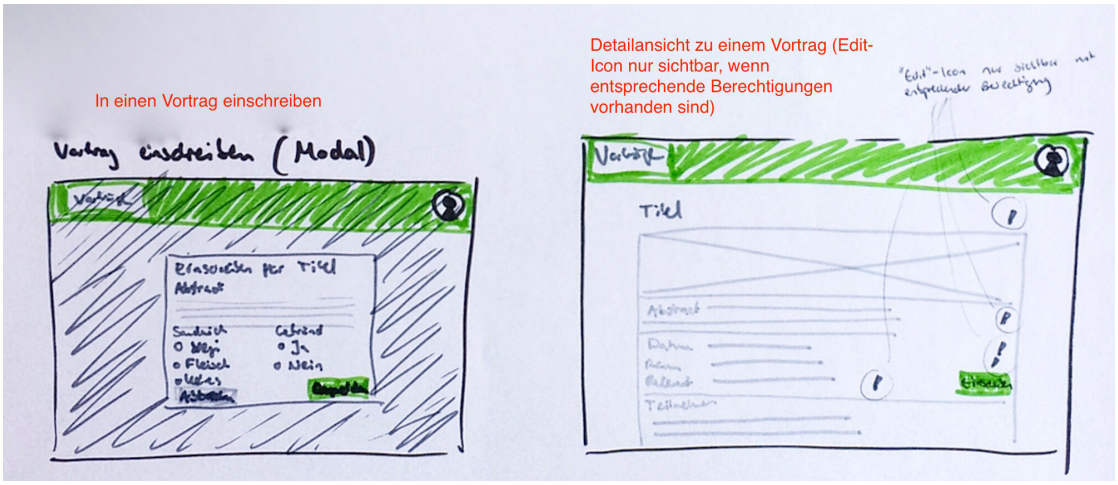
1.2 Übersicht


Externe Personen Übersicht
Externe
Externe
Externe Person, die sich abgemeldet hat
Externe Personen erfassen
Externe
Externe
Name
Vorname
E-Mail

1.3 Detailansicht



1.4 Einschreiben



2 Architektur - Backend

Wie bereits in der Einleitung erwähnt, implementierten wir dieses Projekt als Microservices. Wir verwenden dazu "Eureka", eine von Netflix entwickelte Library (siehe auch: https://github.com/netflix/eureka). Die Applikation lässt sich in fünf Microservices unterteilen, welche im nachfolgenden Abschnitt genauer beschrieben werden. Die Microservices kommunizieren per REST untereinander.

2.1 Schichtenarchitektur

2.1.1 Webschicht

Die Webschicht wird automatisch durch den RepositoryRestResource bereitgestellt. Das heisst, die betreffenden RepositoryRestResourcen stellen automatisch die so annotierten JPA-Entitäten via REST-Schnittstelle bereit.

2.1.2 Businesschicht

Da das vorliegende Projekt eine relativ einfache Business-Schicht aufweist (vielfach handelt es sich um einfache GET/POST/PUT/DELETE auf bestehenden Entitäten), wird diese ebenfalls von den RepositoryRestResourcen wahrgenommen. Wo eine etwas ausgefeilterte Logik notwendig war (z.B. beim Login oder bei der Passwort-Zurücksetzen-Funktionalität) wurden eigene RestController erstellt (wie beispielsweise das Login-Repository).

2.1.3 Datenbankschicht

Die Datenbankschicht wurde mit Hilfe von JPA-Repositories realisiert. Für eine genauere Diskussion zum Aufbau der Persistenzschicht siehe unterkapitel Datenbank.

2.2 Microservices

2.2.1 Registry

Die Registry stellt das "Bindeglied" zwischen den verschiedenen Microservices dar. Die Microservices verbinden sich auf die Registry um dort die URLs der anderen Microservices zu bekommen. Dadurch wird es möglich die Applikation auf mehreren Servern verteilt laufen zu lassen. Im Falle einer bestehenden Eureka Installation/Architektur muss dieser Microservice nicht nochmals zusätzlich gestartet werden (bestehende Registry kann verwendet werden). Durch die Verwendung des "Loadbalanced" RestTemplates lösen Microservices die logischen Namen anderer Microservices selbstständig mit Rückgriff auf die Registry zu korrekten IP-Adressen auf.

2.2.2 Eventmanagement

Der Microservice Eventmanagement ist der zentrale Service unserer Applikation. Er beinhaltet die Persistenzschicht mit allen Entitäten, sowie die Businesslogik der Anwendung. Zudem werden alle Zugriffe auf die Daten und die Logik via diesem Service getätigt. Hierfür werden die CRUD Operationen über eine REST-API zur Verfügung gestellt (mittels den oben erwähnten RepositoryRestResource-Controllern). Bestimmte Endpunkte stehen dabei nur nach erfolgreicher Authentifizierung und Autorisierung zur Verfügung. Die komplette Schnittstellendokumentation ist im Anhang (Swagger Schnittstellendokumentation) aufgeführt.

2.2.3 Frontend

Dieser Microservice stellt nur einen Container bereit, der dazu dient, statische Inhalte wie HTML-/CSS-Files, Bilder und so weiter zur Verfügung zu stellen. Er enthält keinerlei Business-Logik und speichert keine Benutzerdaten. Konkret wird der Build Folder unserer Ionic-App in den assets/static Order kopiert. Sollte bereits eine bestehende Webserver Infrastruktur zur Verfügung stehen (z.B. Apache oder Nginx), können diese Assets auch ohne zusätzlichen Microservice gehostet werden.

2.2.4 Mailer

Der Mailer ist ein simpler Mailservice, welcher eine API zum Versenden von Mails bietet. Diese API wird durch ein statisches Token geschützt, das alle Aufrufer mitschicken müssen, um Mails versenden zu können. Da der Token nur intern (via application.properties-Files der entsprechenden Microservices) bekannt ist, werden ausschliesslich interne (von anderen Microservices) Anfragen entgegengenommen.

2.2.5 Scheduler

Der Scheduler-Service führt zu bestimmten Zeiten verschiedene Tasks aus. Ein Beispiel hierfür wäre, unreferenzierte Mediendateien in der Nacht zu löschen, Erinnerungsmails zu versenden oder Events aufgrund verschiedener Kriterien zu archivieren.

2.3 Sicherheit

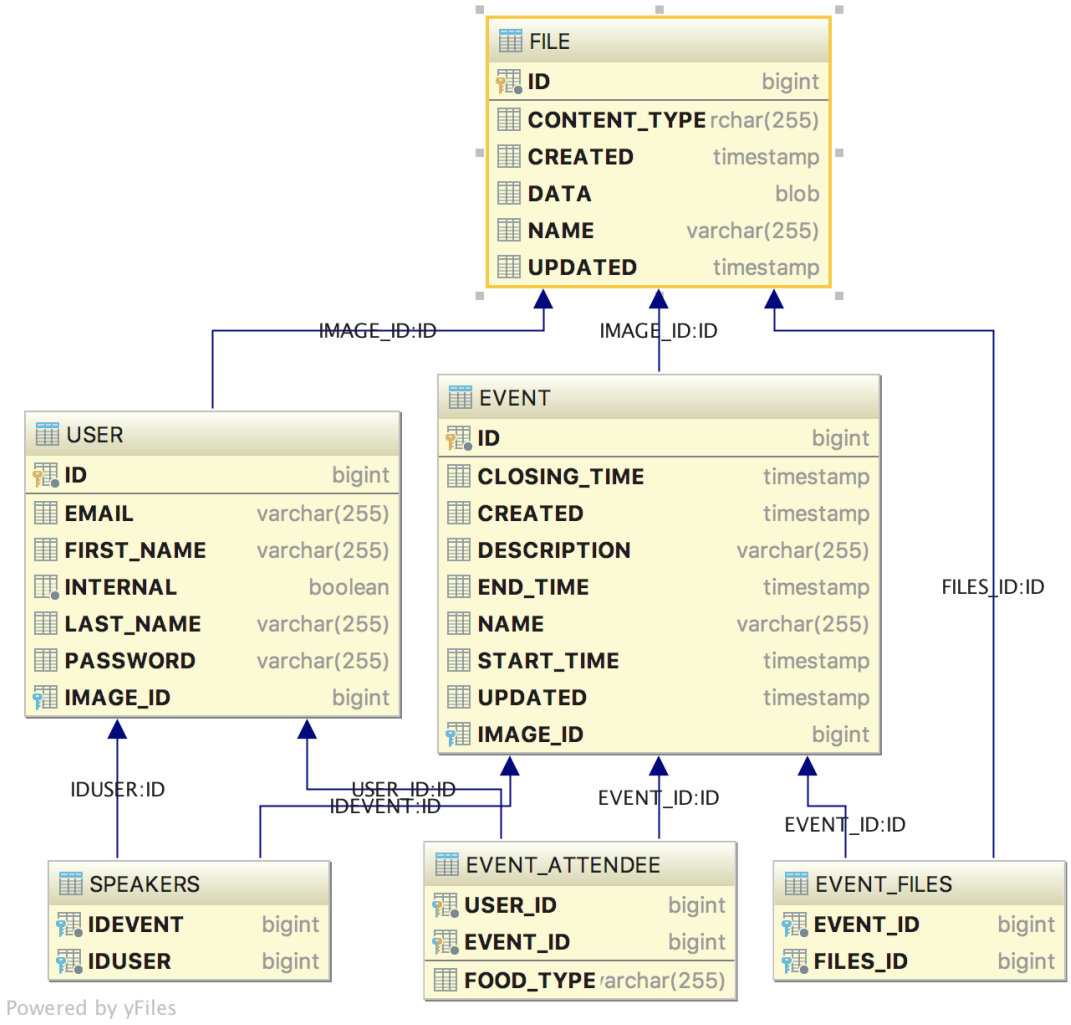
Mithilfe von Pac4J können Permissions und Rollen definiert werden. Abhängig von der Rolle, erhält der Benutzer zusätzliche Privilegien. Ein normaler Besucher kann lediglich Events ansehen und nichts editieren. Wir haben folgende Rollen identifiziert:

1. Besucher
2. Angemeldeter Gast (kann sich zusätzlich einschreiben)
3. Referent (kann zusätzlich den eigenen Event bearbeiten und Dateien anhängen)
4. Koordinator (kann alle Entitäten bearbeiten und löschen)

Rollenkonzept
DirectAuth
Cookie
Cors
Csrf

2.4 Datenbank

Um für die Datenintegrität garantieren zu können, werden alle Entitäten wie oben durch einen einzigen Microservice verwaltet (https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/).



3 Architektur - Frontend

3.1 Ionic

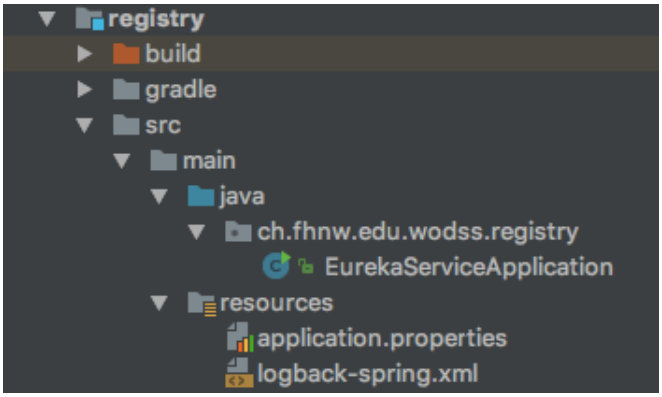
Pages usw.

4 Projektaufbau - Backend

Für alle Microservices wurde jeweils ein eigenes Sub-Projekt erstellt, die jeweils aus einem build-, gradle- und src-Ordner bestehen. Für jeden Microservice existiert ein eigenes build.gradle, das die Projektspezifischen Abhängigkeiten enthält. Der Root-Folder enthält alle Microservices, sowie ein Ordner "ionic", der allen Frontend-Code enthält.
Zudem haben wir vier verschiedene .sh-Files erstellt, mit denen die vier Microservices Eventmanagement, Registry, Mailer und Scheduler je einzeln gestartet werden können. Dazu ein .sh-File mit dem alle erwähnten Services zusammen gestartet (run.sh) werden können, gestoppt werden können (kill.sh) oder gestoppt und gleich neu gestartet werden können (restart.sh).
Nachfolgend gehen wir kurz auf den internen Aufbau der einzelnen Microservices ein. Dabei werden wir nur den Aufbau des src-Ordners präsentieren, da sich der Build- und Gradle-Ornder jeweils sehr ähnlich zeigen. Zusätzlich werden auch die relevanten Klassen-Diagramme (in UML) pro Microservice präsentiert.

4.1 Registry

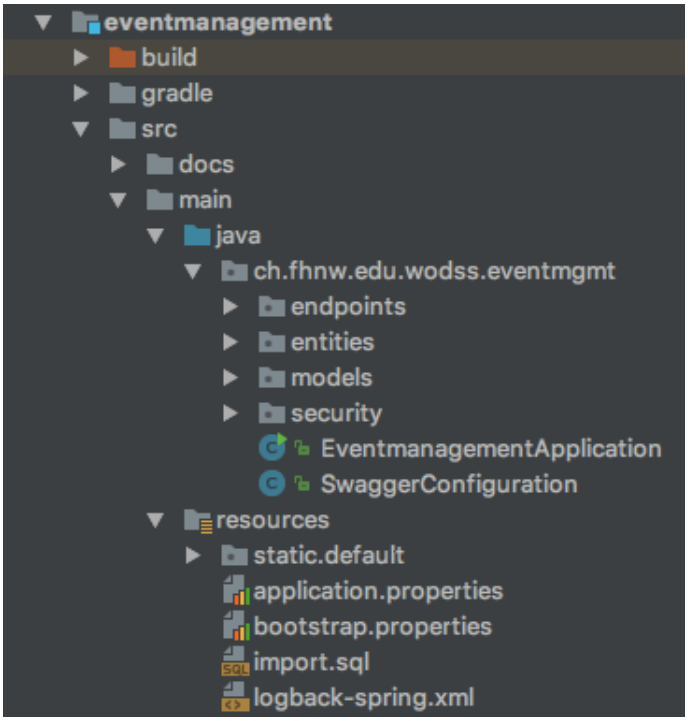
Die Registry hat folgende sehr simple Projektstruktur:



Der Grund für die schlanke Struktur liegt auf der Hand: da sich die Verwendung der Eureka-Library in diesem Projekt grösstenteils auf Annotationen beschränkt besteht die Registry nur aus einer einzigen Klasse, die mittels "EnableEurekaServer" und "SpringBootApplication" annotiert wurde. Die restlichen Microservices verwenden als Klienten die Annotation "EnableDiscoveryClient".

4.2 Eventmanagement

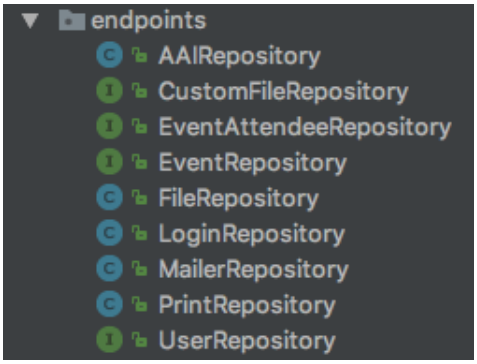
Dieser Microservice ist der komplexeste der fünf. Die grobe Struktur präsentiert sich wie folgt:



Die Swagger-Configuration-Klasse übernimmt, wie der Name erahnen lässt, die Konfiguration der Swagger-Dokumentation der REST-Schnittstelle. Die Klasse Eventmanagement-Application dient der Initialisierung des Microservices. Nachfolgend werden alle Ordner kurz präsentiert und deren Inhalt etwas genauer beschrieben.

4.2.1 Endpoints

Im Folder "endpoints" sind alle Interfaces und Klassen enthalten, die entweder mittels RepositoryRestResource (sofern die Klasse auf einer konkreten JPA-Entität basiert) oder RestController (sofern die Klasse zusätzliche Geschäftslogik bereitstellt) annotiert sind. Die Struktur sieht folgendermassen aus:



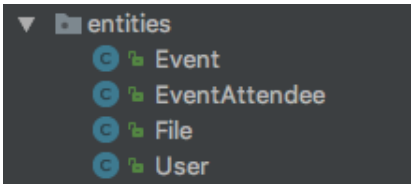
Relativ schnell erkennt man die zusätzliche Geschäftslogik, die zusätzlich zu den RepositoryRestResource-Controllern programmiert wurde und die nicht durch reine Datenzugriffe verarbeitet werden konnten:

1. AAI-Repository - verarbeitet Login via Switch-AAI
2. FileRepository - bietet Methoden zum Fileupload und -display
3. LoginRepository - bietet Methoden zum Login und zurücksetzen von Passwörtern
4. MailerRepository - bietet Methoden zum Versenden von Einladungmails
5. PrintRepository - bietet Methoden zum Zugriff auf Druckgerecht-aufbereitete Daten

Die restlichen Interfaces beinhalten zum einen die vom JPA-Repository zur Verfügung gestellten Methoden für den Datenzugriff. Zusätzlich dazu wurden, je nach Bedarf, weitere Methoden für die Datenmanipulation erstellt.

4.2.2 Entities

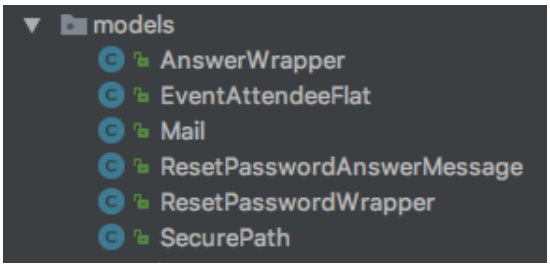
Der Ordner "entities" präsentiert sich wie folgt:



Dieser Ordner enthält die tatsächlichen JPA-Entitäten die schlussendlich in der Datenbank als konkrete Tabellen auftauchen (dazu kommen weitere von JPA erstellte Tabellen wie z.B. "Speakers", siehe dazu auch das Kapitel über die Datenkbank). Die Entitäten definieren gemäss JPA die verschiedenen Attribute der jeweiligen Objekte.

4.2.3 Models

Der Folder "model" enhält alle weiteren Objekte, die für das Funktionieren der Applikation notwendig sind, aber nicht persistiert werden sollen:

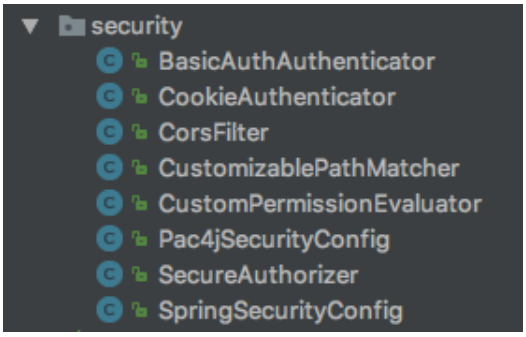


Eine kurze nähere Beschreibung der verschiedenen Models:

1. AnwerWrapper
2. EventAttendeeFlat - Container für die druckgerecht aufbereiteten Daten für eine Listenansicht aller EventAttendees (siehe obige Beschreibung des PrintRepository)
3. Mail - Container für den Datenaustausch mit dem Frontend (alles bezüglich Mail) und dem Mailer-Microservice
4. ResetPasswordAnswerMessage - Daten-Container für den Rückgabewert einer ResetPassword-Anfrage (wird im LoginRepository verwendet)
5. ResetPasswordWrapper - Container für die Datenübertragung vom Front- zum Backend während einer ResetPasssword-Anfrage
6. SecurePath - Container der intern verwendet wird, um abgesicherte Pfade zu speichern (wird nur vom CustomizablePathMatcher verwendet)

4.2.4 Security

Der Ordner "security" schlussendlich hat folgenden Inhalt:



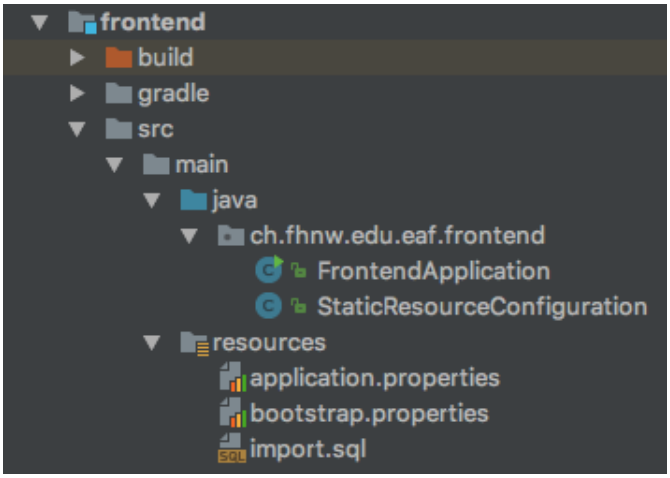
Eine kurze nähere Erläuterung der verschiedenen Klassen:

- SpringSecurityConfig - einer der beiden zentralen SecurityConfigs (die andere wäre die Pac4jSecurityConfig). Enthält grundlegende Sicherheitseintstellungen wie z.B. csrf-Schutz
- Pac4jSecurityConfig - die zweite SecurityConfig. Setzt den CookieAuthenticator, den DirectBasicAuthClient, den Authorizer und legt fest, welche Pfade wie gesichert werden sollen.
- BasicAuthAuthenticator - verantwortlich für die Authentifizierung via BasicAuth. Verleiht den Usern die ihnen zustehenden Rollen.

- CookieAuthenticator - validiert einen User anhand eines Cookies, d.h. falls der User bereits eingeloggt ist.
- SecureAuthorizer - Setzt verschiedene Sicherheitsrelevante HTTP-Header (wie beispielsweise korrekte XSS-Headers usw.)
- CustomizablePathMatcher - Container für ein Array aus "SecurePaths" (siehe den Unterabschnitt zu "Models"). Matched auf Pfade (wie z.B. /api/login) und gibt die Informationen zurück, welche HTTP-Methoden auf diesen Pfaden erlaubt sind.
- CustomPermissionEvaluator - bestimmt, ob die korrekten Permissions vorhanden sind, um bestimmte Aktionen auszuführen. Der CustomPermissionEvaluator wird auf den Interfaces im Ordner "Endpoints" aufgerufen, wenn geprüft werden soll, ob ein User die korrekten Permissions hat, um die mit "hasPermission(...)" annotierte Methode auszuführen.
- CorsFilter - liefert die Funktionalität, um Cors korrekt zu verarbeiten. Wird im vorliegenden Projekt nicht verwendet, da alle Microservices intern gestartet werden bzw. hinter ein und demselben Endpoint (cs.fhnw.technik.ch). Cors stellt somit kein Problem dar. Kann bei Bedarf eingeschaltet werden.

4.3 Frontend

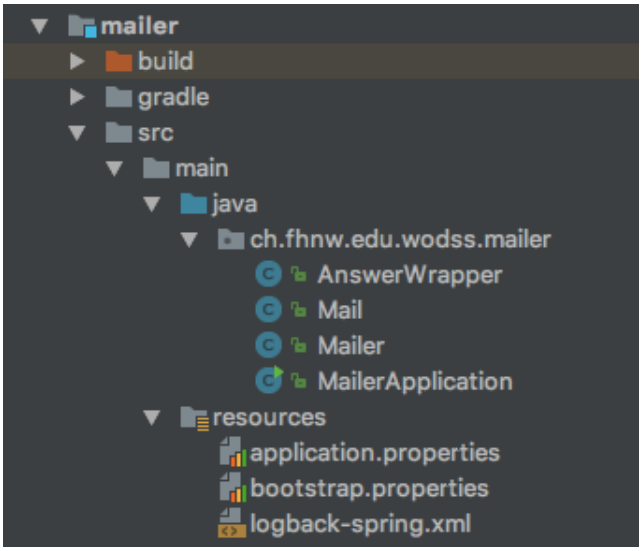
Der Folder Frontent hält einen relativ bescheidenen Inhalt:



Die Klasse FrontendApplication dient dazu, den Microservice zu starten, die Klasse StaticRessourceConfiguration dazu, statische Ressourcen (also die HTML-, Javascript- und CSS-Dateien) auszuliefern.

4.4 Mailer

Der Mailer-Microservice ist ebenfalls relativ klein:

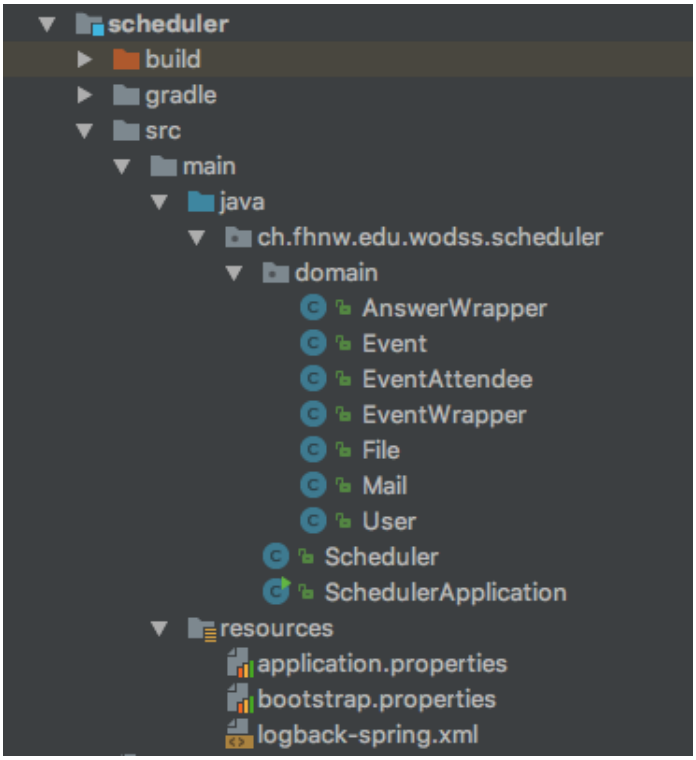


Eine kurze Erläuterung dazu:

- AnswerWrapper - DatenContainer für die Antworten an die aufrufenden Services
- Mail - DatenContainer für den Input, den die aufrufenden Services an den Mailer schicken
- Mailer - enthält den zentralen Endpoint für den Mailer (/api/send) an den ein Objekt vom Typ Mail geschickt wird. Das Mail enthält den Text mit eventuellen Platzhaltern und je einem keys- und values-Array mit den Werten, mit denen die Platzhalter ersetzt werden sollen. Der Mailer ersetzt die Platzhalter eigenständig und verschickt das Mail an die im Mail-Container gelieferten Adressen. Er ist somit ein relativ "simpler" Service, der die übergebenen Infos nimmt und mehr oder weniger einfach weiterschickt.
- MailerApplication - dient dazu, den Microservice zu starten

4.5 Scheduler

Der Scheduler-Microservice besteht aus zwei Klassen und einem Unterordner "domain":



Die SchedulerApplication dient dem Start des Service. Die Klasse "Scheduler" selbst enthält die Logik der Tasks, die in regelmässigen Abständen auszuführen sind.
Eine kurze Erläuterung des Unterordners "domain":

- AnswerWrapper - DatenContainer, der bei einem Post auf den Mail-Microservice zurückgeliefert wird. Wird verwendet, um die Antwort auszulesen.
- Event - Wird verwendet, um die korrekten Daten für regelmässig zu versendende Mails zu erhalten. Die so erhaltenen Informationen werden via keys- und values-String-Arrays innerhalb des Containers "Mail" dem Mailer übergeben (siehe auch Beschreibung des Mailer-Service oben).
- EventAttendee - Wird ebenfalls verwendet, um die korrekten Daten für regelmässig zu versendende Mails zu erhalten. Die so erhaltenen Informationen werden via keys- und values-String-Arrays innerhalb des Containers "Mail" dem Mailer übergeben (siehe auch Beschreibung des Mailer-Service oben).
- EventWrapper - DatenContainer der nur jene Attribute eines Events enthält, die in einer "scheduled" Task aktualisiert werden sollen.
- File - Nimmt dieselbe Funktion ein wie ein Event oder ein EventAttendee (siehe oben).
- Mail - Container für den Datenaustausch mit dem Mailer-Microservice.

- User - Nimmt dieselbe Funktion ein wie ein Event, ein EventAttendee oder ein File (siehe oben).

5 Projektaufbau - Frontend

6 Technologien

6.1 Frontend

Nach dem Start mit Angular 2, entschieden wir uns, das Framework "Angular Material 2" zu verwenden. So wollten wir sicherstellen, dass das Projekt durchgängig eine einheitliche Designsprache verwendet und zudem eine Gestaltung verwendet wird, die aus bekannten Apps (z.B. Google Docs) bereits vielen Usern bekannt sein würde. Das vorliegende Projekt wäre somit intuitiv nutzbar. Relativ schnell realisierten wir, dass "Angular Material 2" aber noch einige Lücken aufwies (beispielswiese fehlten uns einige Standardkomponenten zudem liess die Dokumentation teilweise sehr zu wünschen übrig). Ausserdem war der Aufwand die Seite Mobile-tauglich zu machen grösser als erwartet. Aus diesem Grund wechselten wir nach kurzer Zeit auf Ionic 2. Da Ionic 2 auf Angular 2 aufbaut, war der Wechsel schnell vonstatten gegangen. Mithilfe der neuen "Sidemenu-Komponente" konnte die Webseite ohne weitere Probleme für Desktops optimiert werden.
Als Technologien verwendet Ionic 2 wie erwähnt Angular 2 mit Typescript, SASS und HTML.

6.2 Backend

Wie von der Projektbeschreibung vorgeschrieben, verwenden wir Java mit Spring Boot im Backend. Zusätzlich verwenden wir die spring-boot-data Erweiterung, welche uns das einfache erstellen von Rest Repositories ermöglicht.
Für den Mailer wurde die Java Mail API, bzw. die Implementierung von Spring Boot verwendet. Um die Informationen in den Platzhaltern von Mailtexten zu ersetzen verwendeten wir die kleine Library "StringTemplate".

6.2.1 Authentifizierung und Autorisierung

Für die Authentifizierung verwenden wir Pac4J und Spring Security.

7 Design-Entscheide

Anfangs wollten verschiedene neue Technologien wie z.B. Elm, GraphQL und Microservices testen. Es stellte sich aber schnell heraus, dass es dabei einige Probleme gibt, welche im folgenden Abschnitt erläutert werden.

1. Elm
   Elm ist eine relativ neue funktionale Sprache, die zu Javascript kompiliert. Der grosse Vorteil von Elm sind die Typsicherheit sowie der funktionale Aspekt. Es sollte dadurch beispielsweise keine Runtime-Fehler, die in Javascript an der tagesordnung sind, mehr geben. Es hat sich aber herausgestellt, dass das Ökosystem zwar bereits viele Funktionen bietet, wenn man aber genaue Anforderungen hat muss man teilweise Kompormisse eingehen. Weiter wird user Projekt durch die Wahl einer "neuen" Programmiersprache weniger wartbar.

2. GraphQL
   Durch die JPA-GraphQL ist das initiale Aufsetzen eines GraphQL Endpoints nach einigen Versuchen relativ gut gegangen. Leider fehlen der Library aber noch einige zentrale Features, weshalb man nicht komplett auf die REST-API hätte verzichten können. Anstelle von zwei verschiedenen Endpunkten haben wir uns schlussendlich aus gründen der Wartbarkeit für REST entschieden.

8 Lessons learned

Im Laufe unseres Projekts wurden wir mit einigen Problemen konfrontiert, die insgesamt einen grossen Teil unserer Zeit in Anspruch genommen haben.

1. Microservices
   asdfasdfasdfasdf
2. Pac4J
3. Angular
   Angular Material2 noch nicht ready, Ionic mehrheitlich ohne Probleme
4. ResourceRestController

Appendices

Swagger Schnittstellendokumentation
https://htmlpreview.github.io/?https://raw.githubusercontent.com/lukeisontheroad/simple_event_planner/master/docs/doc.html

Github Repository
https://github.com/lukeisontheroad/simple_event_planner

Microservices best practices
https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/

PAC4J
http://www.pac4j.org/