



IPA

NAME DER IPA

IPA von Niculin Steiner

Ergon Informatik AG
20. November 2024

Inhalt

I	Umfeld und Ablauf	4
1	Aufgabenstellung	5
1.1	Ausgangslage	5
1.2	Detaillierte Aufgabenstellung	6
1.3	Mittel und Methoden	7
1.4	Vorkenntnisse	7
1.5	Vorarbeiten	7
1.6	Neue Lerninhalte	8
1.7	Arbeiten in den letzten 6 Monaten	8
2	Projektaufbauorganisation	9
3	Benützte Firmenstandards	10
4	Arbeitsumgebung	11
4.1	Arbeitsplatz	11
4.2	Verwendete Tools	12
5	Versionierung und Sicherung der Arbeitsergebnisse	13
5.1	Git als Versionierungstool	13
5.2	Git im Zusammenspiel mit Gerrit	13
6	Projektmanagementmethode	14
6.1	IPERKA	14
6.1.1	Informieren	14
6.1.2	Planen	14
6.1.3	Entscheiden	14
6.1.4	Realisieren	14
6.1.5	Kontrollieren	15
6.1.6	Auswerten	15
6.2	Alternative Methode - Scrum	15
7	Arbeitsprotokoll	16
8	Zeitplan	26

II	Projekt	29
9	Informieren	30
9.1	Analyse	30
9.1.1	Futurae	30
9.2	Technische Referenzen	31
9.3	Anforderungen	31
9.3.1	Rest / Backend	31
9.3.2	SPA	32
9.3.3	Kundendokumentation	32
10	Planen	33
10.1	Arbeitspakete	33
10.1.1	Informieren	33
10.1.2	Planen	34
10.1.3	Entscheiden	35
10.1.4	Realisieren	36
10.1.5	Kontrollieren	37
10.1.6	Auswerten	38
10.1.7	Rahmenaufgaben	38
10.2	Lösungskonzept Backend	40
10.2.1	REST	40
10.2.2	Rollenlogik	42
10.2.3	Wichtige Klassen und Interfaces	42
10.3	Lösungskonzept SPA	43
10.3.1	Mockups	43
10.3.2	Lösungsvariante ohne Popup	45
10.3.3	Rollenlogik	46
10.3.4	Wichtige Komponenten und Services	46
10.3.5	Translation Keys	47
10.4	Testkonzept	47
10.4.1	Benötigte Testmittel	47
10.4.2	Wiremock	48
10.4.3	Unit-Tests	48
10.4.4	REST-Integration-Tests	48
10.4.5	UI-Integration-Tests	49
10.4.6	Manuelle Tests	50
10.5	Qualitätssicherungskonzept	52
11	Entscheiden	53
11.1	REST-Interface Backend	53
11.2	SPA-Design	54
12	Realisieren	55

12.1	Backend erweitern	55
12.1.1	Neuer REST-Endpunkt	55
12.1.2	Requests zu Futurae	57
12.1.3	Logging	62
12.1.4	Resultat	63
12.1.5	Tests	63
12.1.6	Testen der Access Control	67
12.2	SPA erweitern	67
12.2.1	Request zur IAM API	68
12.2.2	Darstellung im UI	68
12.2.3	i18n Translation Keys	71
12.2.4	UI Integration-Tests	73
12.3	Kundendokumentation	76
12.3.1	Kapitel 18.5 Airlock 2FA Configuration erweitern	76
13	Kontrollieren	77
13.1	Durchführen der Tests	77
13.1.1	Unit-Tests	77
13.1.2	REST-Integration-Tests	78
13.1.3	UI-Integration-Tests	79
13.1.4	Manuelle Tests	79
13.2	Qualitätssicherung	80
13.2.1	Manuelle Refactorings	80
13.2.2	Automatisierte Code-Analyse	80
14	Auswerten	82
	Abbildungsverzeichnis	82

Teil I

Umfeld und Ablauf

1 Aufgabenstellung

In diesem Kapitel sind die Aufgabenstellung und die Rahmenbedingungen aufgeführt. Der grösste Teil des Inhalts stammt aus der originalen Aufgabenstellung.

1.1 Ausgangslage

Airlock Identity and Access Management (IAM) ist ein bestehendes, in unserer Abteilung entwickeltes Produkt, das unter anderem Logins (Authentisierungen) ermöglicht. Eine weitere Funktionalität eines IAMs ist der Admin-Bereich (adminapp). Airlock IAM unterstützt unterschiedliche Stufen von Administratoren, um beispielsweise Mitarbeitenden im Support oder an einem Kundenshalter spezifisch eingeschränkten Zugriff für die Verwaltung von Usern zu erlauben.

Airlock 2FA erlaubt es, nebst beispielsweise Usernamen und Passwort, einen weiteren Authentisierungsfaktor zu verwenden. Üblicherweise wird dazu die Airlock 2FA App auf dem Smartphone installiert und aktiviert.

Es gibt mehrere Möglichkeiten, wie Kund:innen für den eigenen Login die Airlock 2FA aktivieren können. Ein Weg ist beispielsweise über einen Brief mit einem QR Code, welchen Kund:innen dann mit der Airlock 2FA App scannen können. Die Aktivierung ist auch über einen 16-stelligen Aktivierungscode möglich.

Immer wieder kommt es vor, dass Kund:innen Unterstützung bei der Aktivierung von Airlock 2FA benötigen und sich telefonisch beim Firmen-Helpdesk oder am physischen Schalter melden. Damit das Support- oder Schalterpersonal der Kundschaft helfen kann, die Airlock 2FA zu aktivieren, braucht es eine Möglichkeit, den 16-stelligen Aktivierungscode für den spezifischen User anzuzeigen.

Bisher gibt es in Airlock IAM noch kein Feature, damit der Administrator-Bereich solche 16-stelligen Aktivierungscodes pro User anzeigen kann.

1.2 Detaillierte Aufgabenstellung

Ziele

- UC1: Helpdesk kann Kunden am Telefon helfen, ein Gerät zu aktivieren.
- UC2: Schaltermitarbeiter kann Kunde am Schalter helfen, ein Gerät zu aktivieren
- UC3: Es soll möglich sein, den Zugriff auf die userspezifischen 16-stelligen Aktivierungscodes nur für bestimmte Administratoren-Rollen (bspw. Rolle Helpdesk) freizugeben, damit nicht alle Administratoren sich den 16-stelligen Aktivierungscode anzeigen lassen können.
- UC4: Im User Activities Logfile des spezifischen Users soll geloggt werden, welcher Administrator-Account zu welchem Zeitpunkt den 16-stelligen Aktivierungscode angezeigt hat, damit im Nachhinein nachvollziehbar ist, welche Administratoren je Zugriff auf den Aktivierungscode hatten.

Weitere Anforderungen

- Der Code soll auf Knopfdruck in der Adminapp angezeigt werden. Dabei sind UI-Komponenten zu verwenden, die an anderen Stellen in der Adminapp auch schon verwendet werden. Eine mögliche Lösung ist ein SPA Popup (kein Browser Popup) mit einem 'Schliessen' Knopf.
- Neue Plugins oder Plugin Properties sollen einen klaren und vollständigen Hilfetext haben.

Erwartete Artefakte

Nebst der IPA Dokumentation werden diese technischen Artefakte erwartet:

- Sinnvolles Slicing und Anzahl von Gerrit Changes mit der implementierten Lösung und Git Kommentaren, die unseren internen Konventionen entsprechen. Der Kandidat entscheidet selbst, wie viele Gerrit Changes sinnvoll sind. Er hat dabei zu beachten, dass die Changes aufeinander aufbauen sollten und «verdaubare» Review-Größen haben.
- Beschreibung wie das neue Feature konfiguriert werden kann in der Airlock IAM Kundendokumentation. Dazu soll das Kapitel 18.5 Airlock 2FA configuration sinnvoll erweitert werden. Die angepasste Kundendokumentation soll auf Englisch und in den restlichen PDF-Unterlagen enthalten sein (es ist nicht nötig, mit unserem Kundendokumentation-Tool SMC zu arbeiten).

Abgrenzung

- Administratoren können pro User bereits Aktivierungsbriefe erstellen oder anfordern. An dieser Logik soll im Rahmen dieses Issues nichts verändert oder erweitert werden.

1.3 Mittel und Methoden

Es wird auf dem aktuellen Stand der Entwicklung von Airlock IAM 8.4 aufgebaut.

REST Technologien

- Java(Guice als Dependency Injection Framework), JSON, JUnit
- Jackson, Jersey, Guice
- REST Integration Tests

SPA Technologien

- Angular (Typescript/RXJS)
- Bootstrap (HTML/CSS/SASS)
- Selenium UI Testing

Wichtigste Tools

- IntelliJ(IDE)
- Gerrit + Git (SCM)

1.4 Vorkenntnisse

Der Kandidat war involviert in die Implementation von SPA und REST Features im Bereich IAM Protected Self-Service.

Das Grundgerüst der SPA und REST Endpunkte ist bekannt.

1.5 Vorarbeiten

Der Kandidat hat für die Probe-IPA keine vorbereiteten Tätigkeiten erarbeitet, hat sich aber in das Thema Airlock 2FA eingelesen.

1.6 Neue Lerninhalte

Erfahrung bei der selbständigen Entwicklung einer produktrelevanten Erweiterung unter realistischen Bedingungen.

- Futurae API: <https://www.futurae.com/docs/api/auth/>
- IAM Kunden Dokumentation: <https://docs.airlock.com/iam/8.3/>

1.7 Arbeiten in den letzten 6 Monaten

In den letzten sechs Monaten hat der Kandidat Erfahrungen in folgenden Bereichen gesammelt:

- OAuth 2.0 / OpenID Connect Consent Management Self-Service, SPA und REST
- Have I Been Pwnd Scriptable Step, 3rd Party REST API, Lua
- HTTP Cache Control Konfiguration im Zusammenhang mit JWKS REST End-point

2 Projektaufbauorganisation

Die folgenden Personen sind in dieses Projekt involviert:

Person	Rolle	Aufgabe/Verantwortung
Niculin Steiner	Kandidat (K)	Umsetzen der Facharbeit
Pascal Knecht	Verantwortliche Fachkraft (VF)	Facharbeit begleiten, technische Fragen beantworten, Bewertung der Facharbeit
Bernd Lienberger	Hauptexperte (HEX)	IPA bezogene Fragen beantworten, Entscheiden bei auftretenden Problemen, Besuchstermine festlegen, Fachgespräch leiten, Bewertung der Facharbeit

3 Benützte Firmenstandards

Für die Umsetzung dieser Probe-IPA wurde für den Bericht und den Zeitplan eine Vorlage verwendet.

4 Arbeitsumgebung

In diesem Abschnitt wird der Arbeitsplatz und die Umgebung, während der Probe-IPA, des Kandidaten beschrieben.

4.1 Arbeitsplatz

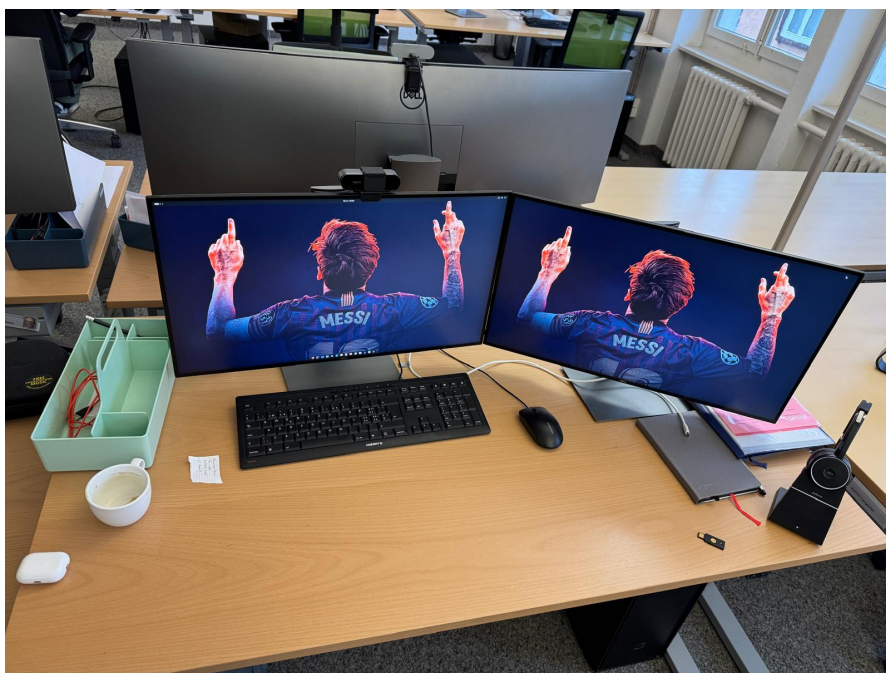


Abbildung 4.1: Arbeitsplatz während der Probe-IPA

Da seit der Mitarbeit im IAM nie im Homeoffice gearbeitet wurde, findet auch die Probe-IPA wie gewohnt vor Ort statt. Der Desktop PC mit dem Betriebssystem Linux (Distribution Ubuntu) ist für die maximale Effizienz mit 2 Bildschirmen verbunden. Um im Grossraumbüro möglichst ungestört zu arbeiten, liegen dem Kandidaten ein Paar AirPods Pro mit Noise Cancelling vor.

4.2 Verwendete Tools

Die folgende Tabelle zeigt einen Überblick der wichtigsten Tools, welche für die Umsetzung der Probe-IPA verwendet wurden:

Tool	Einsatzzweck	Link
IntelliJ Ultimate	Entwicklungsumgebung, zur Entwicklung des Features	https://www.jetbrains.com/de-de/idea/
TeXstudio	Entwicklungsumgebung für Latex, mit welchem die Dokumentation geschrieben wurde	https://www.texstudio.org/
Gerrit	Quellcode Verwaltung	https://www.gerritcodereview.com/
Git	Versionskontrollsystem	https://git-scm.com/
Jenkins	Automatisierte Testruns	https://www.jenkins.io/
Outlook	Termin für den Expertenbesuch im Blick behalten	https://www.microsoft.com/de-ch/microsoft-365/outlook
Postman	Testen der eigenen API und der API von Futurae	https://www.postman.com/
LibreOffice	Erstellen und warten des Zeitplans	https://de.libreoffice.org/
Github	Backup und Versionierung des Berichts und des Zeitplans	https://github.com/
Gliffy	Diagramme & Mockups erstellen	https://www.gliffy.com/
Mockito	Mocking für Unit-Tests	https://site.mockito.org/
JUnit5	Testframework für Java	https://junit.org/junit5/
Selenium	Framework für automatisierte Softwaretests von Webanwendungen	https://www.selenium.dev/
Wiremock	Simulieren des Futurae Server für Unittests	https://www.wiremock.io/

5 Versionierung und Sicherung der Arbeitsergebnisse

Die Arbeitsergebnisse sollten gesichert werden. Damit, im Falle eines unerwarteten Ausfalls während der Probe-IPA, z.B. des Rechners, von einem anderen Gerät wieder auf den letzten Stand zugegriffen werden kann. Zu dem sollte es generell möglich sein jeder Zeit auf einen älteren Stand zurück zukommen. Dies gilt natürlich für den Quellcode und den Bericht.

5.1 Git als Versionierungstool

Für die Versionierung der Arbeitsergebnisse wurde Git verwendet. Git ist weit verbreitet und ist auch aus der Schule und diversen anderen Projekten bekannt. Es wird verwendet um Änderungen am Code zu verfolgen und erstellt dabei eine Versionshistorie.

Zur Sicherung werden die Zwischenstände regelmässig in das jeweilige Remote-Repository gepusht. Das Repository für den Bericht liegt auf dem Ergon Github Account des Kandidaten(<https://github.com/niculinstei/probe-ipa-doku.git>). Der Quellcode welcher das Produkt erweitert liegt in einem Repository auf Gerrit.

5.2 Git im Zusammenspiel mit Gerrit

Der Quellcode liegt in einem Git-Repository auf Gerrit. Gerrit dient dazu als Review und Code Management Tool. Im Vergleich zur «gewöhnlichen» Entwicklung mit Git, bei der man für neue Features, Branches mit Commits erstellt, arbeitet man bei Gerrit sozusagen auf Commitbasis. Pusht man einen neuen Commit auf Gerrit, erstellt dieser ein neues «Changeset» mit einem Patchset. Gibt es nun weitere Änderungen werden diese einfach amandiert, dies erstellt dann ein weiteres Patchset auf diesem Changeset. Für grössere und komplexere Änderungen können auch aufeinander aufbauende Changesets erstellt werden. Für die Probe-IPA wurde zuerst ein Change für die Implementation des Backends erstellt. Danach ein zweiter Change mit der Implementation des Frontends, welcher auf dem ersten Change aufbaut. Dies macht das Feature übersichtlicher, was vor allem für den Reviewer viel angenehmer ist.

6 Projektmanagementmethode

In diesem Kapitel wird die Projektmanagementmethode IPERKA beschrieben. Es wird dargelegt wieso diese Methode gewählt wurde und was die Vor und/oder Nachteile daran sind.

6.1 IPERKA

Für die Probe-VA wurde IPERKA als Projektmanagementmethode gewählt. Sie eignet sich gut für kleine Projekte. Sie lassen sich damit einfach und strukturiert planen sowie umsetzen. Die IPERKA Methode setzt sich aus folgenden 6 Schritten zusammen:

6.1.1 Informieren

Der erste Punkt bei IPERKA ist das Informieren. Dabei wird sich ein Überblick über das Projekt / den Projektauftrag verschaffen. Es gilt zu klären was genau der Auftrag ist, und ob alle Informationen vorhanden sind.

6.1.2 Planen

Als zweiten Schritt kommt das Planen. Hier wird das Projekt konkreter und es wird ein Zeitplan erstellt. Es werden Lösungsvorschläge erarbeitet und je nach Team größe, werden bestimmte Aufgaben zugeteilt. Im Probe-IPA Fall fällt die Aufgabenverteilung natürlich weg.

6.1.3 Entscheiden

Beim Entscheiden, wird entschieden welchen Lösungsweg gegangen werden soll. Es werden die verschiedenen Lösungsvarianten mit einander verglichen und evaluiert, welche die beste ist. Des weiteren wird z.B definiert mit welchen Tools / Technologien gearbeitet wird. Im Fall dieser Probe-IPA ist dies durch das bereits bestehende Projekt schon vorgegeben. Wichtig ist auch, dass die Kriterien, welche zu diesen Entscheidungen geführt haben, definiert werden.

6.1.4 Realisieren

In diesem Teil geht es an die Umsetzung. Das Projekt wird nach dem definierten Plan sowie Zeitplan versucht umzusetzen.

6.1.5 Kontrollieren

Der fünfte Schritt erfolgt teilweise parallel zum Vierten. In diesem Schritt wird von oben auf das laufende Projekt geblickt und geschaut, ob alles nach Plan läuft. Gibt es Abweichungen und falls ja, können diese begründet werden?

6.1.6 Auswerten

Der letzte Schritt dient dazu, nochmals auf das Projekt zurückzublicken und es zu Reflektieren.

6.2 Alternative Methode - Scrum

Nebst IPERKA gibt es auch alternative Projektmanagementmethoden. Eine davon ist Scrum. Scrum eignet sich allerdings nicht besonders für die Umsetzung eines Projekts wie die Probe-IPA. Sie ist eine Agile Projektmanagementmethode, welche sich für Projekte eignet, die sehr dynamisch und doch komplex sind. Meistens sind die konkreten Anforderungen zu Beginn sogar noch unklar. Zudem kann Scrum nur teilweise alleine durchgeführt werden. Dies ist bei der Probe-IPA nicht der Fall. Deshalb wurde sich für IPERKA entschieden.

7 Arbeitsprotokoll

Datum	06.11.2024
Bearbeitete Arbeitspakete	1.1, 1.2, 2.1, 2.2, 7.1, 7.2, 7.3
Arbeitszeit	8h
Überzeit	0
Vergleich mit dem Zeitplan	Da ich den Zeitplan noch nicht fertig erstellt habe, kann ich für heute keinen Vergleich ziehen.
Erfolge und Probleme	Zu Beginn wusste ich nicht genau wie ich am besten vorgehe resp. was ich zuerst angehe, da hat mir das vorhandene Template einen sehr guten Leitfaden gegeben. Und so habe ich begonnen alles der Reihe nach auszufüllen/ zu dokumentieren. Und bin am Schluss weiter gekommen als gedacht.
Tagesreflexion	Heute bin ich sehr gut voran gekommen. Ich konnte bereits den Teil 1 der Dokumentation abschliessen und mit den Arbeitspaketen beginnen.
In Anspruch genommene Hilfe	Fragen an Pascal bezüglich der Aufgabenstellung. War mir unsicher, wo genau die Kundendokumentation hin muss. Jetzt weiss ich, dass es reicht, wenn ich sie im Anhang anhänge.

Datum	07.11.2024
Bearbeitete Arbeitspakete	2.1, 2.2, 2.3, 2.4
Arbeitszeit	8h
Überzeit	0
Vergleich mit dem Zeitplan	Eine Stunde voraus
Erfolge und Probleme	Beim Zeitplan hatte das Template nicht richtig funktioniert. Da kam ein bisschen extra Aufwand dazu, da ich aber ansonsten etwas schneller war hat sich das wieder kompensiert. Ich konnte bereits heute mit dem SPA Lösungskonzept beginnen. Da gestern der Zeitplan noch nicht stand, erwähne ich es heute: Ein weiterer Erfolg, ich konnte Meilenstein A (Informieren) gestern erfolgreich und überpünktlich abschliessen.
Tagesreflexion	Ich bin auch heute wieder sehr gut voran gekommen, und bin somit dem Zeitplan eine Stunde voraus. Dies finde ich sehr angenehm, denn es lässt einem etwas ruhiger und weniger gestresst Arbeiten.
In Anspruch genommene Hilfe	keine

Datum	08.11.2024
Bearbeitete Arbeitspakete	2.4, 2.5, 7.4
Arbeitszeit	8h
Überzeit	0
Vergleich mit dem Zeitplan	Ich hatte für das Lösungskonzept der SPA eine Stunde länger als gedacht. Dadurch bin ich gerade genau im Zeitplan, da ich für das Backend weniger Zeit als geplant brauchte.
Erfolge und Probleme	Heute hatte ich eine kurze Zeit Probleme mit Latex. Dies hat mich etwas Zeit gekostet. Ansonsten bin ich gut voran gekommen und auf Kurs.
Tagesreflexion	Heute habe ich am Lösungskonzept der SPA und dem Testkonzept gearbeitet. Das für die SPA konnte ich bereits abschliessen. Am Montag geht es dann weiter mit dem Testkonzept, mit welchem ich heute schon begonnen habe. Zudem hatte ich am Morgen meinen ersten Expertenbesuch, welcher im Rahmen der Probe-IPA von Bernd durchgeführt worden ist.
In Anspruch genommene Hilfe	keine

Datum	11.11.2024
Bearbeitete Arbeitspakete	3.1, 4.1
Arbeitszeit	8h
Überzeit	0
Vergleich mit dem Zeitplan	Ich konnte etwas früher mit der Backend Implementation beginnen.
Erfolge und Probleme	Ich hatte heute etwas Schwierigkeiten Code stellen und Regexe in Latex einzufügen. Dies hat mich etwas Zeit gekostet, ich konnte es aber lösen.
Tagesreflexion	Heute habe ich den Rest Endpunkt implementiert und dokumentiert. Ich konnte mit der weiteren Logik bereits beginnen.
In Anspruch genommene Hilfe	keine

Datum	13.11.2024
Bearbeitete Arbeitspakete	4.1, 4.2
Arbeitszeit	8h
Überzeit	0
Vergleich mit dem Zeitplan	Ich bin dem Zeitplan etwas voraus. Ich konnte heute bereits mit den Tests für das Backend beginnen.
Erfolge und Probleme	Ich bin heute sehr gut voran gekommen, und konnte das Backend fertig implementieren. So konnte ich bereits mit den Tests beginnen. Allerdings hatte ich da zu Beginn noch kleine Schwierigkeiten mit Wiremock. Diese liessen sich aber mit etwas Geduld beheben.
Tagesreflexion	Heute habe ich die Hauptfunktionalitäten des Backends implementiert. Und bereits mit den Tests begonnen. Es war ein sehr produktiver Tag.
In Anspruch genommene Hilfe	keine

Datum	14.11.2024
Bearbeitete Arbeitspakete	4.1, 4.2, 4.3
Arbeitszeit	8h
Überzeit	0
Vergleich mit dem Zeitplan	Ich bin im Zeitplan bisschen voraus, und konnte bereits mit der SPA-Implementierung beginnen.
Erfolge und Probleme	Heute bin ich sehr weit gekommen ich konnte das Backend abschliessen inkl. Tests. Im Frontend konnte ich bereits den Durchstich erzielen.
Tagesreflexion	Es war ein sehr produktiver Tag. Am morgen war ich beschäftigt mit der Dokumentation des Backends und den Tests für das Backend. Am späteren Nachmittag begann ich dann mit der SPA.
In Anspruch genommene Hilfe	Frage an Pascal bzgl. Tiefe der Dokumentation.

Datum	15.11.2024
Bearbeitete Arbeitspakete	4.3, 4.4, 7.4
Arbeitszeit	8h
Überzeit	0
Vergleich mit dem Zeitplan	Im Vergleich zum Zeitplan bin ich ein bisschen voraus, und konnte bereits heute mit den Selenium Integration-Tests beginnen.
Erfolge und Probleme	Ich konnte wieder ein Arbeitspaket vor der geplanten Zeit abschliessen. Ich hatte zu Beginn etwas Probleme mit den Selenium-Tests, bzw. ich brauchte einige Zeit, bis ich die richtigen Selektoren für die jeweiligen HTML-Elemente gefunden habe.
Tagesreflexion	Heute bin ich gut vorangekommen und konnte die Implementation in der SPA abschliessen, und bereits mit den Selenium-Tests beginnen. In der ersten Hälfte des Nachmittags hatte ich den 2. Expertenbesuch, welcher wieder durch Bernd simuliert wurde. Dieser lief auch gut, sprich ich konnte alles zeigen, was gefordert wurde.
In Anspruch genommene Hilfe	keine

Datum	18.11.2024
Bearbeitete Arbeitspakete	4.4, 4.5, 5.1
Arbeitszeit	8h
Überzeit	0
Vergleich mit dem Zeitplan	Da ich für die Selenium-Tests etwas länger gebraucht habe, bin ich dem Zeitplan nicht mehr voraus. Dies ist allerdings nicht dramatisch, denn ich bin jetzt genau soweit ich geplant habe.
Erfolge und Probleme	Bei den Selenium-Tests hatte ich heute einige Probleme. Ich verschwendete zu viel Zeit, für einen zu komplizierten Lösungsweg, den man auch anders lösen könnte. Schlussendlich bin ich dann auf die einfachere andere Lösung gekommen. Nichts desto trotz konnte ich heute den Meilenstein «Realisiern» pünktlich abschliessen. Und mit dem nächsten Schritt dem «Kontrollieren» weiter machen.
Tagesreflexion	Heute war ich vom Morgen bis zur Hälfte des Nachmittags mit den Selenium-Tests beschäftigt. Danach startete ich mit der neuen IPERKA-Phase «Kontrollieren». Bei dieser bin ich schon sehr weit gekommen. Ich konnte alle manuellen Tests, Unit- und Integration-Tests bereits ausführen.
In Anspruch genommene Hilfe	keine

Datum	20.11.2024
Bearbeitete Arbeitspakete	5.1, 5.2, 5.3
Arbeitszeit	8h
Überzeit	0
Vergleich mit dem Zeitplan	Ich bin super im Zeitplan und mit den letzten Arbeiten der Dokumentation beschäftigt.
Erfolge und Probleme	Heute konnte ich die Arbeitspakete 5.1 und 5.2 unter der geplanten Zeit abschliessen und mit 5.3 beginnen.
Tagesreflexion	Heute war der Tag, der mir bisher am wenigsten Freude bereitet hat. Ich war ab der zweiten Hälfte des Morgen viel mit Dokumentation / Code durchlesen und Schreibfehler verbessern beschäftigt. Ich allerdings schon sehr weit gekommen, so das morgen nicht mehr diesbezüglich gemacht werden muss.
In Anspruch genommene Hilfe	Fragen an Pascal, ob und wie ich den geschriebenen Code im Anhang angeben darf. Ich darf den Code anhängen, am besten der «Diff» zum Master des letzten Patchsets.

Datum	21.11.2024
Bearbeitete Arbeitspakete	
Arbeitszeit	8h
Überzeit	0
Vergleich mit dem Zeitplan	
Erfolge und Probleme	
Tagesreflexion	
In Anspruch genommene Hilfe	

8 Zeitplan

Die folgenden 2 Seiten beinhalten den Zeitplan. Er ist der Leitfaden während der Probe-IPA. Der Zeitplan ist dargestellt in einem GANT-Diagramm. In diesem werden 2h Blöcke verwendet. Die Zeitangaben sind sehr grosszügig geschätzt, um zusätzlichen Stress zu vermeiden.

[illegible]

[illegible]

Teil II

Projekt

9 Informieren

In diesem Kapitel geht es um die erste von 6 Phasen der IPERKA-Methode, dem Informieren. Es bietet Platz um aufzuzeigen, was während dieser Phase unternommen wurde.

9.1 Analyse

Als erster Schritt wurde die Aufgabe analysiert, einen Überblick verschaffen und in folgender Kurzfassung zusammengefasst:

Kurzfassung

Umzusetzen ist eine Funktion in der Admin App der Airlock IAM Applikation, welche es den Adminnutzern ermöglicht, die 16 stelligen Airlock 2FA Aktivierungscode der Nutzer anzeigen zu lassen. Dies hilft Ihnen, die Endnutzer bei der Aktivierung der Airlock 2FA zu unterstützen. Der Aktivierungscode sollte per Konpfdruk angezeigt werden können, zum Beispiel als Popup. Dies ist jedoch noch zu evaluieren, vielleicht bieten sich auch noch andere Optionen an. Sicher ist, dass der Code im Airlock 2FA Management angezeigt werden soll und nur, falls durch den Admin gewollt.

Weiter soll es möglich sein, dass nicht alle Admins sich den Code anzeigen können, sondern nur die mit der entsprechenden Rollen.

Das ganze muss mit UI, Unit und Integration Tests getestet werden, und in der Kundendokumentation ergänzt werden.

Abgrenzung

Es gibt bereits die Funktionalität, dass Admins pro Nutzer Aktivierungsbrieфе generieren können. Im Rahmen dieses Auftrags, soll dieser Bereich nicht erweitert oder verändert werden.

9.1.1 Futurae

Die Airlock 2FA App wird von Futurae entwickelt. Das hat zur Folge das zwischen IAM und Futurae wichtige Informationen ausgetauscht werden müssen. Dafür bietet Futurae 3 verschiedene API's an: die Auth API, die Admin API und die Log API. Für die Probe-IPA ist sowohl die Admin-API als auch die Auth-API relevant. Bei der Anfrage für den 16-stelligen Aktivierungscode wird zwar die Admin-API gebraucht, allerdings um den Code bei dem Start-Enrollment-Request zu erzwingen, wird auch die Auth-API

verwendet werden müssen. Denn, Aktivierungen aus dem Protected so wie Public Self-Service führen über Auth-API.

9.2 Technische Referenzen

Für die Technischen Infos sind folgende zwei Links sehr hilfreich:

- Futurae API: <https://www.futurae.com/docs/api/auth/>
Da der Aktivierungs Code von Futurae kommt, ist dessen API Dokumentation eine wichtige Quelle.
- IAM Kunden Dokumentation: <https://docs.airlock.com/iam/8.3/> ' Notwendig, um allgemeine Informationen bezüglich Airlock 2FA nachzulesen

9.3 Anforderungen

Nach der Analyse und nachdem der Auftrag verstanden wurde, konnten die Anforderungen definiert werden. Diese sind immer in funktionale und nicht-funktionale aufgeteilt.

Folgende Abkürzungen werden verwendet:

- FA <Zahl> ... bedeutet funktionale Anforderung, mit numerisch aufsteigendem Index.
- NFA <Zahl> ... bedeutet nicht-funktionale Anforderung, mit numerisch aufsteigendem Index.

9.3.1 Rest / Backend

Folgend, sind die Anforderungen für das Backend resp. den Rest-Teil definiert.

Funktionale Anforderungen

- FA 1: Das Backend soll der SPA den Activation Code anbieten.
- FA 2: Der Activation Code darf nur angeboten werden, wenn der Admin auch die notwendige Rolle hat.
- FA 3: Im User Activities Logfile, soll geloggt werden, welcher Administrator zu welchem Zeitpunkt den 16-stelligen Aktivierungscode angezeigt hat.
- FA 4: Neue Plugins oder Properties sollen einen klaren und vollständigen Hilfetext haben.

Nicht-funktionale Anforderungen

- NFA 1: Sämtliche Fehlerfälle werden korrekt behandelt.
- NFA 2: Der Code entspricht dem bestehenden Codeschema.
- NFA 3: Alle neuen Funktionalitäten werden durch Tests abgedeckt.
- NFA 4: Veränderte / neue Restendpoints werden um die notwendige Doku erweitert.

9.3.2 SPA

Folgend, sind die Anforderungen für die SPA definiert.

Funktionale Anforderungen

- FA 5: Die SPA muss in der Lage sein den 16-stelligen QR-Code auf Knopfdruck anzuzeigen.
- FA 6: Das neue UI gleicht sich dem bisherigen Verhalten der SPA an.
- FA 7: Das neue UI hat den gleichen Style wie das bisherige.
- FA 8: Die Aktion, um den 16-stelligen Aktivierungscode anzuzeigen, sollte nur verfügbar sein, falls der Admin die nötigen Rollen hat, und ein offener Aktivierungscode existiert.

Nicht-funktionale Anforderungen

- NFA 5: Es werden nur in der Adminapp existierende UI Komponenten verwendet.
- NFA 6: Das UI lädt in jedem Fall ohne Probleme.
- NFA 7: Alle neuen Funktionalitäten werden durch Selenium Integration Tests abgedeckt.

9.3.3 Kundendokumentation

Folgend, sind die Anforderungen für die Kundendokumentation definiert.

Funktionale Anforderungen

- FA 9: Die Kunden Doku wird sinnvoll um das neue Feature erweitert.
- FA 10: Die Kundendoku ist auf Englisch geschrieben.

Nicht-funktionale Anforderungen

- NFA 8: Die Kundendoku hat keine Schreibfehler.
- NFA 9: Die Kundendoku passt in das bestehende Produkt.

10 Planen

In diesem Abschnitt, wird die Planung beschrieben. In dieser Phase werden basierend auf den Anforderungen Arbeitspakete erstellt, und in einem GANT-Diagramm auf die 10 Tage eingeteilt.

10.1 Arbeitspakete

Um den ganzen Auftrag in kleine übersichtliche Teile aufzuteilen, wird er in verschiedene kleine Arbeitspakete unterteilt. Die Arbeitspakete sind jeweils nummeriert, haben einen Namen, einen geschätzten Aufwand in Stunden und eine «Definition of Done»/ ein erwartetes Ergebnis. Die Aufwände sind oft mit einem gewissen Puffer geschätzt.

Die Pakete sind nach den 6 Phasen der IPERKA Methode aufgelistet. Arbeiten welche IPA-spezifisch sind, werden unter Rahmenaufgaben aufgeführt.

10.1.1 Informieren

Hier, sind die Arbeitspakete, welche während der IPERKA-Phase «Informieren» bearbeitet wurden, aufgelistet.

Nummer	1.1
Name	Projektumfeld analysieren und beschreiben
Geschätzter Aufwand	2h
Erwartetes Ergebnis	Das Ziel der Arbeit ist klar und ein grober Überblick besteht.

Nummer	1.2
Name	Anforderungen definieren
Geschätzter Aufwand	2h
Erwartetes Ergebnis	Die funktionalen und nicht-funktionalen Anforderungen sind definiert und beschrieben.

10.1.2 Planen

Hier, sind die Arbeitspakete, welche während der IPERKA-Phase «Planen» bearbeitet werden, aufgelistet.

Nummer	2.1	
Name	Arbeitspakete definieren	
Geschätzter Aufwand	3h	
Erwartetes Ergebnis	Er-	Die ganze Arbeit ist in kleine logische Arbeitspakete unterteilt. Alle Arbeitspakete sind klar definiert.
Nummer	2.2	
Name	Zeitplan erstellen	
Geschätzter Aufwand	1h	
Erwartetes Ergebnis	Er-	Der GANT-Zeitplan ist anhand der Arbeitspakete erstellt. Es sind alle Arbeitspakete auf dem Zeitplan vorhanden.
Nummer	2.3	
Name	Lösungskonzept für das Backend erarbeiten	
Geschätzter Aufwand	4h	
Erwartetes Ergebnis	Er-	Es ist mindestens ein Lösungsvorschlag definiert und so weit wie Sinnvoll beschrieben und durchgedacht. Der relevante Backendcode ist verstanden.

Nummer	2.4
Name	Lösungskonzept für die SPA erarbeiten
Geschätzter Aufwand	4h
Erwartetes Ergebnis	Es ist mindestens ein Lösungsvorschlag definiert und so weit wie Sinnvoll beschrieben und durchgedacht. Es sind verschiedene Mockups vorhanden, und der relevante SPA Code ist verstanden.

Nummer	2.5
Name	Test- und Qualitätssicherungskonzept erstellen
Geschätzter Aufwand	4h
Erwartetes Ergebnis	Das Testkonzept ist erstellt und dokumentiert. Das Qualitätssicherungskonzept ist erstellt und dokumentiert.

10.1.3 Entscheiden

Hier, sind die Arbeitspakete, welche während der IPERKA-Phase «Entscheiden» bearbeitet werden, aufgelistet.

Nummer	3.1
Name	Lösungsvarianten evaluieren
Geschätzter Aufwand	2h
Erwartetes Ergebnis	Aus den verschiedenen Lösungsvarianten der SPA und des Backends wurde sich für eine entschieden. Die Entscheidung ist begründet dokumentiert.

10.1.4 Realisieren

Hier, sind die Arbeitspakete, welche während der IPERKA-Phase «Realisieren» bearbeitet werden, aufgelistet.

Nummer	4.1
Name	Das Backend erweitern
Geschätzter Aufwand	14h
Erwartetes Ergebnis	Alle Anforderungen für das Backend sind nach dem definierten Lösungsansatz umgesetzt. Die Lösung ist vollständig dokumentiert.

Nummer	4.2
Name	Unit- und Integration-Tests schreiben
Geschätzter Aufwand	6h
Erwartetes Ergebnis	Alle neuen Funktionalitäten sind mit Unit- und/oder Integration-Tests getestet.

Nummer	4.3
Name	Die SPA erweitern
Geschätzter Aufwand	6h
Erwartetes Ergebnis	Alle Anforderungen für die SPA sind nach dem definierten Lösungsansatz umgesetzt. Die Lösung ist vollständig dokumentiert.

Nummer	4.4
Name	Selenium Integration-Tests implementieren
Geschätzter Aufwand	5h
Erwartetes Ergebnis	Alle neuen Funktionalitäten sind mit Selenium Integration-Tests getestet.

Nummer	4.5
Name	Kundendokumentation schreiben
Geschätzter Aufwand	2h
Erwartetes Ergebnis	Die neue Funktionalität ist in der Kundendokumentation dokumentiert, und alle Anforderungen sind erfüllt.

10.1.5 Kontrollieren

Hier, sind die Arbeitspakete, welche während der IPERKA-Phase «Kontrollieren» bearbeitet werden, aufgelistet.

Nummer	5.1
Name	Tests durchführen, und Fehler beheben
Geschätzter Aufwand	4h
Erwartetes Ergebnis	Tests sind gemäss Testkonzept durchgeführt, und mögliche Fehler sind behoben.

Nummer	5.2
Name	Codequalität prüfen, und Refactorn
Geschätzter Aufwand	1h
Erwartetes Ergebnis	Code ist nochmals durchgeschaut, und Unschönheiten sind bereinigt.

Nummer	5.3
Name	Dokumentation finalisieren
Geschätzter Aufwand	8h
Erwartetes Ergebnis	Die Dokumentation ist soweit wie möglich finalisiert und entspricht den Vorgaben.

10.1.6 Auswerten

Hier, sind die Arbeitspakete, welche während der letzten IPERKA-Phase «Auswerten» bearbeitet werden, aufgelistet.

Nummer	6.2
Name	Reflexion schreiben
Geschätzter Aufwand	2h
Erwartetes Ergebnis	Reflexion zu der Arbeit ist geschrieben.

10.1.7 Rahmenaufgaben

Hier, sind die Arbeitspakete, welche IPA-spezifische Arbeit erfordern, aufgelistet.

Nummer	7.1
Name	Projektstruktur aufsetzen
Geschätzter Aufwand	2h
Erwartetes Ergebnis	Das Grundgerüst für den Bericht steht. Der Latex-Build ist lauffähig und generiert ein anschauliches PDF.

Nummer	7.2
Name	Aufgabenstellung und Rahmenbedingungen beschreiben
Geschätzter Aufwand	1h
Erwartetes Ergebnis	Die Aufgabenstellung ist in den Bericht übernommen. Benützte Firmenstandards sowie die Projektaufbauorganisation sind definiert und beschrieben.

Nummer	7.3
Name	Projektmanagementmethode definieren
Geschätzter Aufwand	1h
Erwartetes Ergebnis	Er- Es steht fest mit welcher Projektmanagementmethode die Probe-IPA umgesetzt werden soll. Der Bericht wurde so gegliedert.

Nummer	7.4
Name	Expertenbesuche
Geschätzter Aufwand	4h
Erwartetes Ergebnis	Er- Infos aus dem Gespräch sind am richtigen Ort festgehalten.

Nummer	7.5
Name	Anhang erstellen
Geschätzter Aufwand	2h
Erwartetes Ergebnis	Er- Der Anhang ist erstellt und beinhaltet alle nötigen und verlangten Inhalte.

10.2 Lösungskonzept Backend

In diesem Kapitel ist das Lösungskonzept für das Backend beschrieben. Das Konzept richtet sich nach den in Kapitel 9.3.1 definierten Anforderungen. Es wurden diverse TODO's im Code hinzugefügt, dies macht die Implementation danach etwas effizienter.

10.2.1 REST

Damit die SPA den 16-stelligen Aktivierungscode anzeigen kann, muss er mit Hilfe einer REST-Schnittstelle übermittelt werden. Dass er aber überhaupt von Futurae erstellt wird, muss er bei dem Enrollement, also dem Call der einen neuen Nutzer erstellt, explizit gefordert werden. Dies funktioniert in dem man den Requestparameter «short_code» auf true setzt. Für die Übermittlung dieser Information an die SPA stehen 2 Optionen im Raum:

- Option 1: Den Endpunkt, welcher alle Accountdaten von jedem Nutzer zurückgibt um den Activation Code erweitern. Dies hätte zur Folge das der Endpunkt um ein optionales Feld «activation_code_short» erweitert wird.
- Option 2: Einen neuen Endpunkt erstellen, welcher den offenen Aktivierungscode zurück gibt. Dies wäre ein einfacher GET-Endpunkte, welcher, falls vorhanden den neusten, ausstehenden Aktivierungscode zurück gibt. Folgend eine kurze Spezifikation des Endpunktes:

Pfad: /auth-admin/rest/users/userId/tokens/airlock-2fa/activation-code-short

HTTP-Methode: GET

Pfadparameter: userid

Response: Optionaler Activation Code, kann leer sein

Status Codes:

200 Ok	16-stelliger Aktivierungscode oder eine leere Response
401 Unauthorized	Invalide oder fehlende Authentifizierung
403 Forbidden	Der Zugriff ist verboten (z.B bei fehlender oder falscher Adminrolle)
404 Not Found	Mögliche Error Codes: <ul style="list-style-type: none">– USER_NOT_FOUND– ACCOUNT_NOT_FOUND

In beiden Fällen müsste der Restflow so aussehen:

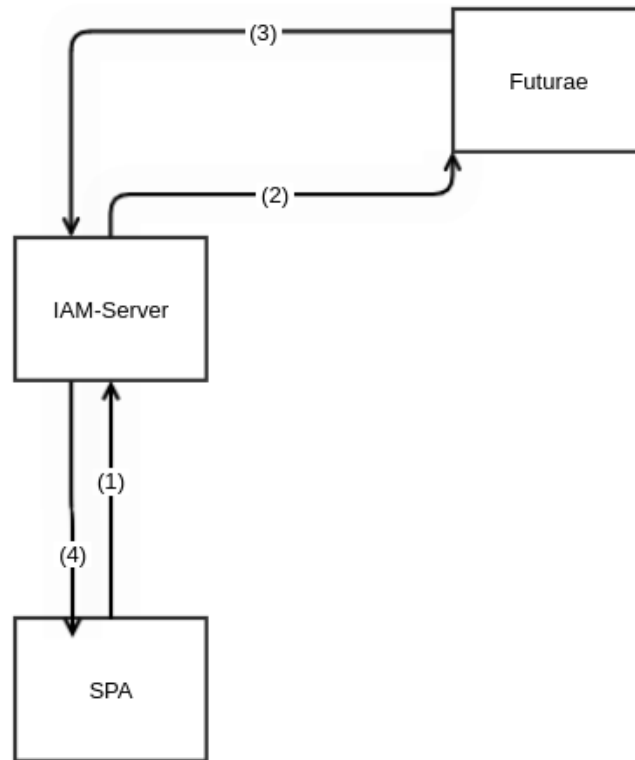


Abbildung 10.1: Restflow um 16-stelligen Aktivierungscode zu bekommen

- (1) Die SPA macht als Reaktion auf einen Klick einen Request, ans IAM-Backend. Je nach Option, geht dieser an einen anderen Endpunkt.
- (2) Das Backend macht folgenden Request an Futuræ:
`/srv/admin/v1/enrollments?status=pending`
- (3) Da, bei dem Enrollment Request zu Futuræ der 16-stellige Aktivierungscode explizit gefordert wurde, wird dieser Request, falls überhaupt ein austehendes Enrollment vorhanden ist, dieses auch zurück geben. Da mehrere offene Enrollments vorhanden sein können, muss immer das Neuste genommen werden. So ist immer klar um welchen 16-stelligen Aktivierungscode es sich handelt. Helpdesk oder Schalter Mitarbeiter können so die Aktivierung direkt mit dem Kunden durchspielen.
- (4) Das Backend gibt den Aktivierungscode an die SPA weiter. Je nach Option auch noch die anderen Accountdaten. Falls keiner vorhanden ist, wird das Feld in der

Response einfach leer gelassen.

10.2.2 Rollenlogik

Es gibt bereits eine Regel, welche das Ansehen von Airlock 2FA Aktivierungsdaten einschränkt. Diese Regel kann wiederverwendet werden. Dazu gibt es einen «Airlock2FAAccessController.java.» Dieser kann beim erstellen der Response injected werden. Mit der Methode «isAllowedToSeeActivationSecrets» kann dann überprüft werden, ob der Adminnutzer diese Info überhaupt sehen darf. Am besten wird dieser Check noch vor dem Futurae Request ausgeführt, um einen unnötigen Roundtrip zu vermeiden und es möglichst effizient zu halten. Falls die Option mit einem neuen Endpunkt gewählt wird, kann dieser separat geschützt werden, dann fällt die Logik mit dem «Airlock2FAAccessController.java.» weg.

10.2.3 Wichtige Klassen und Interfaces

Option 1

- UserAirlock2FADeviceResource.java: In dieser Klasse befindet sich der GET-Endpunkt «/auth-admin/rest/users/userId/tokens/airlock-2fa », welcher erweitert werden muss.
- Airlock2FAUserAccount.java dies ist die Klasse welche die wichtigen Daten über den Account beinhaltet. Diese Klasse muss um das Feld «activationCodeShort» erweitert werden.
- Airlock2FAAdminService.java: Dieser Service wird aus der Resource aufgerufen, um den Account von der Datenbank zu bekommen.
- Airlock2FAUserAccountRepository.java: Das Repository ist die Schnittstelle zwischen der Datenbank und dem Service. Die darin enthaltene «findBy()» Methode gibt schluss endlich den zusammengestellten «Airlock2FAUserAccount.java» zurück.

Option 2

- UserAirlock2FADeviceResource.java: In dieser Klasse wird der oben definierte Endpunkt «/auth-admin/rest/users/userId/tokens/airlock-2fa/activation-code-short » erstellt.
- Airlock2FAAdminService.java: Dieser Service wird aus der Resource aufgerufen, hier muss eine neue Methode erstellt werden, welche den 16-stelligen Aktivierungscode, via dem «FuturaeAdminApiEnrollmentServiceImpl.java», der Resource zur Verfügung stellt.

Request zu Futurae

Der Request zu Futurae wird in beiden Optionen gleich aussehen. Lediglich der «FuturaeAdminApiEnrollmentServiceImpl.java» wird an verschiedenen Orten gebraucht.

- FuturaeAdminApiEnrollmentServiceImpl.java: In diesem Service muss der Airlock2FAAccessController injected werden. Zusätzlich wird es eine neue Methode geben müssen, welche zuerst mit Hilfe des Airlock2FAAccessController prüft, ob der Adminnutzer die richtige Rolle hat. Danach wird via die «FuturaeAdminApiEnrollmentRequestFactory.java» der Request zusammengestellt. Dieser Request wird dann via den RestClient ausgeführt. Die Response wird dann in ein neu erstelltes ...Response Objekt gespeichert und zurück gegeben.
- FuturaeAdminApiEnrollmentRequestFactory.java: In dieser Klasse braucht es eine neue Methode welche einen Request zusammenstellt, der von Futurae die neueste offene Aktivierung anfragt.
- AdminEnrollmentRequest.java: Diese Klasse bildet den Admin-Request ab, welcher gesendet wird, um neue Geräte zu aktivieren. Er muss um das Feld «short_code» erweitert werden. Dieses Feld muss anschliessend für jeden Enrollment-Request auf true gesetzt werden.
- FuturaeAuthApiEnrollmentRequestFactory.java: Es ist wichtig auch in dieser Klasse auf dem Enrollmentrequest «short_code» auf true zu setzen, ansonsten wird der 16-stellige Aktivierungscode nicht erzwungen, wenn das Enrollment von einem Endnutzer aus dem Self-Service gestartet wird. Denn dann verläuft es über die Auth API.

10.3 Lösungskonzept SPA

In diesem Kapitel werden die Lösungsideen für das Frontend / die SPA dokumentiert. Das Konzept richtet sich nach den in Kapitel 9.3.2 definierten Anforderungen. Es wurden diverse TODO's im Code hinzugefügt, dies macht die Implementation danach viel effizienter.

10.3.1 Mockups

Um sich eine Vorstellung zu machen, wurden zuerst verschiedene Mockups erstellt. Da es fürs UI keine grosse Änderung ist, konnten die Mockups teils direkt im Code erstellt werden, natürlich ohne Funktionalität.

Auf dem folgende Bild ist die Adminapp im Airlock 2FA Management-Tab des Nutzers «itester». Es wird davon ausgegangen, dass der Admin die nötigen Rollen hat, um sich den Aktivierungscode anzuzeigen.

Abbildung 10.2: Das folgende Bild zeigt den UI Vorschlag

Overview Profile Authentication Methods Password **Airlock 2FA** Activities

Airlock 2FA Account

This user does not have any devices and therefore is not able to login u

There are no activation letters for this user.

Account ID	e98b0adb-031d-4c38-a5a0-3b518d3c5
Display name	
Created at	07.11.2024 10:55:21 (UTC+01:00)
Updated at	09.11.2024 09:50:56 (UTC+01:00)
Failed/Max attempts	0/40
Allowed factors	One-Touch, Online QR code, Offline QR code, Passcode, Mobile-only (SDK or app-to-app)
Status	Disabled

Activation Code



acse 43n4 ak83 lo90

Close

Delete

Assign hardware token

Order activation letter

Create activation letter

View Activation Code

Es wird der Ansatz verfolgt, dass unten rechts ein weiterer Button hinzukommt. Dieser wird allerdings nur dann angezeigt, wenn der Admin auch die nötigen Rollen dazu hat und ein offener Aktivierungscode vorhanden ist(spricht nicht null zurück kommt). Wird der Button geklickt, soll sich ein Popup öffnen, in dem der 16-stellige Aktivierungscode angezeigt wird. Ev. könnte es auch eine Option sein, das Enrollmentdatum auch noch anzuzeigen, das könnte bei Fehlversuchen dem Admin eventuell hilfreich sein. Dies könnte dann in etwa so aussehen:

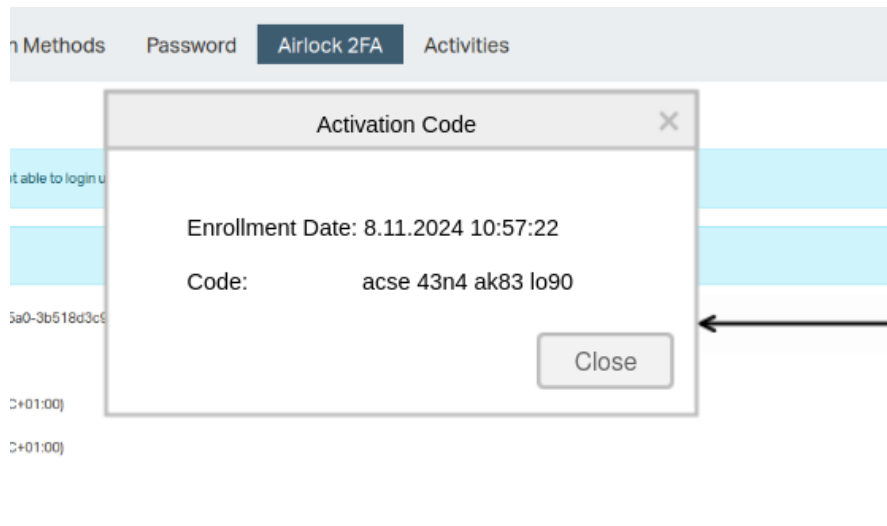


Abbildung 10.3: Popup Variante mit Datum

10.3.2 Lösungsvariante ohne Popup

Anstelle eines Popups, welches durch einen Button ausgelöst wird, könnte man in der Accountübersicht auch folgendes einbauen:

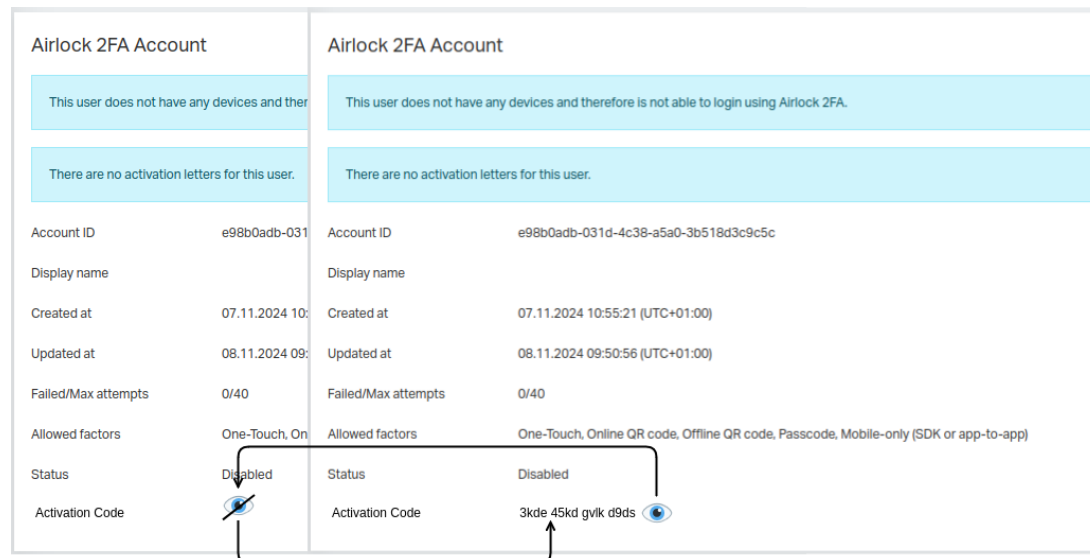


Abbildung 10.4: SPA Lösung ohne Popup

Analog zu einer Passwortanzeige könnte man es mit einem Auge darstellen. Wird auf das durchgestrichene Auge geklickt, wird der Code angezeigt. Klickt man nun auf das offene Auge, verschwindet der Aktivierungscode wieder.

Diese neue Zeile wird nur dann angezeigt, wenn auch ein Aktivierungscode vorhanden ist und der Admin die erforderlichen Rechte hat.

10.3.3 Rollenlogik

Damit sichergestellt wird, dass der Button nur dann angezeigt wird, wenn der Benutzer dies möchte, kann ihm das Attribut «hideOnAccessDenied» gesetzt werden. Der Button braucht zudem ein sprechende ID. Diese ID wird dann im Access Controller im Backend, als Permissonkey verwendet.

10.3.4 Wichtige Komponenten und Services

- `airlock-2fa.component.html`: In diesem File wird die Darstellung des UI definiert. Hier muss der neue Button hinzugefügt werden.
- `airlock-2fa.service.ts`: In diesem Service wird der Request an das IAM-Backend gemacht, um den Aktivierungscode zu bekommen. Anschliessend wird die Response geparsed und zurückgegeben.
- `airlock-2fa.component.ts`: In dieser Component, wird die Logik für den neuen Button implementiert. Konkret:

- Beim laden der Komponente wird via den obigen Service, der Aktivierungscode abgefragt.
- Er darf nur angezeigt werden, falls auch ein Aktivierungscode vorhanden ist. Dies kann über die Response des Backends ermittelt werden, ist sie leer, existiert keine anstehende Aktivierung.
- Wird er angeklickt, öffnet sich ein Popup mit dem aktuellsten, offenen, 16-stelligen Aktivierungscode.
- Zudem wird es ein neues Model für den Aktivierungscode geben müssen, falls das Datum auch angezeigt werden soll. Das sollte wie folgt aussehen:

```
export interface Airlock2FAActivationCodeData {
    activationCodeShort: string;
    enrollmentDate: Date;
}
```

10.3.5 Translation Keys

Um in der Adminapp die Sprachen Deutsch, Englisch und Französisch zu unterstützen wird i18n verwendet. Dafür wird für jeden String in der SPA ein Translation Key definiert. Hierfür gibt es 3 verschiedene JSON-Files; für jede Sprache eines. Das Value ist der übersetzte Text in die jeweilige Sprache. Je nach Sprache wird nun ein anderes JSON-File angezogen, dies führt dazu, dass immer die korrekte Übersetzung verwendet wird. In diesem Fall benötigt es mindestens folgende 3 Keys:

- user.airlock-2fa.activation.view-activation-code.button: Text im Button
- user.airlock-2fa.activation.view-activation-code.popup.title: Titel des Popup's
- user.airlock-2fa.activation.view-activation-code.popup.text: Text im Popup

10.4 Testkonzept

Das neue Feature soll fehlerfrei und sowie in den Anforderungen definiert, funktionieren. Um dies sicherzustellen wird folgend ein Testkonzept zusammen gestellt, nach welchem das Feature später getestet werden sollte. Alle erwähnten Technologien sind in Kapitel 4.2 verlinkt.

10.4.1 Benötigte Testmittel

Unit-Tests und sämtliche Integration-Tests werden von der Entwicklungsumgebung IntelliJ ausgeführt. Da die Applikation lokal auch aus dem IntelliJ gestartet werden kann, werden auch die manuellen Tests mit der Hilfe von IntelliJ durchgeführt. Die Applikation läuft dann auf dem Port 8443.

Damit sichergegangen werden kann, dass nicht nur das neue Feature funktioniert, sondern alles darum herum auch noch, führt die Jenkinspipline bei jedem Push eines neuen Patchsets eine Auswahl an Tests aus.

10.4.2 Wiremock

In gewissen Unit- sowie allen Integration-Tests wird Wiremock verwendet um den Futuræ Server zu simulieren. Wiremock stellt dabei einen Dummy-Server zurverfügung, das heisst alle Anfragen gehen an diesen Server und somit nicht über das Netzwerk. Dies verhindert Flaky Tests und bietet eine konstante und auf den Testcase angepasste API. Denn, dieser Server wird für jeden Test individuell konfiguriert.

10.4.3 Unit-Tests

Unit-Tests werden verwendet, um die Kernlogik in kleinen isolierten Einheiten zu testen. Hierfür werden komplexe Umsysteme und Klassen gemockt. Für die Klassen wird dies mit Mockito gemacht. Die Tests ansich werden mit JUnit5 implementiert. Konkret für diesen Auftrag müssen folgende Kernfunktionalitäten mit Unit-Tests abgedeckt werden:

- Check, ob ein Admin die richtigen Rollen besitzt.
- Die neue Funktion im FuturæAdminApiEnrollmentServiceImpl.java welche den neusten, offenen Aktivierungscode zurück gibt:
 - Verhalten im Fehlerfall
 - Verhalten im Erfolgsfall
 - Verhalten im Erfolgsfall aber ohne ausstehender Aktivierung
- Die neue Methode im Airlock2FAAdminService.java inkl. dem Logging.

10.4.4 REST-Integration-Tests

Die REST-Integration-Tests werden auch mit hilfe von JUnit geschrieben. Anderst als bei den Unit-Tests wird hier nicht eine kleine Einheit getestet sondern der ganze Teil von der REST-Resource bis zur Datenbank. Mit den Integration-Tests wird sicher gestellt, dass das ganze Feature im Backend richtig funktioniert. Für das sind folgende zwingende Fälle zu testen:

- Es dürfen nur berechtigte Admins den Code erhalten, resp. Zugriff auf den neuen Endpunkt haben.
- Es muss eine 403 Response zurück kommen, wenn ein Admin nicht berechtigt ist.
- Falls kein offenes Enrollment existiert, der Admin aber berechtigt wäre, muss die Response leer sein.

10.4.5 UI-Integration-Tests

Zusätzlich zu den Rest-Integration-Tests werden auch UI-Integration-Tests erstellt. Diese Tests werden mit Selenium ausgeführt. Mit ihnen wird das UI / die SPA im Zusammenspiel mit dem IAM-Backend getestet. Folgende Fälle sind zu Testen:

- Hat der Admin keine Berechtigung, darf der Button gar nicht erst angezeigt werden.
- Hat der Admin die Berechtigung, es gibt jedoch keinen Activation Code, darf der Button auch nicht angezeigt werden.
- Hat der Admin die Berechtigung und es ist ein Activation Code vorhanden, muss der Button angezeigt werden.
- Wird der Button geklickt muss ein Popup aufgehen, in welchem der Activation Code steht.
- Klickt man in diesem Popup auf «Schliessen», sollte man wieder auf der Airlock 2FA Managementseite landen.

10.4.6 Manuelle Tests

Die Funktionalitäten werden durch die vielen automatisierten Tests schon ziemlich gut getestet. Allerdings ist es auch wichtig, das ganze manuell zu Testen. Der wichtigste Grund ist, dass der Futurae Server nicht mehr gemockt ist, sondern nun eine richtige Verbindung besteht. Damit es nicht zu unerwarteten Fehlern in Kombination mit Futurae kommt, sind diese Test sehr wichtig. Für die manuellen Tests sind folgende Testfälle definiert:

Testfall	M1
Testumgebung	<ul style="list-style-type: none">– IAM auf Localhost– Demokonfiguration ergänzt mit der Konfiguration des Futurae Service.
Beschreibung	Der Admin hat die nötigen Rollen, damit er den Aktivierungscode ansehen kann und es gibt ein Enrollment welches offen ist. Der Admin klickt auf den Button, welcher ihm den Aktivierungscode anzeigen soll.
Erwartetes Resultat	Es öffnet sich ein Popup (kein Browserpopup), mit dem Aktivierungscode und allenfalls dem Enrollmentdatum.

Testfall	M2
Testumgebung	<ul style="list-style-type: none">– IAM auf Localhost– Demokonfiguration ergänzt mit der Konfiguration des Futurae Service.– Postman um Request ohne SPA abzusetzen
Beschreibung	Der Admin hat die nötigen Rollen nicht, damit er den Aktivierungscode ansehen kann, es gibt aber ein Enrollment welches offen ist.

Erwartetes Resultat	Der Button wird nicht angezeigt. Der Admin darf auch via einen alternativen REST-Client, wie z.B Postman, den Aktivierungscode nicht bekommen.
----------------------------	--

Testfall	M3
Testumgebung	<ul style="list-style-type: none"> – IAM auf Localhost – Demokonfiguration ergänzt mit der Konfiguration des Futurae Service.
Beschreibung	Der Admin hat die nötigen Rollen nicht, damit er den Aktivierungscode ansehen kann und es gibt kein Enrollment welches offen ist.
Erwartetes Resultat	Der Button wird nicht angezeigt.

10.5 Qualitätssicherungskonzept

Das Qualitätssicherungskonzept wird definiert, um eine möglichst hohe Qualität des Codes zu erhalten.

Die Qualität des Codes ist im IAM-Repository schon sehr gut erzwungen. Durch Architecture Layering Tests und Checkstyle sind IAM-Spezifische Vorgaben getestet, damit der Code einheitlich bleibt. Zusätzlich gibt es auch noch PMD und Spotbugs Checks. Die PMD Checks prüfen den Code auf unschönheiten, basierend auf IAM-spezifischen Regeln. Das selbe gilt für SpotBugs.

Durch all diese Tests wird die Qualität sehr gut sichergestellt. Bei jedem Push eines neuen Patchsets werden diese Checks, zusätzlich zu den normalen Tests, auch in der Jenkinipeline ausgeführt.

11 Entscheiden

Dieses Kapitel bietet Platz, für die IPERKA-Phase «Entscheiden». Es werden die verschiedenen Vorschläge aus der Planungsphase evaluiert. Das Ziel dieser Phase ist es, sich für den besten Weg zu entscheiden.

11.1 REST-Interface Backend

In Abschnitt 10.2.1 wurden folgende 2 Lösungsvarianten für die REST-Schnittstelle definiert:

- Option 1: Den bestehenden Endpunkt, welcher den Account des Nutzers zurück gibt um das Activationcode Feld erweitern.
- Option 2: Einen neuen separaten Endpunkt welcher nur den Activationcode, und eventuell noch weitere Daten wie das Enrollment Datum zurück gibt.

Die 2 Varianten werden basierend auf folgenden Eigenschaften verglichen:

Flexibilität Je flexibler der Activationcode angefragt werden kann, desto besser. Dadurch kann er einfach durch weitere Infos ergänzt werden.

Beachtung der Rollen Zugriffskontrolle basierend auf den Rollen des Adminnutzers kann einfach eingeschränkt werden.

Aufwand Der Aufwand sollte sich im geplanten Rahmen halten, da nur begrenzt Zeit vorhanden ist.

In der folgenden Nutzwertanalyse werden die beiden Varianten miteinander verglichen und jeweils mit 0-10 Punkten bewertet, welche mit der Gewichtung des Kriteriums multipliziert werden:

Kriterium und Gewichtung	Option 1 (Endpunkt erweitern)	Option 2 (Neuer Endpunkt)
Flexibilität(20%)	2(0.4)	10(2.0)
Beachtung der Rollen(50%)	5(2.5)	10(5.0)

Aufwand(30%)	10(3.0)	5(1.5)
Total	5.9	8.5

Aufgrund des Resultats dieser Analyse wird ein neuer Endpunkt erstellt, um die Informationen des 16-stelligen Aktivierungscode an die SPA zu übermitteln.

11.2 SPA-Design

In Abschnitt 10.3 wurden folgende 3 Lösungsvarianten für das UI der SPA vorgeschlagen:

- Popup nur mit Aktivierungscode
- Popup mit Aktivierungscode und Enrollment Datum
- Darstellung in Accountübersicht mit Auge

Folgende Eigenschaften dienen als Grundlage zum Vergleich der 3 verschiedenen Lösungsvarianten:

Aussehen Die neue Komponente fügt sich einwandfrei in das bestehende UI ein. Dafür werden in der Adminapp bereits bekannte Komponenten verwendet.

Verhalten Die neue Komponente verhält sich analog zu bereits implementierten Features in der Adminapp.

Aufwand Der Aufwand sollte sich in Grenzen halten, da nur begrenzt Zeit vorhanden ist.

In der folgenden Nutzwertanalyse werden die beiden Varianten miteinander verglichen und jeweils mit 0-10 Punkten bewertet, welche mit der Gewichtung des Kriteriums multipliziert werden:

Kriterium und Gewichtung	Popup ohne Datum	Popup mit Datum	Auge
Aussehen(40%)	10(4.0)	10(4.0)	2(0.8)
Verhalten(40%)	10(4.0)	10(4.0)	2(0.8)
Aufwand(20%)	10(2.0)	5(1.0)	3(0.6)
Total	10	9	2.2

Aufgrund des Resultats dieser Analyse wird das bestehende UI um ein Button erweitert, welcher beim anklicken ein Popup öffnet, in welcher der Aktivierungscode angezeigt wird.

12 Realisieren

In diesem Kapitel wird die IPERKA-Phase Realisieren dokumentiert. Es werden die wichtigsten und entscheidendsten Punkte während der Implementierung festgehalten.

12.1 Backend erweitern

In diesem Abschnitt wird die Implementierung des Backends dokumentiert.

12.1.1 Neuer REST-Endpunkt

Als ersten Schritt wurde der neue REST-Endpunkt erstellt:

```
@JsonApiResource(attributes =
    Airlock2FAShortActivationCodeData.class)
@Path("activation-code-short")
public Response getShortActivationCode (@ExistingUser
    @PathParam("userId") @Parameter(schema = @Schema(type =
        "string")) UserParam userParam) {
    return ok(new
        Airlock2FAShortActivationCodeData(null)).build();
}
```

Der Endpunkt sieht aktuell so aus. Es sind noch keinerlei Funktionalitäten implementiert. Daher wird auch hardcoded null zurückgegeben. `Airlock2FAShortActivationCodeData.java` ist das DTO welches ein nullable Feld «short_activation_code» enthält.

Rolebased Access Control

Damit der Zugriff auf den neuen Endpunkt nur dann funktioniert, wenn der Admin die nötigen Rollen dazu besitzt, musste eine neue `RestAction` definiert werden. Diese wird in der Klasse `RestActionsDefinitions.java` folgendermassen erstellt:

```
public static RestAction viewAirlock2FAActivationCode () {
    return RestAction
        .builder()
        .action(viewAirlock2FAActivationSecret)
        .rule(Rule.of(GET,
            "/users/[^/]+/tokens/airlock-2fa/activation-code-short"))
        .build();
}
```


Dies bewirkt nun, dass die Action «viewAirlock2FA Secrets», welche es schon gab, auf diesen Pfad matched. Das heisst bei jedem Call auf den neuen Endpunkt, wird zuerst validiert, ob der Nutzer die richtigen Rollen hat, welche für diese Action benötigt werden, ansonsten wird 403 zurückgegeben. Ein Problem hat sich nun hervorgehoben. Es gibt bereits folgendes Pattern:

```
«.rule(Rule.of(GET, "text/users/[^/]+/tokens/airlock-2fa(/.*)""))»
```

Dieses Pattern matched auch auf den neuen Pfad. Da dieses Pattern in der allgemeine Action «viewToken » definiert wurde, könnte es nun zu Konflikten kommen. Deshalb wurde der neue Pfad in diesem Pattern mit Hilfe eines «Negative Lookaheads» ausgeklammert. Das neue Pattern für die viewToken Action sieht nun so aus:

```
/users/[^/]+/tokens/airlock-2fa(?!/activation-code-short(?:/|$))(/.*)"?\$
```

So kann die viewAirlock2FAActivationSecrets Action unabhängig von der viewToken Action konfiguriert werden. Es entstehen dementsprechend keine Fehlkonfigurationen und Überschreibungen der Mappings.

REST-Dokumentation

Eine Anforderung an neue REST-Endpunkte ist deren Dokumentation. Zum Zeitpunkt der Probe-IPA wird diese von Miredot via Javadoc generiert. Sprich Miredot erstellt eine REST-Dokumentation, basierend auf dem Javadoc. Für den neuen Endpunkt setzt sich diese Dokumentation wie folgt zusammen:

The screenshot shows the Miredot REST documentation for the 'Airlock 2FA' endpoint. At the top, it states 'RETRIEVES THE AIRLOCK 2FA SHORT ACTIVATION CODE OF THE LATEST PENDING ENROLLMENT FOR THE CURRENT USER.' Below this is a description: 'Retrieves the Airlock 2FA short activation code of the latest pending enrollment for the current user.' The main part of the documentation is a green box containing the HTTP method 'GET' and the URL 'https://internal-iam-host.com/auth-admin/rest/users/{userId}/tokens/airlock-2fa/activation-code-short/'. Below this, under 'Path parameters', it lists 'userId' as a 'string' parameter with the description 'The identification of the user.'

Abbildung 12.1: Miredot REST-Dokumentation Request

Zu oberst ist immer der Titel des Requests. Darunter folgt eine Kurze Zusammenfassung. Danach ist der Pfad dargestellt mit der entsprechenden HTTP-Methode. Zum Schluss folgen die Pfadparameter, was in diesem Fall die User ID ist. Anschliessend folgt die Response:

Returns

A resource document with the Airlock 2FA short activation code of the user. The Response can be empty, if no pending enrollment was found.

Content-Type: `application/vnd.api+json,application/json`

```
{
  data: // The primary data of the response.
  {
    attributes: // Attributes of this resource.
    {
      shortActivationCode: string // The short activation code. [...]
    }
    id*: string // Identifies the resource.
    type*: string // "user.token.airlock-2fa.short-activation-code".
  }
}
```

Status codes

200 OK	Retrieved Airlock 2FA short activation code.
401 Unauthorized	Invalid or missing authentication.
403 Forbidden	Access to the requested service is forbidden. Authentication will not help.
404 Not Found	Possible error codes: ACCOUNT_NOT_FOUND The user has no Airlock 2FA account. USER_NOT_FOUND The user does not exist.
500 Internal Server Error	The service call did not succeed.
501 Not Implemented	The active configuration does not support the requested operation.
Always expect generic error responses as documented here .	

Abbildung 12.2: Miredot REST-Dokumentation Response

Die Response der IAM-Endpunkte ist immer gleich aufgebaut. Sie verfügen über ein Data-Objekt, welches das Attributes-Objekt mit den jeweiligen geforderten Daten enthält. Nebst diesem Objekt hat es zwei weitere Felder, die Id, welche zur Identifizierung der Response gilt und zum anderen, der Type der Response.

12.1.2 Requests zu Futurae

Damit die SPA den 16-stelligen Aktivierungscode wie geplant angeboten bekommt, müssen in der Kommunikation zwischen IAM und Futurae einige Erweiterungen getroffen werden.

16-stelliger Aktivierungscode erzwingen

Damit der 16-stellige Aktivierungscode bei dem Request, welcher das neuste offene Enrollment sucht, auch zurück kommt musste zuerst sichergestellt werden, dass bei dem Start-Enrollment-Request das Property «shortcode» auf true gesetzt wird. Dafür wurde auf dem Request Objekt ein weiteres Feld hinzugefügt.

Hier galt es zu beachten, dass jeweils 2 verschiedene Enrollment Requests gemacht werden:

- Für einen neuen Nutzer, sprich ohne Account.
- Für einen Nutzer welcher schon einen Account besitzt.

Dies musste sowohl für den Adminapp API Call zu Futurae als auch den Loginapp Call gemacht werden.

Nebst dem Request musste natürlich auch die Response geändert werden und um dieses Feld erweitert werden.

Zum serialisieren und deserialisieren des Requests- / Response-Body wird Jackson verwendet. Dabei werden mit einfachen Annotationen die Java-Objekte auf die verlangten JSON-Felder gemapped oder umgekehrt:

```
@NoArgsConstructor
@Getter
@Setter
@JsonInclude(Include.NON_NULL)
public class FuturaeAuthApiEnrollmentRequest {

    @JsonProperty("user_id")
    private String userId;
    @JsonProperty("username")
    private String username;
    @JsonProperty("display_name")
    private String displayName;
    @JsonProperty("valid_secs")
    private Integer validSecs;

    @JsonProperty("short_code")
    private Boolean shortCode;

    @JsonProperty("success_callback_url")
    private String successCallbackUrl;
    @JsonProperty("phone_number")
    private String phoneNumber;
    @JsonProperty("enrollment_flow_binding_enabled")
    private Boolean enrollmentFlowBindingEnabled;
    @JsonProperty("account_recovery_flow_binding_enabled")
    private Boolean recoveryFlowBindingEnabled;
}
```

Beispiel, wie Jackson auf dem Futurae Request zur Auth API (für die Self-Services) verwendet wird. (Enthält bereits das neue Feld «short_code»).

Erweitern der Requestfactory

Als nächster Schritt wurde die `FuturaeAdminApiEnrollmentRequestFactory.java` um eine neue Methode erweitert:

```
public RestRequest getLatestPendingEnrollment (String
    airlock2FAAccountId) {
    return
        futuraeRequestFactory.createGetRequest(ENROLLMENTS.usersP
            getQueryParams(airlock2FAAccountId));
}

private static Map<String, Object> getQueryParams (String
    airlock2FAAccountId) {
    return Map.of("user_id", airlock2FAAccountId,
        "status", "pending",
        "sort_by", "created_at",
        "order", "desc",
        "limit", "1");
}
```

Diese Methode «`getLatestPendingEnrollment`» mit der statischen Hilfsmethode «`getQueryParams`» baut einen GET Request mit folgendem Pfad zusammen:

```
/srv/admin/v1/enrollments?user_id=userid
&status=pending
&sort_by=created_at
&order=desc
&limit=1
```

Dies ist der Request, welcher ausgeführt werden muss, um das neuste offene Enrollment zu bekommen.

Erweitern des Enrollment Service

Damit die Response von Futurae richtig geparkt werden kann, musste zuerst ein äquivalentes Java Entity Objekt erstellt werden:

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@JsonIgnoreProperties(ignoreUnknown = true)
public class FuturaeAdminEnrollmentListResponse {

    @NotNull
    @JsonProperty("count")
    private Integer count;
```

```

    @NotNull
    @JsonProperty("enrollments")
    private List<FuturaeAdminEnrollmentResponse>
        enrollmentResponseList;
    @NotNull
    @JsonProperty("limit")
    private Integer limit;
    @NotNull
    @JsonProperty("offset")
    private Integer offset;
    @NotNull
    @JsonProperty("total")
    private Integer total;
}

```

Dieses Objekt bildet folgende JSON-Response, welche von Futurae so definiert wurde, in Java ab:

```

{
  "count": 0,
  "enrollments": [{FuturaeAdminEnrollmentResponse}],
  "limit": 0,
  "offset": 0,
  "total": 0
}

```

Der FuturaeAdminApiEnrollmentServiceImpl.java ist die Schnittstelle zwischen IAM und Futurae, für alle Aktionen, welche das Enrollment betreffen. Für das wurde im seinem Interface die Methode «getLatestPendingEnrollment» definiert. Die Implementation sieht folgendermassen aus:

```

@Override
public Optional<Airlock2FAEnrollment>
    getLatestPendingEnrollment (Airlock2FAAccountId accountId) {
    RestRequest request =
        adminApiRequestFactory
            .getLatestPendingEnrollment(accountId.getId());
    ...
}

```

Zuerst wird der Request über die vorhin erstellte Methode zusammengestellt. Danach wird er via den RestClient ausgeführt. Die Response wird anschliessend im folgenden Responsehandler behandelt:

```

...
return restClient.read(request, response -> {
    if (response.getStatusInfo().equals(BAD_REQUEST)) {

```

```

        LOG.info("Invalid parameters for request to fetch
            latest pending enrollment for account with id: " +
                accountId.getId());
        return Optional.empty();
    }
    if (!response.getStatusInfo().equals(OK)) {
        LOG.info("Request to fetch the latest pending
            enrollment for account with id '" +
                accountId.getId() + "' failed with code: " +
                response.getStatusInfo());
        return Optional.empty();
    }
    FuturaeAdminEnrollmentListResponse result =
        readResponse(response,
            FuturaeAdminEnrollmentListResponse.class, request)
        .orElseThrow(Airlock2FAAdminApiException::new);
    if (result.getEnrollmentResponseList().isEmpty()) {
        LOG.info("Could not find a pending enrollment for
            account with id: " + accountId.getId());
        return Optional.empty();
    }
    LOG.info("Latest pending Enrollment fetched successfully
        for account with id: " + accountId.getId());
    return Optional.of(mapEnrollment(result
        .getEnrollmentResponseList().getFirst()));
});

```

Falls die Response einen 400 beinhaltet wird eine Info geloggt und `Optional.empty` zurück gegeben. Dieser Fall sollte eigentlich nicht auftreten, es sei denn, das Futurae API ändert sich. Denn dieser Fehler kommt nur dann, wenn die Parameter falsch sind. Diese sind Hardcoded, können also nicht vom Nutzer beeinflusst werden. Um unerwartete Fehler zu vermeiden wird dieser Fall hier dennoch explizit abgefangen.

Da dies der einzige von Futurae definierte Error für diesen Request ist, werden die anderen mit dem darauf folgenden if-Statement abgefangen.

Danach wird in einem bereits bestehenden `ResponseReader` die Response in das Erwartete Objekt geparkt. Ist das zurückkommende Result präsent, wird geprüft, ob es Einträge in der Enrollmentliste gibt. Falls ja wird ein `Optional` vom ersten Eintrag genommen. Dies geht, da man davon ausgehen kann, dass entweder genau 0 oder 1 Eintrag vorhanden ist. Dies weil im Request der «limit» -Parameter auf 1 gesetzt wurde.

Dieses `Optional` wird dann über den `Airlock2FAAdminService.java` an die Resource überliefert, welche dann schlussendlich den 16-stelligen Aktivierungscode oder nichts, falls nichts vorhanden, als `ResourceDocument` zurückgibt.

12.1.3 Logging

Das Logging muss die Anforderung «FA 3» , definiert in Abschnitt 9.3.1, erfüllen. Hierfür wurde im Airlock2FAAdminService.java folgende Methode implementiert:

```
public Optional<String>
    findLatestPendingShortActivationCode
    (Airlock2FAUserAccount account) {
    return adminApiEnrollmentService
    .getLatestPendingEnrollment(account.getAccountId())
    .map(enrollment -> {
        signingLogger.log(format("Administrator '%s' accessed
            the Airlock 2FA short activation code of user
            '%s'.", administrator.getName(),
            account.getUserId()));
        helpdeskLogger.log(account.getUserId(),
            format("Administrator '%s' accessed the Airlock 2FA
            short activation code.", administrator.getName()));
        return enrollment.getActivationCodeShort();
    })
    .or(Optional::empty);
}
```

In dieser Methode wird aus dem erhaltenen Optional Enrollment der 16-stellige Aktivierungscode herausgefiltert und je nach Fall, ob präsent oder nicht, wird ein Optional String oder Empty zurückgegeben. Da dies der Service ist, welcher der Resource dient, in welcher der neue Endpunkt ist, wurde hier dann das Logging eingebaut. Das heisst diese Methode ist weniger allgemein, als jene im Enrollment-Service und es kann davon ausgegangen werden, dass sie nur für diesen Zweck verwendet wird. Der Signing Logger ist für das Audit Log, diese Message darf deshalb auch technisch sein und muss nicht für jeden verständlich sein. Damit es auch im User Activities Logfile nachvollziehbar ist, welcher Admin wann auf einen Aktivierungscode zugegriffen hat, wurde der Helpdesklogger verwendet. In der Adminapp im Activities-Tab des aktuellen Nutzers, sieht dies nun wie folgt aus:

13.11.2024 14:39:10 (UTC+01:00)	uid:itester - provided_uid:admin - Administrator 'admin' accessed the Airlock 2FA short activation code. Level: INFO - Session: 821477658274758698 - Request: 496622032337144729 - IP: 0.0.0.0:0.1
---------------------------------	---

Abbildung 12.3: Logeintrag bei vorhandenem Aktivierungscode

Im obigen Bild links, ist der Zeitpunkt, zudem der Admin auf den Code zugegriffen hat, zusehen. Auf der rechten Seite der Text, welcher im obigen Code im ersten Logstatement steht.

12.1.4 Resultat

Die oben aufgezeigten Schritte führten schlussendlich zum gewollten Resultat. Der Endpunkt liefert nun den neusten, offenen Aktivierungscode, falls er vorhanden ist

```
{
  "meta": {
    "type": "jsonapi.metadata.document",
    "timestamp": "2024-11-14T13:12:29.093+01:00"
  },
  "data": {
    "type": "user.token.airlock-2fa.short-activation-code",
    "id": "2600345384",
    "attributes": {
      "shortActivationCode": "wdgj zjt4 m5h2 faxw"
    }
  }
}
```

und sonst eine leere Response.

```
{
  "meta": {
    "type": "jsonapi.metadata.document",
    "timestamp": "2024-11-14T13:10:25.036+01:00"
  },
  "data": {
    "type": "user.token.airlock-2fa.short-activation-code",
    "id": "0",
    "attributes": {
      "shortActivationCode": ""
    }
  }
}
```

Der Endpunkt ist durch den Role-Based Access Controller geschützt, sodass nur Admins mit den nötigen Rollen Zugriff haben. Alle anderen erhalten einen 403. Jeder erlaubte Zugriff wird nach den Anforderungen geloggt.

12.1.5 Tests

Die Unit-Tests so wie die Integration-Tests werden nach dem AAA(Arrange, Act, Assert)-Pattern gegliedert. Dies bedeutet, dass im ersten Abschnitt(Arrange) alles was für den Test gebraucht wird zusammengestellt wird. Im zweiten Abschnitt(Act) wird die zu testende Methode, resp. in den Integration-Tests der zu testende Endpunkt, aufgerufen. Und in einem dritten Abschnitt(Assert), wird das erwartete Resultat mit dem tatsächlichen

Resultat verglichen. Dies dient der Übersicht und der Leserlichkeit des Tests. In einigen Fällen kann es sein, dass der erste Schritt weg fällt, da z.B nichts arranged werden muss, oder auf der Testklasse bereits arranged wurde. Nach jedem der Abschnitte kommt jeweils eine leere Zeile.

Ausbau Api Stubber

Wie in der Planung unter 10.4.2 bereits angesprochen, laufen die Tests nicht gegen das originale Futurae API, sondern gegen einen mit Wiremock simulierten Server.

Da der Endpunkt, der für den neuen Call zu Futurae (12.1.2) angesprochen wird, auf dem Stubber noch nicht existiert, muss dieser zuerst erstellt werden. Dabei wird der Pfad definiert inkl. allen Queryparametern und der Response. Da als Response nicht immer das selbe erwartet wird, gibt es 2 verschiedene Methoden im Stubber:

```
public FuturaeAdminApiStubber withPendingEnrollmentList
    (String accountId, int limit) {
    wireMockClient.register(
        get(urlPathEqualTo(FUTURAE_ADMIN_ENROLLMENTS_PATH))
        .withQueryParams(getQueryParams(accountId, limit))
        .willReturn(aJsonResponse(
            defaultEnrollmentList(List.of(
                defaultEnrollment(DEFAULT_ACTIVATION_CODE)
                .activationCodeShort(DEFAULT_ACTIVATION_CODE_SHORT)
                .toJsonObject()), limit)
            .toJsonObject()))
        );
    return this;
}

public FuturaeAdminApiStubber withEmptyPendingEnrollmentList
    (String accountId) {
    wireMockClient.register(
        get(FUTURAE_ADMIN_ENROLLMENTS_PATH)
        .withQueryParams(getQueryParams(accountId, 1))
        .willReturn(aJsonResponse(emptyEnrollmentList())
            .toJsonObject()))
    );
    return this;
}

private static Map<String, StringValuePattern> getQueryParams
    (String airlock2FAAccountId, int limit) {
    return Map.of("user_id", equalTo(airlock2FAAccountId),
        "status", equalTo("pending"),
        "sort_by", equalTo("created_at"),
```

```

    "order", equalTo("desc"),
    "limit", equalTo(valueOf(limit)));
}

```

Die erste gibt eine Response, welche erfolgreich ist zurück. Die zweite eine Response in der die Enrollment Liste leer ist, heisst es wird simuliert, dass kein offenes Enrollment existiert. Diese 2 Fälle reichen aus, um das neue Feature zu testen. Dieser Stubber wird für alle REST-Integration-Tests verwendet. Für die Unit-Tests wird direkt im Test gestubbed. Das Prinzip ist aber das gleiche.

Integration-Tests

Für die Integration-Tests ist wichtig, dass eine IAM-Instanz läuft. In jedem Test wird eine IAM-Konfiguration zusammengestellt, welche die für die Tests notwendigen Plugins einkonfiguriert hat:

```

private IamConfig adminApp (String
    allowedToViewAirlock2FAActivationSecrets) {
    return iamConfig()
        .newAdminapp()
        .addRoleBasedAccessControl(roleBasedAccessControl()
            .viewAirlock2FAActivationSecret(
                allowedToViewAirlock2FAActivationSecrets))
        .withStartPages(MANAGE_USER)
        .addTokenController(
            airlock2FATokenController(futuraeApiUrl)
                .build())
        .build();
}

```

In diesem Fall sieht es so aus. Diese Methode hat einen Parameter um die Role Based Access Control je nach Testcase zu beeinflussen. Vor jedem Test wird nun diese Config geladen und Aktiviert, es sei denn, es werden Custom Configs verwendet. Dieser Fall ist in Kapitel 12.1.6 beschrieben.

Hier ist ein Beispiel wie die zuvor erstellten Stubbermethoden in einem Test genutzt werden können:

```

@Test
void shouldGetAirlock2FAShortActivationCode
(FuturaeAdminApiStubber stubber) {
    addAirlock2FAAccount(DEFAULT_USERNAME);
    stubber.withPendingEnrollmentList(defaultAccount().userId().
        1);

    JsonApiResponseResponse<Airlock2FAShortActivationCodeAttribute>
        response =
            getAirlock2FAShortActivationCode(DEFAULT_USERNAME);
}

```

```

    assertThat(response)
    .satisfies(r -> {
        assertThat(r.getData().getAttributes()).isNotNull();
        String actualCode =
            r.getData().getAttributes().getShortActivationCode();
        assertThat(actualCode).isNotNull();
        assertThat(actualCode)
            .isEqualTo(DEFAULT_ACTIVATION_CODE_SHORT);
    })
    .hasStatus(OK);
}

```

Zuerst wird ein neuer Airlock2FA Account hinzugefügt. Danach kommt die neue Stubbermethode zum Einsatz. Sie registriert auf dem Server den in der Methode definierten Endpunkt. Der Test macht danach den Call auf den neuen Endpunkt im IAM. Dieser löst dann den Request zum Mock-Server aus, welcher die definierte Response liefert. Am Schluss wird die Response, des IAM-Endpunktes, verifiziert. So kann sichergestellt werden, ob der neue Endpunkt wie erwartet funktioniert.

Unit-Tests

Für den Unit-Test des FuturaeAdminApiEnrollmentServiceImpl.java musste der Futurae Server auch gemockt werden. Denn, die Aktionen dieses Services sind von den Antworten des Futurae Servers abhängig.

```

@Test
void getLatestPendingEnrollmentShouldBeEmpty () {
    withEnrollmentListResponse(EXISTING_FUTURAE_USER_ID.getId(),
        aJsonResponse()
        .withBody(emptyEnrollmentList()));

    Optional<Airlock2FAEnrollment> enrollmentOptional =
        adminApiEnrollmentService()
        .getLatestPendingEnrollment(EXISTING_FUTURAE_USER_ID);

    assertThat(enrollmentOptional).isEmpty();
}

```

Das obige Beispiel zeigt einen Unit-Test welcher sicherstellt, dass falls von Futurae eine Response ohne Enrollments zurück kommt, die Response der neuen Methode wie erwartet Optional Empty zurück gibt.

Für die Unit-Tests des Airlock2FAAdminService.java konnte schon der Airlock2FAAdminApiEnrollmentService.java gemockt werden, das heisst hier war Wiremock nicht mehr nötig. Nebst dem richtigen Resultat wurde in diesen Tests, auch das

Logging getestet. Mit der Hilfe von Mockito.verify konnte einfach verifiziert werden, ob die Logger die erwarteten Statements geloggt haben:

```
verify(signingLogger).log("Administrator 'admin' accessed the  
    Airlock 2FA short activation code of user 'user'.");  
verify(helpdeskLogger).log(account.getUserId(), "Administrator  
    'admin' accessed the Airlock 2FA short activation code.");
```

12.1.6 Testen der Access Control

Um mit automatisierten Tests sicherzustellen, dass nur berechtigte Admins Zugriff auf den 16-stelligen Aktivierungscode haben, musste zuerst die Konfiguration im REST-Integration-Test angepasst werden. Hierfür wurde zuerst im «AdminappConfigBuilder.java» eine neue Methode hinzugefügt, damit man die Role Based Access Controller Config dynamisch setzen kann. Danach musste diese im Test konfiguriert werden. Da es je nach Test eine andere Konfiguration der berechtigten Rollen braucht, musste dies dynamisch mit Parameter und Custom Configs gelöst werden. Das bedeutet, die Default-Config hat den Wert «superadmin». Dies erlaubt dem Default-User den Zugriff auf die Airlock2FA Activation Secrets. Um zu testen, ob ein Admin ohne die Erforderliche Rolle auch wirklich kein Zugriff hat, kann man nun in diesem Test die Konfiguration bearbeiten und manuell aktivieren:

```
@Test  
@CustomConfig  
void shouldGet403IfAccessNotAllowed () {  
    adminApp("helpdesk").activate();  
    addAirlock2FAAccount(DEFAULT_USERNAME);  
  
    var response =  
        getAirlock2FAShortActivationCode(DEFAULT_USERNAME);  
  
    assertThat(response).hasStatus(FORBIDDEN);  
}
```

Damit der Rest der Konfiguration gleich bleibt, wurde nur der Inhalt des Rollenfelds, für die Action «viewAirlock2FAActivationCode» als Parameter definiert. Hier wird jetzt die Default-Config überschrieben und der Inhalt auf «helpdesk» gesetzt. Da der Adminnutzer in diesem Test nur die Rolle superadmin hat, wird der Status FORBIDDEN erwartet.

12.2 SPA erweitern

Im folgenden Abschnitt wird beschrieben wie die Adminseite in der SPA erweitert wurde.

12.2.1 Request zur IAM API

Zuerst wurde ein neues Model erstellt, welches auf die JSON-Response gemapped werden kann:

```
export class Airlock2FAShortActivationCode {  
    activationCodeShort: string;  
}
```

Im Rahmen der Probe-VA ist dies ein Objekt mit einem string-Property, welches den 16-stelligen Aktivierungscode abbildet.

Danach wurden die Funktionalitäten eingebaut, um den neuen Endpunkt aus der SPA anzusteuern. Dazu wurde der «airlock-2fa.service.ts» um folgende 2 Methoden erweitert:

- `getLatestPendingActivationCode`: Diese Methode führt den Request, mit Hilfe des Http Client aus dem Angular «common» Package, aus.
- `toAirlock2FAShortActivationCode`: Dies ist die Mapper-Methode, welche die `JsonApiResponse` ausliest und in ein neues Objekt, des zuerst erstellten Model's, parsed.

Dieses Objekt wird schlussendlich von der ersten Methode «`getLatestPendingActivationCode`» zurückgegeben.

12.2.2 Darstellung im UI

Damit der 16-stellige Aktivierungscode wie erwartet im Popup angezeigt wird, musste die bestehende «airlock-2fa»-Komponente im HTML für die Struktur und im Typescript für die Logik, erweitert werden.

airlock-2fa.component.html

Zuerst wurde das HTML erweitert. Es wurde der neue Button hinzugefügt, welcher das Popup triggert:

```
<iam-button  
    *ngIf="isCodePresent"  
    [id]="viewShortActivationCodeButtonId"  
    [hideOnAccessDenied]="true"  
    [text]=" 'Activation code' "  
    [right]="true"  
    [outsideOfCard]="true"  
    (buttonClick)="viewActivationCode()">  
</iam-button>
```

Ohne Implementation des Typescript's funktioniert dieser Button natürlich noch nicht.

airlock-2fa.component.ts

Im Typescript wurde die ganze Logik des Buttons implementiert. Sprich, wann darf er angezeigt werden und was passiert, wenn man in anklickt.

Damit sichergestellt wird, dass der Button nur angezeigt wird, wenn der Admin auch die erforderlichen Rollen hat, gibt es auf der «iam-button» Komponente ein Feld, namens «hideOnAccessDenied». Wird dieses Feld auf true gesetzt, wird der Button nur angezeigt, wenn, die in Airlock2FATokenPermissions.java definierte Action, für diesen Button von dem Access Controller für diesen Admin erlaubt wird:

```
private void addActivationCodePermissions
(PagePermissionsBuilder page) {
    page.permission("viewShortActivationCode", accessController
        .isActionAllowed(viewAirlock2FAActivationSecret));
}
```

Dieses Mapping funktioniert über die Button-ID. Sprich der Key der Permission muss den selben Stringwert haben, wie die ID des Buttons. Doch wie in der Anforderung «FA 8» in Kapitel9.3.2 beschrieben, sollte der Button auch nur angezeigt werden, falls auch ein offener Aktivierungscode existiert. Für das wird in der ngOnInit-Methode, falls der Admin die nötigen Rollen hat, der Request ausgeführt, und falls die Response nicht leer ist, das boolean-Flag «isCodePresent» auf true gesetzt. Um den Request auszuführen wird die zuvor im Service erstellte Methode aufgerufen. Der 16-stellige Aktivierungscode wird bewusst nicht im ngOnInit lokal in die Komponente gespeichert, da es sein kann, dass er plötzlich nicht mehr offen ist, oder nicht mehr der neuste ist. Der Aktivierungscode wird deshalb immer neu angefragt wenn der Button geklickt wird. Mit anderen Worten, im ngOnInit wird nur validiert, ob der Button angezeigt werden darf oder nicht. Wird der Button geklickt, führt er folgende Methode aus:

```
viewActivationCode () {
    this.findPendingActivationCode().subscribe(
        res => {
            this.pageDialog.show({
                showCancel: false,
                okText: 'Close',
                title: '16-digit activation code',
                text: this.isCodePresent ?
                    this.languageService
                        .translate('Activation code:')
                    + ' '
                    + res.activationCodeShort :
                    'Code is not pending anymore.'
            });
        }
    );
}
```

Diese führt zuerst wieder den Request aus. Sobald eine Antwort zurück kommt, wird der Dialog angezeigt. Das Aussehen und Verhalten des Dialog's kann ein Stück weit über die PageDialogSettings, welche der show-Methode übergeben werden, beeinflusst werden. So sorgt bspw. «showCancel: false» dafür, dass nur der Ok Button angezeigt wird, welcher bei einem Klick, das Popup schliesst. Die PageDialog Komponente existierte bereits. Jedoch hatte sie gerade bei diesem Feld noch einen Bug, welcher noch nicht entdeckt wurde, da dieses Feld noch nirgends, versucht worden ist zu überschreiben. Das Problem war folgendes:

Die Booleans wurden in der «show»-Methode bisher wie folgt versucht zu überschreiben:

```
this.showCancel = settings.showCancel || true;
```

Dies hat zwar für alle Stringvalues gut funktioniert, denn bei einem String kann so direkt geprüft werden, ob er undefined, empty oder null ist. Falls ja, wird der Defaultwert verwendet. Bei einem Boolean funktioniert das nicht, da dieser false als einen gültigen Wert hat. Sprich setzt man showCancel beim Aufruf auf false, wird hier automatisch true verwendet. Dies weil das erste Statement false ist und der Operator || auf falsy-Werte prüft, welches unter anderem auch der Wert false ist. Die Lösung für dieses Problem war, dass man den Boolean mit dem Nullish Coalescing Operator «??» explizit nur auf null und undefined prüft, und nicht auf andere falsy Werte, wie false:

```
this.showCancel = settings.showCancel ?? true;
```

Also falls dieses Flag nicht von aussen gesetzt wird, dann ist es undefined, und wird auf den Standardwert true gesetzt. Ansonsten, falls es von aussen gesetzt wird, wird dieser Wert verwendet.

Der Inhalt des Dialogs, wird über das erforderliche Feld «text» gesetzt. Hier wird dann die Antwort aus dem Request, sprich der 16-stellige Aktivierungscode, eingefüllt und somit im Popup angezeigt.

```
text: 'Activation code:'  
      + ' '  
      + res.activationCodeShort
```

Falls nun der Button geklickt wird, der Aktivierungscode aber im Hintergrund bereits nicht mehr offen ist, wird dies in Form von einem Page-Alert gelöst:

```
this.pageAlertService.error('The activation is not pending  
any more.').display();
```

Dieser Alert sieht im UI dann so aus:

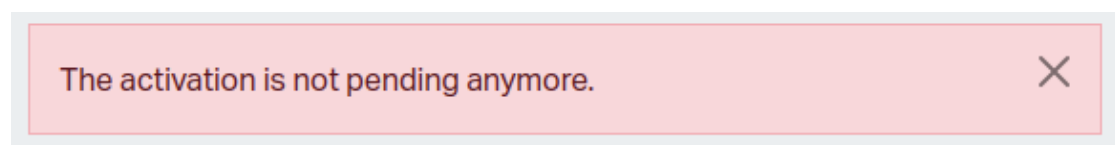


Abbildung 12.4: Page Alert, falls der Aktivierungscode bereits nicht mehr offen ist.

12.2.3 i18n Translation Keys

Da die Admin-App die 3 Sprachen, Deutsch, Englisch und Französisch unterstützt, können die Texte nicht, so wie sie aktuell sind, «hardcoded» gesetzt werden. Alle Texte, also Titel, Popuptext oder der Text im Button, müssen über Translation Keys verfügen. Für jede Sprache gibt es ein JSON-File, welches zu diesen Keys jeweils den richtig übersetzten Wert hat. Mithilfe des Language Service, können diese dann auf die gewünschte Sprache eingefügt werden. Das Json für die deutsche Sprache sieht so aus:

```
{
  ...
  "user": {
    ...
    "airlock-2fa"{
      ...
      "activation": {
        "view-activation-code": {
          "button": "Aktivierungs-Code anzeigen",
          "popup": {
            "close": "Schliessen",
            "title": "16-stelliger Aktivierungs-Code",
            "text": {
              "code": "Aktivierungs-Code:",
              "not-pending": "Die Aktivierung ist nicht mehr offen."
            }
          }
        }
      }
    }
  }
}
```

Dies ist die Struktur für die folgenden Keys:

- user.airlock-2fa.activation.view-activation-code.button
- user.airlock-2fa.activation.view-activation-code.popup.close
- user.airlock-2fa.activation.view-activation-code.popup.title
- user.airlock-2fa.activation.view-activation-code.popup.text.code
- user.airlock-2fa.activation.view-activation-code.popup.text.not-pending

Am Beispiel des Popup Textes sieht man, wie diese verwendet werden:

- Ohne Keys:

```
text: 'Activation code:'  
+ ' '  
+ res.activationCodeShort
```

- Mit Keys:

```
this.languageService  
  .translate('user.airlock-2fa.activation  
            .view-activation-code.popup.text.code')  
+ ' '  
+ res.activationCodeShort
```

Im tatsächlichen UI sieht das ganze mit den Keys schlussendlich so aus:

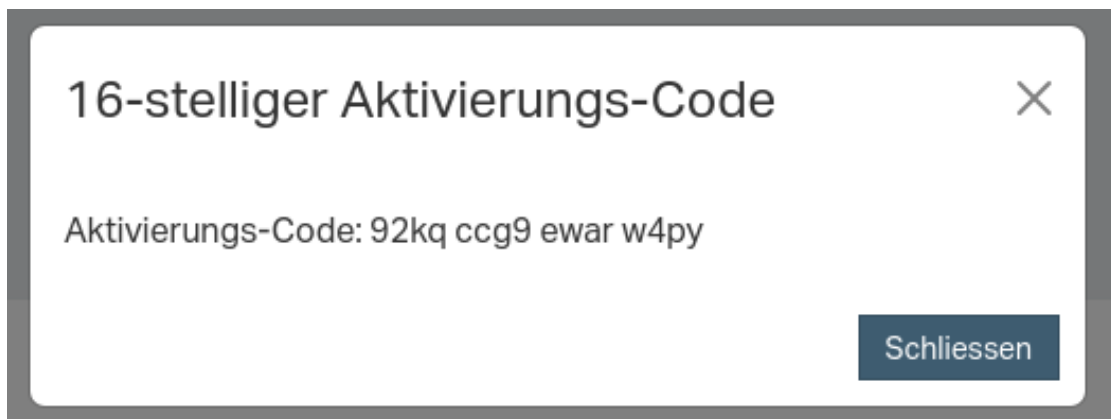


Abbildung 12.5: Popup deutsch



Abbildung 12.6: Popup englisch



Abbildung 12.7: Popup französisch

Wie man sieht, wird der Text im Popup nun immer auf die aktive Sprache der SPA übersetzt.

12.2.4 UI Integration-Tests

Die UI Integration-Tests werden im IAM mit der Hilfe von Selenium durchgeführt. Durch dies wird das komplette Feature von der SPA bis zum Backend / der DB (Futuræ (Wiremock)-Server) und wieder zurück getestet.

Wie bei den REST-Integration-Tests braucht es auch bei den UI-Integration-Tests für jeden Tests eine Config und einen Default User. In diesem Fall sind diese dieselben wie in Kapitel 12.1.5 beschrieben.

Java Component erweitern

Damit die Testklasse frei von Seleniumlogik bleibt, werden die einzelnen SPA-Seiten in separaten Klassen abgebildet. Diese Klassen beinhalten die notwendigen Methoden mit den passenden Selektoren, um auf der Seite zu navigieren und die Seite zu asserten. Diese Methoden werden dann von den Testklassen aufgerufen. Für das neue Popup wurde, eine neue Dialog Component erstellt. Sie besitzt folgende Methoden:

- `assertNoCancelButton()`
- `assertCloseButton()`
- `assertShortActivationCode(String shortActivationCode)`
- `clickCloseButton()`

Mit Hilfe dieser Methoden konnte das ganze Popup getestet werden. Das Prinzip der Methoden ist immer das selbe. Sie selektieren ein HTML-Element mit einem Selektor erbend vom Typ «By». Auf dem selektierten Element wird dann die gewünschte Aktion durchgeführt. Bspw: `assertShortActivationCode(String shortActivationCode)`:

```
public Airlock2FAShortActivationCodeDialogComponent
    assertShortActivationCode (String shortActivationCode) {
        assertTagWithIdContains("pageDialog",
            translate("user.airlock-2fa.activation
.view-activation-code.popup.text.code")
        + " "
        + shortActivationCode);
        return this;
    }
```

Hier wird der HTML-Tag mit der ID «pageDialog» selektiert. Dies entspricht dem Popup-Element. Danach wird verglichen, ob der Wert welcher im JSON-Key steht, plus der erwartete Aktivierungscode, auch wirklich im Popup enthalten sind. Schlussendlich wird die aktuelle Component wieder zurück gegeben, um weitere Aktionen darauf zu tätigen. Nebst dieser neuen Dialog Component, muss die `UserDetailsAirlock2FATabComponent.java` auch noch um 3 Methoden erweitert werden, da der Button um das Popup zu öffnen auf dieser Component ist:

- `assertViewShortActivationCodeButtonPresent()`
- `clickViewActivationButton()`
- `reload()`

Refactoring API Stubber

Im Rahmen der Selenium Tests wurden die Methoden im API-Stubber, welche im Abschnitt 12.1.5 erstellt wurden, umgebaut. Den für die UI-Integration-Tests muss es möglich sein, für einen einzelnen Test, sprich den selben Stubber, den gleichen Endpunkt auf

verschiedene Arten zu simulieren. Deshalb wurde auf der Stubber Klasse ein neues Feld «private Optional<MappingBuilder> pendingEnrollmentList; » hinzugefügt. Dieses Feld enthält immer die aktuelle Version des Endpunktes. Danach wurde eine neue Methode hinzugefügt:

```
private void withPendingEnrollmentList (MappingBuilder
newEnrollmentListMapping) {
    if (pendingEnrollmentList.isPresent()) {
        wireMockClient
            .removeStubMapping(pendingEnrollmentList.get());
    }
    pendingEnrollmentList =
        Optional.of(newEnrollmentListMapping);
    wireMockClient.register(newEnrollmentListMapping);
}
```

Diese Methode wird nun von den beiden bisherigen Methoden aufgerufen, jedoch mit anderen Endpunktkonfigurationen. Sie stellt sicher, dass immer nur eine Version des Endpunktes registriert ist. Sie löscht im Falle, dass bereits ein Mapping vorhanden ist, das bestehende Mapping und speichert das neue. So kann in den Tests, zwischen den beiden Versionen hin und her gewechselt werden, ohne dass es zu unerwarteten Antworten kommt, weil das alte Mapping verwendet wird.

Beispiel Test

Mit Hilfe all dieser Methoden, kann nun getestet werden:

```
@Test
void
shouldRemoveViewShortCodeButtonOnUserReloadIfNoCodePendingAnymor
(FuturAeAdminApiStubber stubber) {
    stubber.withPresentPendingEnrollmentList(defaultAccount()
        .userId()
        .toString(), 1);
    UserDetailsAirlock2FATabComponent airlock2FATabComponent =
        addAirlock2FA();

    airlock2FATabComponent
        .assertViewShortActivationCodeButtonPresent(true)
        .clickViewActivationButton()
        .assertShortActivationCode(DEFAULT_ACTIVATION_CODE_SHORT)
        .clickCloseButton();

    stubber.withEmptyPendingEnrollmentList(defaultAccount()
        .userId()
        .toString());
}
```

```

    airlock2FATabComponent
      .reload()
      .assertViewShortActivationCodeButtonPresent(false);
  }

```

In diesem Test wird sichergestellt, dass falls die Aktivierung nicht mehr offen ist, und die Page reloaded wurde, der Button um das Popup zu öffnen verschwindet. Hierfür wird im ersten Abschnitt zuerst eine offene Aktivierung simuliert und assertet, dass der Button noch hier ist. Danach, im zweiten Abschnitt, wird simuliert, dass die Aktivierung nicht mehr offen ist, in dem einfach eine leere Liste von Futuræ(Wiremock-Server) zurück kommen soll. Dies ist nur möglich so, dank dem Refactoring im API Stubber. Ohne dem, würde hier nun immer noch das alte Mapping verwendet werden, und es käme keine leere Liste zurück.

Vor dem Assert, ob der Button noch hier ist, wird nun der Reload durchgeführt. Danach sollte der Button nicht mehr angezeigt werden.

12.3 Kundendokumentation

In den Anforderungen steht, dass die IAM-Kundendokumentation um das neue Feature erweitert werden soll. Da nicht mit dem Kundendokumentations-Tool SMC gearbeitet werden muss, folgt die Erweiterung in diesem Kapitel.

12.3.1 Kapitel 18.5 Airlock 2FA Configuration erweitern

Unter dem «How to verify» Abschnitt im Kapitel 18.5 wird folgender neuer Abschnitt erstellt:

16-digit activation code

The 16-digit activation code can be used to activate an Airlock 2FA account. In case a user needs support during activation, administrators can assist them using this short activation code. The provided code always corresponds to the latest pending enrollment. It is displayed in a popup that appears when clicking the following button in the **Adminapp** -> **{current user}** -> **Airlock 2FA Tab** -> **View activation code**.

Note: As this is a sensitive activation secret, access to it is regulated by the «**View Airlock 2FA Activation Secret**» action in the **Access Control**. Only administrators with the necessary permissions to view Airlock 2FA activation secrets can access this short activation code.

13 Kontrollieren

Dieses Kapitel bietet Rahmen für die Arbeiten, welche in der IPERKA-Phase «Kontrollieren» angefallen sind. In dieser Phase wird die Qualität der Umsetzung geprüft. Es werden alle im Test und Qualitätssicherungskonzept definierten Checks durchgeführt. Falls Fehler auftreten, werden diese behoben.

13.1 Durchführen der Tests

In diesem Abschnitt sind die Ergebnisse der Tests dokumentiert. Die durchgeführten Tests sind im Testkonzept in Kapitel 10.4 aufgeführt.

Testdatum	18.11.2024 & 20.11.2024
Testperson	Niculin Steiner

13.1.1 Unit-Tests

Alle geschriebenen Unit-Tests sollen erfolgreich durchlaufen. Folgende 2 Services hatten Logik welche mit Unit-Tests abgedeckt werden musste:

- FuturaeAdminApiEnrollmentServiceImpl.java

*Die rot umrahmten Tests wurden im Rahmen der Probe-IPA erstellt.

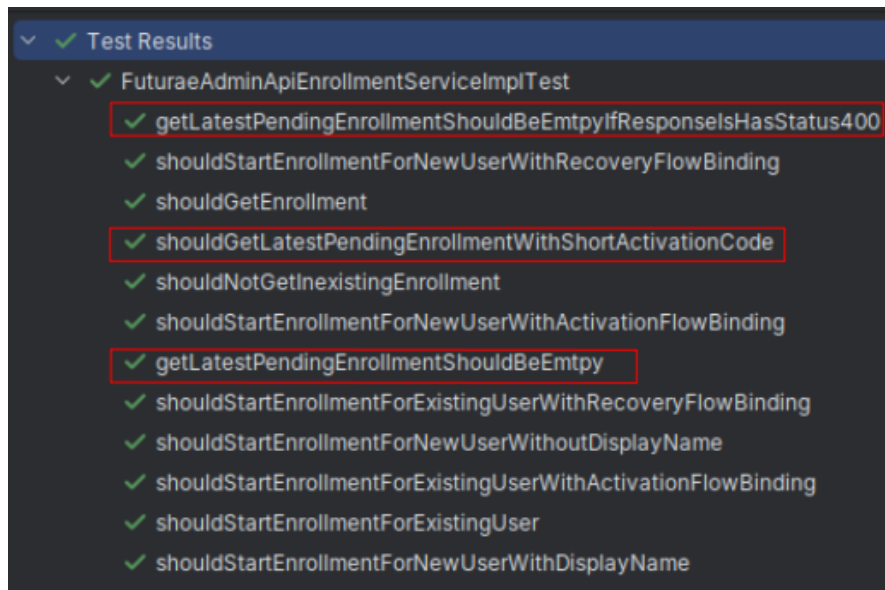


Abbildung 13.1: Unit-Test Resultate FuturaeAdminApiEnrollmentServiceImpl.java

– Airlock2FAAdminService.java

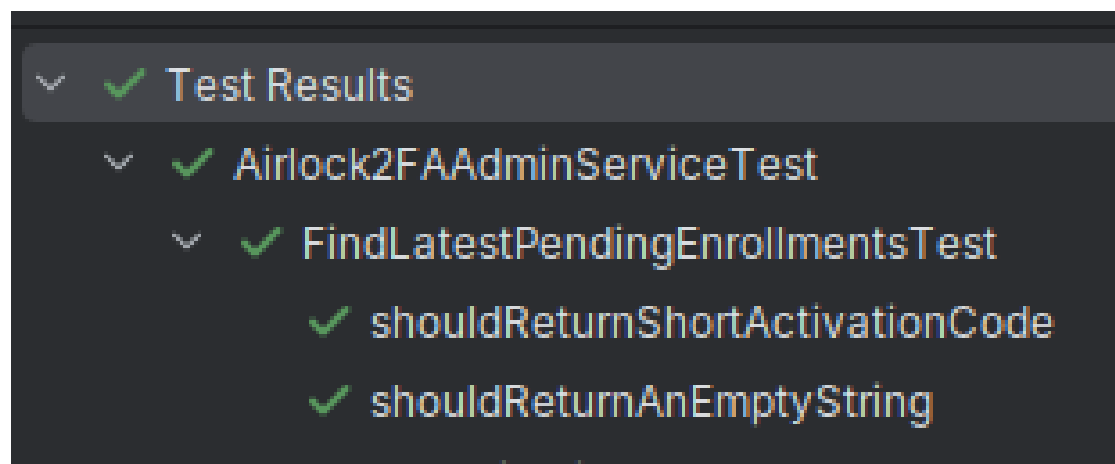


Abbildung 13.2: Unit-Test Resultate Airlock2FAAdminService.java

13.1.2 REST-Integration-Tests

Alle geschriebenen REST-Integration-Tests sollen erfolgreich durchlaufen.

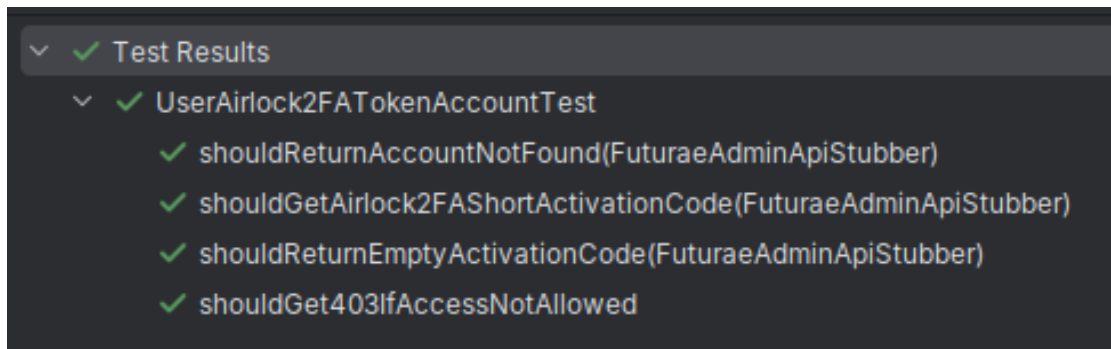


Abbildung 13.3: REST-Integration-Tests Resultate

13.1.3 UI-Integration-Tests

Alle geschriebenen UI-Integration-Tests sollen erfolgreich durchlaufen.

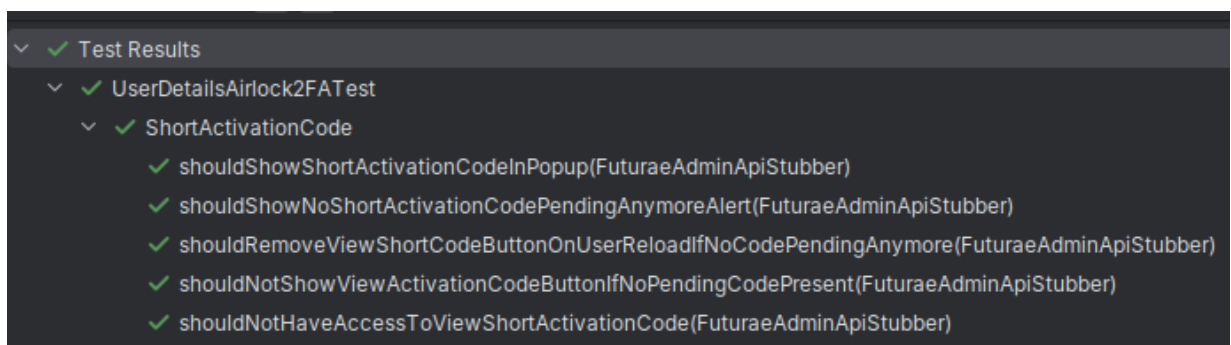


Abbildung 13.4: UI-Integration-Tests Resultate

13.1.4 Manuelle Tests

Sowie definiert in Abschnitt 10.4.6 werden die manuellen Tests durchgeführt. Der Teststatus wird mit folgenden Symbolen dargestellt:

✓: Test erfolgreich

✗: Test fehlgeschlagen

In der folgenden Tabelle sind die Resultate der Tests aufgelistet:

Testfall	Erwartetes Resultat	Tatsächliches Resultat	Status
M1	Es öffnet sich ein Popup (kein Browserpopup), mit dem Aktivierungscode.	Es öffnet sich ein Popup (kein Browserpopup), mit dem Aktivierungscode.	✓
M2	Der Button wird nicht angezeigt. Der Admin darf auch via einen alternativen REST-Client, wie z.B Postman, den Aktivierungscode nicht bekommen.	Der Button wird nicht angezeigt. Der Admin hat auch via einen alternativen REST-Client, wie z.B Postman, den Aktivierungscode nicht bekommen.	✓
M3	Der Button wird nicht angezeigt.	Der Button wird nicht angezeigt.	✓

Alle Tests konnten erfolgreich durchgeführt werden. Somit funktioniert das neue Feature wie erwartet, und es mussten keine Fehler behoben werden.

13.2 Qualitätssicherung

Die Qualität wurde zum Teil manuell, sowie automatisiert durchgeführt.

13.2.1 Manuelle Refactorings


Um Schreibfehler und andere unschönheiten im Code zu finden und zu verbessern wurde nach dem Implementieren, der ganze, neu geschriebene Code nochmals durchgegangen und kleine Findings wurden direkt behoben oder refactored.

13.2.2 Automatisierte Code-Analyse


Die Qualitätssicherung wurde auch während der Entwicklung fortlaufend sichergestellt. Jeder Commit, welcher auf Gerrit landet löst automatisch die Build Pipeline auf dem Jenkins aus. Diese führt neben den ganzen Code-Tests auch statische Code-Analysen durch. Diese sind:

- SpotBugs
- PMD
- CheckStyle


Resultate des letzten, finalen Commits:

**CheckStyle:** No warnings (from 231 analyses) ⓘ

- No issues for 87 builds, i.e. since build: ✓ #2476
- Reference build: ✓ [airlock-iam.verify-8.4 #2560](#)
- Quality gate: Success ✓

**PMD:** No warnings (from 101 analyses) ⓘ

- No issues for 246 builds, i.e. since build: #2317
- Reference build: ✓ [airlock-iam.verify-8.4 #2560](#)
- Quality gate: Success ✓

**SpotBugs:** No warnings (from 103 analyses) ⓘ

- No issues for 32 builds, i.e. since build: ⚠ #2531
- Reference build: ✓ [airlock-iam.verify-8.4 #2560](#)
- Quality gate: Success ✓

Abbildung 13.5: Resultate der Code-Analyse Tasks auf Jenkins

Wie im obigen Bild zusehen, sind alle Checks für diesen Build erfolgreich durchgelaufen. Dies bedeutet die Qualitätssicherung ist so gut wie möglich sichergestellt.

14 Auswerten

Abbildungsverzeichnis

1	Logo der Ergon Informatik AG	1
4.1	Arbeitsplatz	11
10.1	Restflow	41
10.2	UI Mockup	43
10.3	Popup mit Datum	45
10.4	Variante ohne Popup	46
12.1	REST-Dokumentation Request	56
12.2	REST-Dokumentation Response	57
12.3	Logeintrag bei vorhandenem Aktivierungscode	62
12.4	Page Alert	70
12.5	Popup deutsch	72
12.6	Popup englisch	73
12.7	Popup französisch	73
13.1	Unit-Test Resultate FuturaeAdminApiEnrollmentServiceImpl.java	78
13.2	Unit-Test Resultate Airlock2FAAdminService.java	78
13.3	REST-Integration-Tests Resultate	79
13.4	UI-Integration-Tests Resultate	79
13.5	Resultate Code-Analyse	81