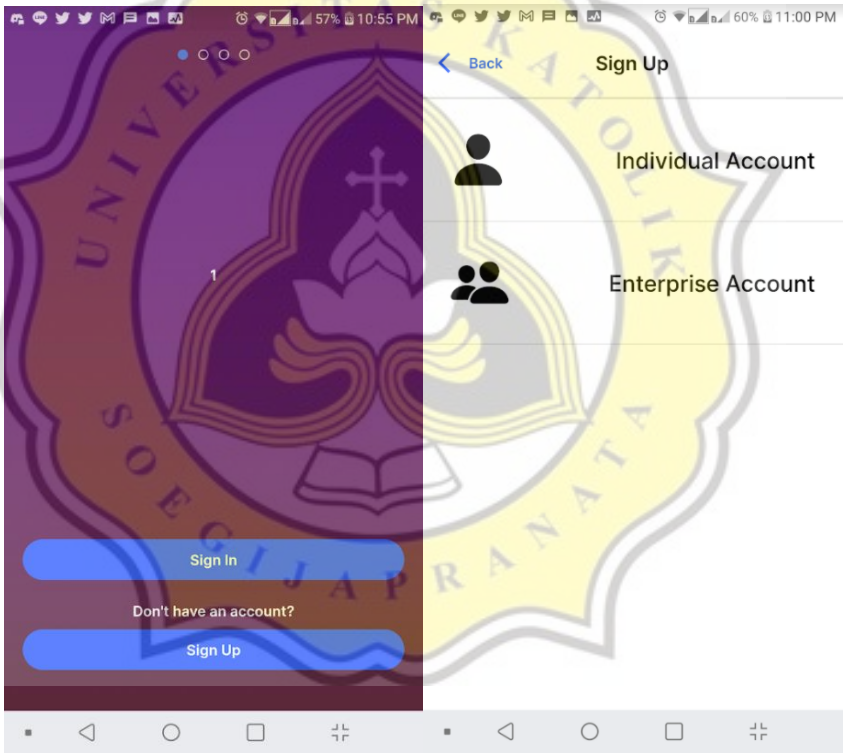
BAB IV
PERANCANGAN DAN PENGUJIAN APLIKASI

4.1 Alur aplikasi

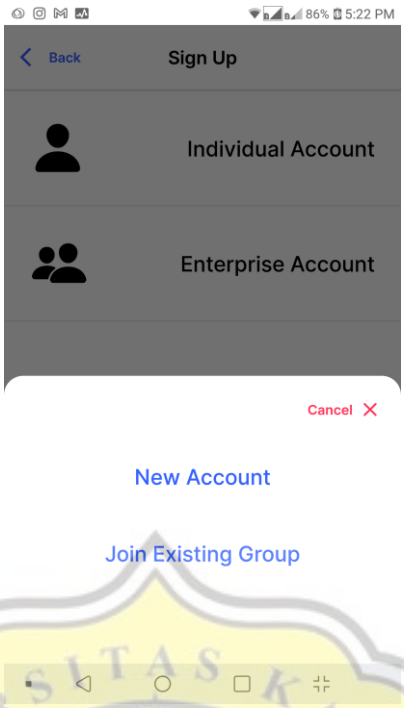
4.1.1 Pendaftaran (Signing up)

Proses pendaftaran dimulai dengan memilih tombol “Sign up” yang ada pada halaman awal aplikasi. Pengguna dapat memilih dua tipe akun pengguna yaitu akun individual (*personal*/perseorangan) atau akun *enterprise* (*group account*) seperti pada gambar 4.1. Apabila pengguna memilih untuk mendaftar group account, pengguna dapat memilih untuk bergabung dengan enterprise group yang sudah ada, atau mendaftar baru sebagai “enterprise group admin”.



Gambar 4.1 Halaman awal saat aplikasi dibuka, dan halaman pilihan tipe akun yang hendak didaftarkan.

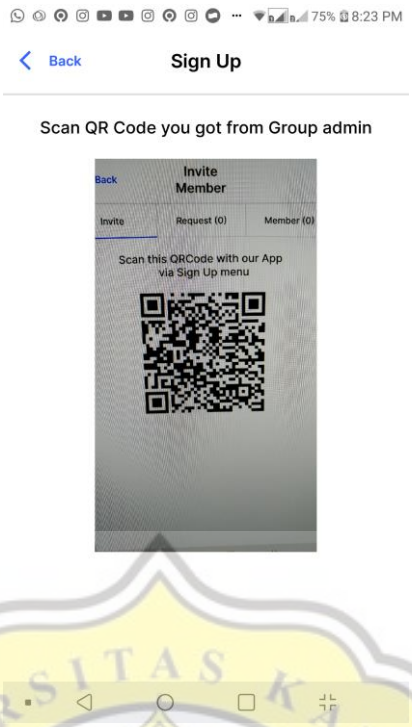
Apabila memilih untuk mendaftar sebagai “Enterprise Account” terdapat dua pilihan yaitu “New Account” apabila hendak mendaftar sebagai group admin (enterprise admin) atau “Join Existing Group” untuk mendaftar sebagai anggota dari group yang sudah ada seperti yang tertampil pada gambar 4.2.



Gambar 4.2 Modal berbentuk drawer untuk memilih new account atau join existing group.

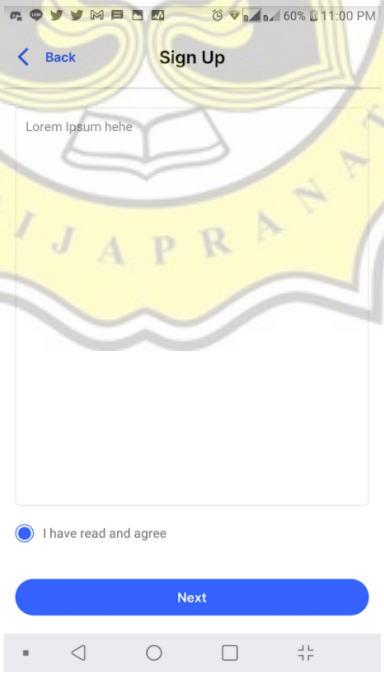
Apabila pengguna hendak mendaftar sebagai anggota dari sebuah "enterprise group", pengguna membutuhkan QR Code yang kemudian harus di *scan* untuk dapat melanjutkan pendaftaran selanjutnya seperti pada gambar 4.4. QR Code dapat dilihat melalui akun admin dari "enterprise group" pada menu "Invite member" di dalam tab "Invite" seperti pada gambar 4.3.

Gambar 4.3 QR Code untuk mendaftarkan anggota enterprise group.



Gambar 4.4 Calon group member melakukan scan QR Code dari group admin

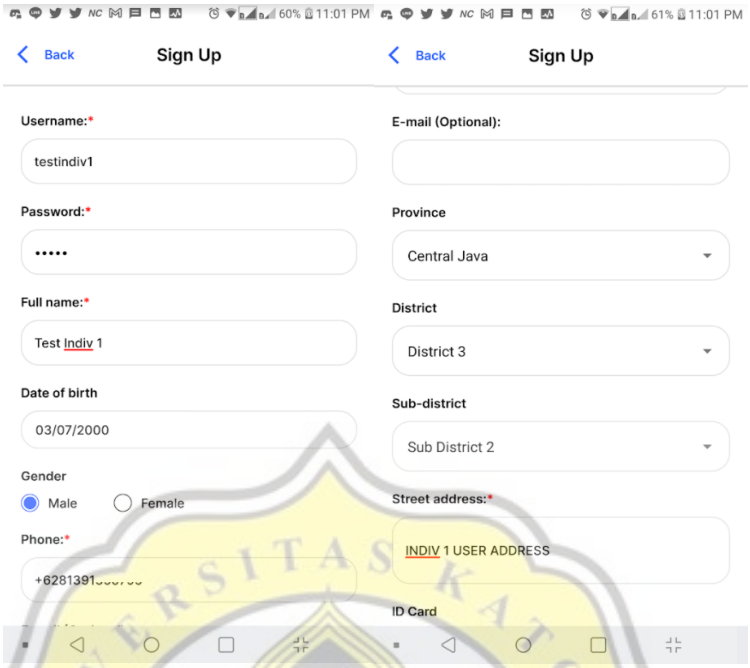
Kemudian, pengguna akan diminta untuk membaca persyaratan dan ketentuan layanan seperti pada gambar 4.5. Apabila pengguna menyetujui, proses pendaftaran dapat dilanjutkan ke pengisian data pengguna.



Gambar 4.5 Halaman persetujuan layanan.

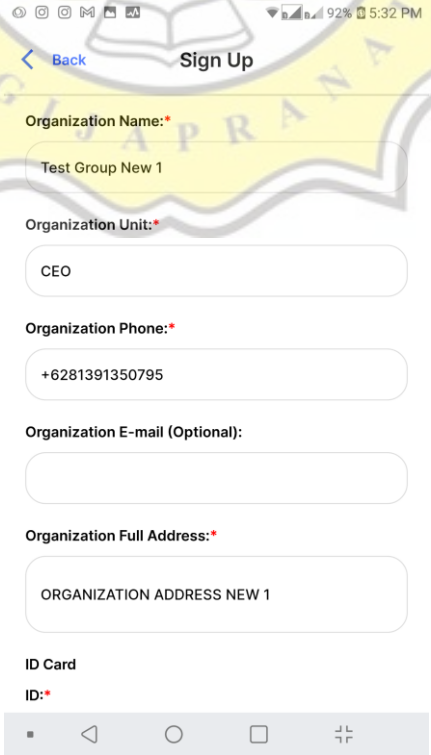
Pada halaman pengisian data pendaftaran, terdapat beberapa kolom yang harus diisikan bergantung pada pilihan tipe akun yang dipilih. Apabila pengguna memilih

akun individual seperti pada gambar 4.6, pengguna tidak usah mengisikan data yang berkaitan dengan organisasi.



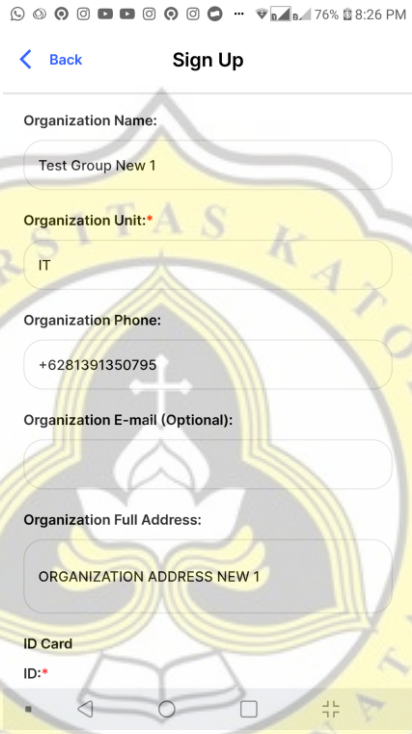
Gambar 4.6 Halaman pendaftaran akun tipe individual.

Apabila pengguna mendaftar sebagai “enterprise account admin” seperti pada gambar 4.7, terdapat field tambahan yang harus diisikan berkaitan dengan informasi organisasi yang nantinya akan digunakan juga untuk sistem pengelolaan aset keuangan anggota grupnya.



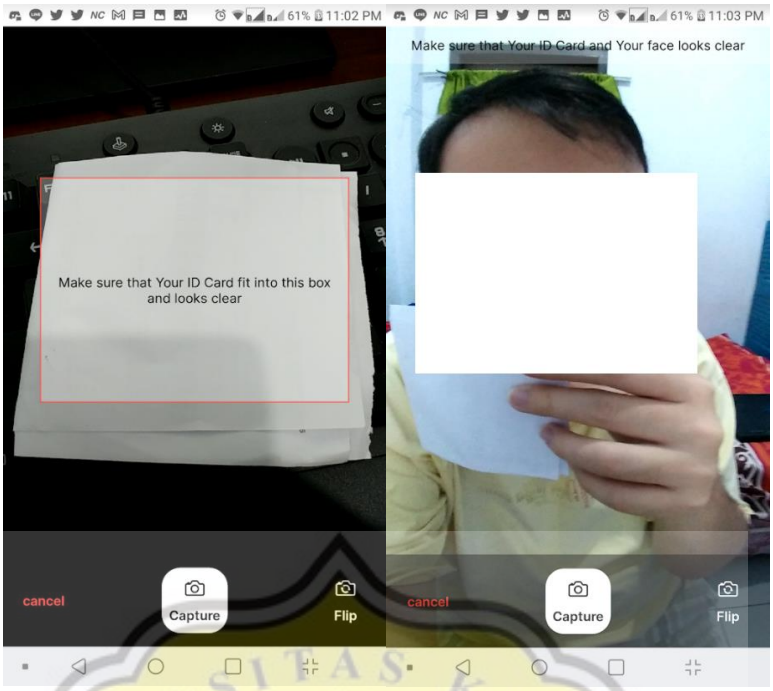
Gambar 4.7 Field tambahan pada pendaftaran enterprise admin account.

Apabila pengguna mendaftar sebagai anggota dari sebuah enterprise account yang telah terdaftar sebelumnya, pengguna cukup melakukan *scan* QR Code *invitation* dari enterprise admin terkait seperti pada gambar 4.4, kemudian dapat mengisi persyaratan yang ada, hanya saja terdapat informasi yang telah terisi sebelumnya dan tidak dapat diubah kembali seperti informasi yang berkaitan dengan organisasi, kecuali unit organisasi dari pengguna itu sendiri yang masih dapat diubah seperti dapat dilihat pada gambar 4.8.

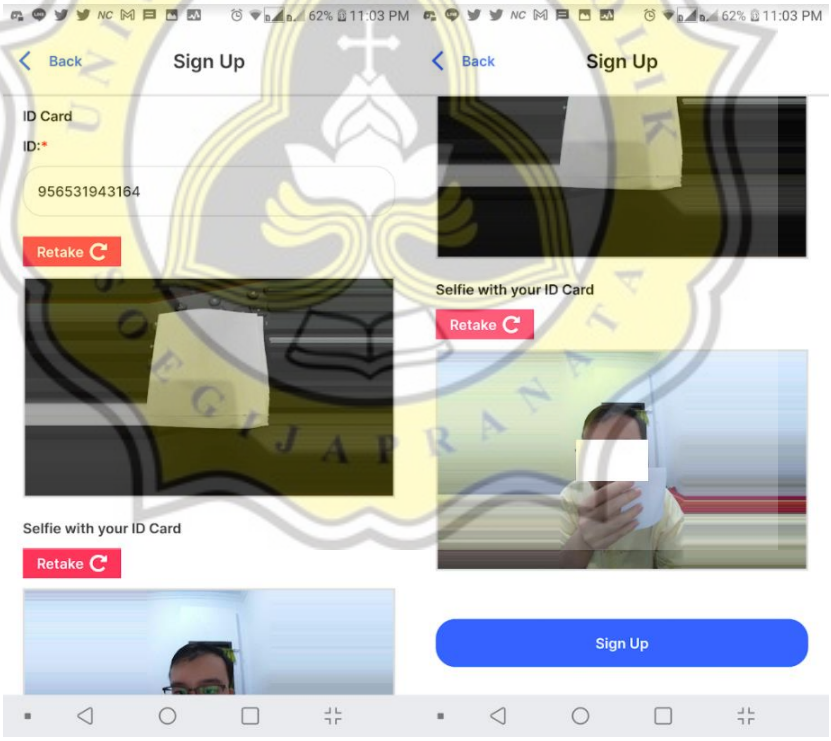


Gambar 4.8 Field yang dapat diisi oleh enterprise member.

Pada tiap pendaftaran diperlukan juga untuk menyertakan foto dari kartu identitas beserta juga foto calon pengguna yang memegang kartu identitasnya dengan jelas seperti pada gambar 4.9. Informasi ini akan disimpan dan akan digunakan dalam proses peninjauan calon pengguna oleh RA (request authority) admin.



Gambar 4.9 Pengambilan foto kartu identitas dan selfie bersama kartu identitas.



Gambar 4.10 Lanjutan halaman pendaftaran pengguna individual.

Apabila terdapat informasi yang bersifat *mandatory* belum terisi, maka aplikasi akan memberi peringatan bahwa terdapat kolom yang belum terisi. Semua kolom yang bersifat *mandatory* diberi tanda '*' berwarna merah setelah *label* dari kolom tersebut. Apabila semua data telah terisi, pengguna dapat menekan tombol "Sign Up" seperti

pada gambar 4.10. Kemudian aplikasi akan meminta pengguna untuk melakukan verifikasi OTP dengan mengirimkan SMS berisikan kode OTP ke nomor telepon pendaftar seperti pada gambar 4.11.

```
{
    otpModal &&
        <OtpModal
            isVisible={otpModal}
            phoneNumber={phone}
            onBack={() => setOtpModal(false)}
            onDone={onOtpDone}
        />
}
```

Kode 4.1 React Component "OtpModal" yang akan muncul apabila state "otpModal" bernilai true.

```
const OtpModal = ({
    isVisible,
    onDone,
    onBack,
    phoneNumber
}) => {

    useEffect(() => {
        sendOtp()
    }, [])

    const [otp, setOtp] = useState(0)
    const [isGetOtp, setIsGetOtp] = useState(false)
    const [otpInput, setOtpInput] = useState(undefined)
    const [timerCurrent, setTimerCurrent] = useState(0)
    const [timerTimeout, setTimerTimeout] = useState(10)

    const sendOtp = async () => {

        const body = { to: phoneNumber }

        return await request.post("sendotp", body)
            .then(async (response) => {

                const responseStatus = response.status
                const json = await response.json()
                const message = json.message
```

```
            const info = json.info

            if (responseStatus !== 200) return
SimpleAlert(message + "\n" + json.info)

            return setOtp(info)
         })
         .catch(throwError)
   }

   const checkOtp = (code) => {
      if (otp === parseInt(code)) return onDone()

      return SimpleAlert("Inputted code did not match.")
   }

   const timer = useTimer({ delay: 1000, callback: () =>
timerCallback(), startImmediately: true })

   const timerCallback = () => {

      if (timerCurrent === timerTimeout) {
         setTimerCurrent(0)
         return timer.stop()
      }

      if(timer.isRunning()) setTimerCurrent(timerCurrent + 1)
   }

   const resendOtp = () => {
      setTimerTimeout(timerTimeout + 5)
      timer.start()
      sendOtp()
   }

   return (
      <Modals
         isVisible={isVisible}
         onBackButtonPress={onBack}
      >
         <HeaderNoNavigation
            onBack={onBack}
```

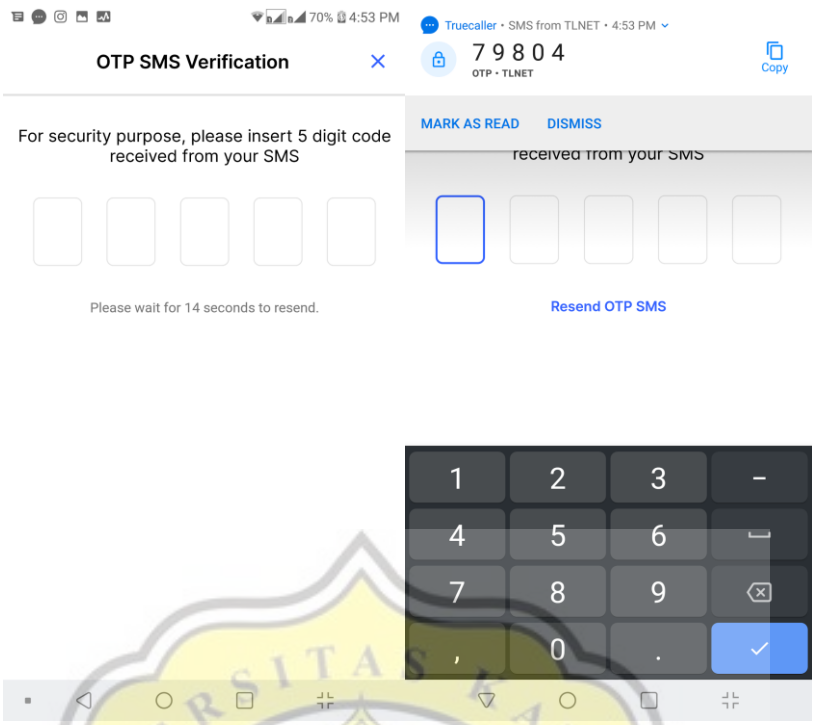
```
            label="OTP SMS Verification"
         />
         <ScrollView style={styles.scrollView}>
            <Spacing v={14} />
            <Text style={styles.text}>For security purpose,
please insert 5 digit code received from your SMS</Text>
            <Spacing v={14} />
            <OTPInputView
               style={styles.otpContainer}
               pinCount={5}
               code={otpInput}
               onCodeChanged={setOtpInput}
               autoFocusOnLoad
               codeInputFieldStyle={styles.otpField}

codeInputHighlightStyle={styles.otpFieldHighlight}
               onCodeFilled={code => checkOtp(code)}
            />
            <Spacing v={15} />
            {
               timer.isRunning() &&
               <Text style={styles.resendWaitText}>Please wait
for {timerTimeout - timerCurrent} seconds to resend.</Text>
            }
            {
              timer.isStopped() &&
               <Button
                  onPress={resendOtp}
                  label="Resend OTP SMS"
                  labelColor={palette.blue1}
                  buttonColor={palette.transparent}
                  styleButton={styles.resendContainer}
               />
            }
         </ScrollView>
      </Modals>
   )
}
```

Kode 4.2 React Component "OtpModal".

```
const sendOtp = router.post('/sendotp', async (req, res, next)
=> {
```

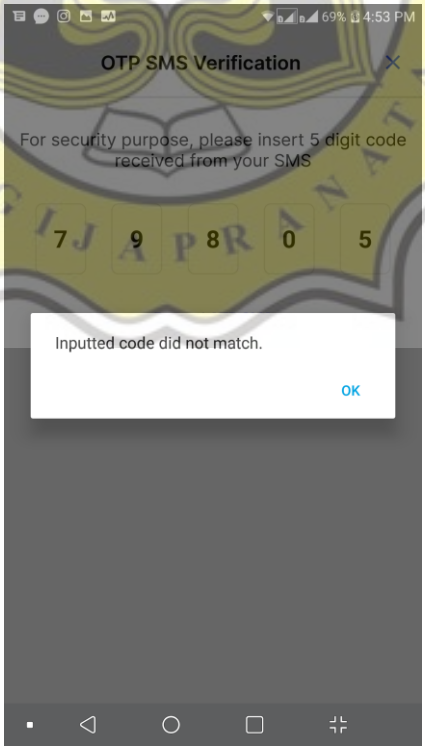
```
    const client = twilio(TWILIO_ACCOUNT_SID, TWILIO_AUTH_TOKEN)

    const to = req.body.to
    const vericode = genOtp()

    if (!to) return next(badRequest())

    return client.messages
        .create({
            body: '<#> Test OTP. Your verification code is: ' +
vericode + '.\n' + APP_HASH,
            from: TWILIO_NUMBER,
            to
        })
        .then(response => {
            return res.status(200).json({
                message: "OTP code request send succesfuly.",
                info: vericode
            })
        })
        .catch(next)
})
```

Kode 4.3 Kode pada endpoint "./sendotp" pada backend mengirim OTP menggunakan vendor Twilio.



Gambar 4.11 Menerima Kode OTP melalui SMS.

Apabila kode OTP yang dimasukan tidak sesuai, akan muncul peringatan apabila seluruh *field* OTP telah terisi seperti pada gambar 4.12 dan proses pengiriman data pendaftaran tidak akan dilanjutkan.



Gambar 4.12 Kode OTP yang dimasukan tidak sesuai.

Apabila kode OTP yang dimasukan sesuai, maka aplikasi akan mengirimkan informasi pendaftar ke *endpoint* "/signup" melalui POST *request* dalam format *raw* JSON seperti pada halnya pada semua *request* ke semua *endpoint* juga akan dikirimkan dengan cara yang sama.

Untuk melakukan pendaftaran, diperlukan CSR (*Certificate Signing Request*) yang akan digunakan untuk melakukan *request* pembuatan sertifikat *digital* apabila *request* pembuatan sertifikat *digital* belum pernah disetujui, atau melakukan *request* sertifikat *digital* yang telah dibuat apabila pengguna telah disetujui sebelumnya. Pembuatan CSR dapat dilakukan dengan memanfaatkan library "node-forge". Dapat diisikan juga detail seperti nama pemilik sertifikat, organisasi, negara, *e-mail*, dan sebagainya. *Public key* digunakan sebagai identitas penandatangan, dan untuk menandatangani CSR menggunakan *private key* dari pengguna. Data dari CSR yang telah dibuat akan kemudian disimpan ke dalam *database* pada *collection* "users". Proses pembuatan CSR dapat dilihat pada kode 4.4 dan kode 4.5.

```
const pkcs10 = csrPkcs10({
    commonName,
    countryNameShort,
    stateOrProvince,
    locality,
    organization,
    organizationUnit,
    email,
    phone,
    publicKey,
    privateKey
})

    const pkcs10pem = pkcs10.pem
    const pkcs10hex = hexEncode(pkcs10pem)
    const pkcs10created = dateNowIso()

    const pkcs10Array = {
        csr: pkcs10hex,
        created: pkcs10created,
    }

    const requestCertificateBody = {
        pkcs10: pkcs10pem,
```

```
        profile: "pc-client"
    }
```

Kode 4.4 Pembuatan CSR untuk dikirimkan ke OpenXPKI dan data CSR yang akan disimpan.

```
    let pki = forge.pki

    const publicKeyPem = Buffer.from(publicKey,
"hex").toString('utf8')
    const privateKeyPem = Buffer.from(privateKey,
"hex").toString('utf8')

    const publicKeyForge = pki.publicKeyFromPem(publicKeyPem)
    const privateKeyForge = pki.privateKeyFromPem(privateKeyPem)

    let csr = forge.pki.createCertificationRequest()

    csr.publicKey = publicKeyForge
    csr.setSubject([{
        shortName: 'CN',
        value: commonName
    }, {
        shortName: 'C',
        value: countryNameShort
    }, {
        shortName: 'ST',
        value: stateOrProvince
    }, {
        shortName: 'L',
        value: locality
    }, {
        shortName: 'O',
        value: organization
    }, {
        shortName: 'OU',
        value: organizationUnit
    }])

    csr.sign(privateKeyForge)

    const pem = forge.pki.certificationRequestToPem(csr)
    return { pem }
```

Kode 4.5 Detail fungsi "csrPkcs10".

Lalu pada aplikasi backend untuk melakukan POST *request* menggunakan fungsi yang dibuat seperti pada kode 4.6.

```
import fetch from 'node-fetch'
import https from 'https'
import { BASE_ADDRESS, OPENXPKI_ADDRESS } from '../../config/'

const header = () => {
    return {
        'Accept': 'application/json',
        'Content-Type': 'application/json',
    }
}

const httpsAgent = new https.Agent({ rejectUnauthorized: false
})

const post = async (base, endpoint, bodyObject) => {

    const url = !base ? BASE_ADDRESS : base === "openxpki" ?
OPENXPKI_ADDRESS : BASE_ADDRESS

    const body = JSON.stringify(bodyObject)
    const headers = header()

    if (url.includes("https"))
        return await fetch(url + endpoint, {
            method: 'POST',
            headers,
            body,
            agent: httpsAgent
        })
            .then((response) => { return response })
            .catch((error) => { return error })

    if (url.includes("http"))
        return await fetch(url + endpoint, {
            method: 'POST',
            headers,
            body
        })
```

```
            .then((response) => { return response })
            .catch((error) => { return error })
}

export const request = { post }
```

Kode 4.6 Fungsi yang digunakan untuk melakukan berbagai POST request menggunakan package "node-fetch".

Berikut pada kode 4.7 adalah fungsi "requestCertificate" yang dibuat untuk melakukan *request* ke OpenXPKI untuk memberikan *certificate* dari pengguna.

```
import { request } from "../../request";
import { badRequest, fatalError, timeout } from
"../../../api/middleware/badResponse"

const requestCertificate = async (csrPkcs10_pem) => {

   if(!csrPkcs10_pem) return badRequest("No CSR PEM.")

   const throwError = (error) => { return fatalError(error) }

   const requestCertificateBody = {
       pkcs10: csrPkcs10_pem,
       profile: "pc-client"
   }

   const reqCert = await request.post("openxpki",
"rpc/enroll/RequestCertificate",
requestCertificateBody).catch(throwError)

   const reqCertOk = reqCert.ok
   const reqCertCode = reqCert.status

   if(!reqCertOk && !reqCertCode) return timeout()

   if (reqCertCode > 299) {
       if (reqCertCode === 408 || reqCertCode === 500) {
           return {
               status: reqCertCode,
               message: "Something went wrong. Please try again
later.",
               info: {}
```

```
            }
        }
        return {
            status: reqCertCode,
            message: reqCertJson.error.message,
            info: {}
        }
    }

    if (reqCertCode === 202) {
        return {
            status: reqCertCode,
            message: "User needs manual authorization by RA
admin.",
            info: {}
        }
    }

    const reqCertJson = await reqCert.json().catch(throwError)
    const reqCertData = reqCertJson.result.data

    const chain = reqCertData.chain
    const certPem = reqCertData.certificate
    const chainchain = [certPem, chain]

    return {
        status: 200,
        message: "Succesfully getting users certificate.",
        info: chainchain
    }
}

export default requestCertificate
```

Kode 4.7 Fungsi yang digunakan untuk melakukan request certificate ke OpenXPKI

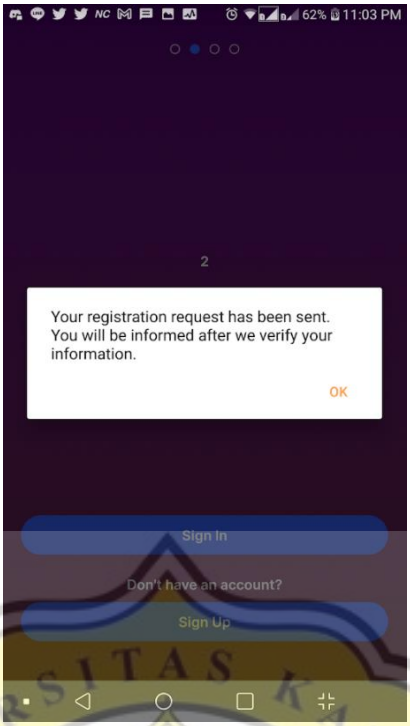
Lalu CSR yang telah dibuat akan dikirimkan ke RPC API OpenXPKI dengan *endpoint* "./RequestCertificate" dengan menyertakan body berupa format PEM dari CSR yang telah dibuat seperti pada gambar 4.8, serta *certificate profile* yang hendak dibuat yang dicantumkan pada fungsi "requestCertificate" pada kode 4.7. Apabila *endpoint* memberikan respon "*non-error*" maka data pendaftar akan disimpan ke dalam

database seperti pada kode 4.8, dan data tersebut akan digunakan untuk melewati proses validasi pendaftaran lebih lanjut.

```
const reqCertResult = await
requestCertificate(pkcs10pem).catch(next)
const reqCertCode = reqCertResult.status

if(reqCertCode !== 202 && reqCertCode !== 200) return
res.status(reqCertCode).json(reqCertResult)

return await UserModel.create({ ...data })
    .then(() => {
        return res.status(200).json({
            message: "Registered successfully. Waiting for
review.",
            info: {}
        })
    })
    .catch(error => {
        if (error.code === 11000) return next(conflicted("Username
already exists"))
        if (error._message === "UserSchema validation failed")
return next(badRequest("Please fill all of the required
fields"))
        return next(error)
    })
```

Kode 4.8 Melakukan request certificate dan menyimpan ke dalam database.

Apabila pendaftaran telah berhasil, pada aplikasi *front-end* akan langsung diarahkan ke halaman awal aplikasi dengan juga memunculkan *pop-up* berisikan pesan bahwa proses registrasi telah terkirim dan akan melewati proses verifikasi seperti pada gambar 4.13. Proses verifikasi akan dilakukan oleh RA (request authority) admin.



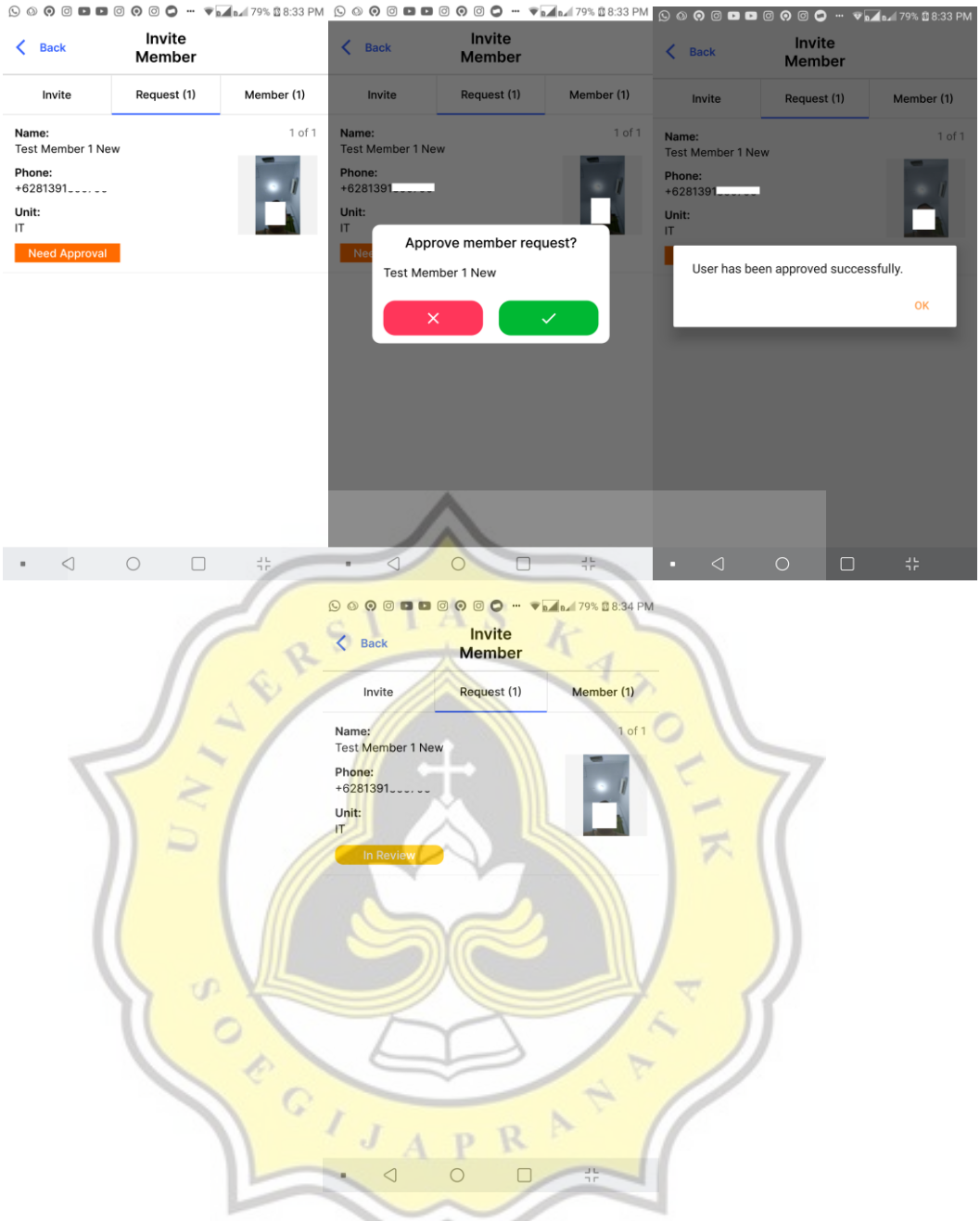
Gambar 4.13 Tampilan apabila pendaftaran berhasil.

Dan pada OpenXPKI, apabila CSR yang dikirimkan belum disetujui oleh RA admin, maka apabila mengirimkan CSR tersebut ke *endpoint* "./RequestCertificate" akan memberikan hasil seperti pada gambar 4.14. Hasil yang diberikan akan berisikan state "MANUAL_AUTHORIZATION", dan pada respon JSON yang diberikan tidak terdapat *certificate* milik pengguna.

```
{
  result: {
    state: 'MANUAL_AUTHORIZATION',
    data: { transaction_id: '7896cf38847e7fe3b08252c1fbddc5dec72f291b' },
    retry_after: 300,
    pid: 3863,
    id: 13311,
    proc_state: 'manual'
  }
}
```

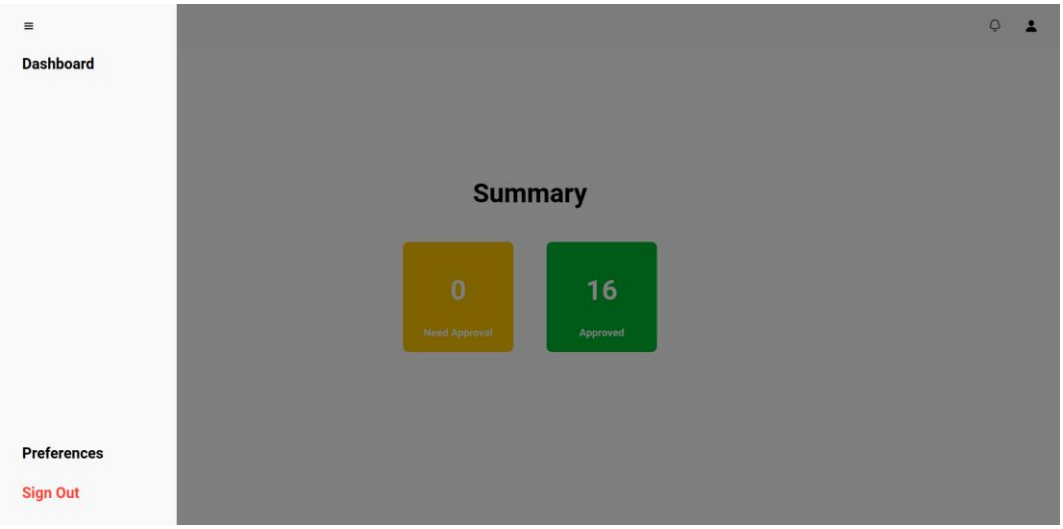
Gambar 4.14 Respon dari OpenXPKI apabila request belum di-approve RA Admin.

Pada pendaftaran *enterprise member*, admin *enterprise* (*group admin*) harus melakukan proses penerimaan terlebih dahulu pada menu "Invite Member" pada tab "Request" seperti pada gambar 4.15 baru kemudian *request* pendaftaran *enterprise member* dapat dilanjutkan ke proses selanjutnya yaitu untuk disetujui oleh RA admin apabila request telah disetujui oleh *enterprise admin*.



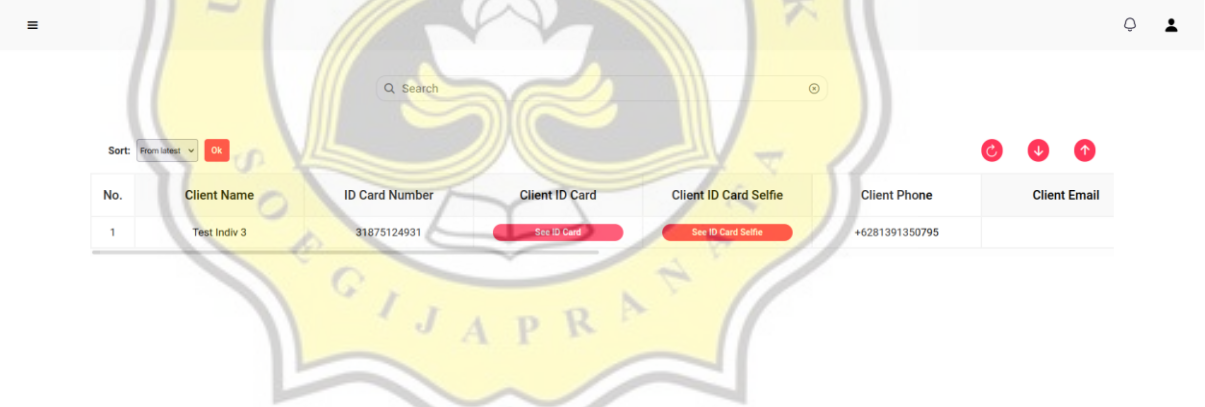
Gambar 4.15 Group admin menyetujui request dan menunggu persetujuan RA operator.

Kemudian *request* tersebut akan diterima oleh RA admin (RA operator) yang memang khusus menangani akan pendaftaran pengguna aplikasi. Operator dapat mengecek *request* yang masuk melalui *web dashboard* khusus yang dibuat untuk menampilkan *request* pendaftaran pengguna yang masuk seperti pada gambar 4.16.



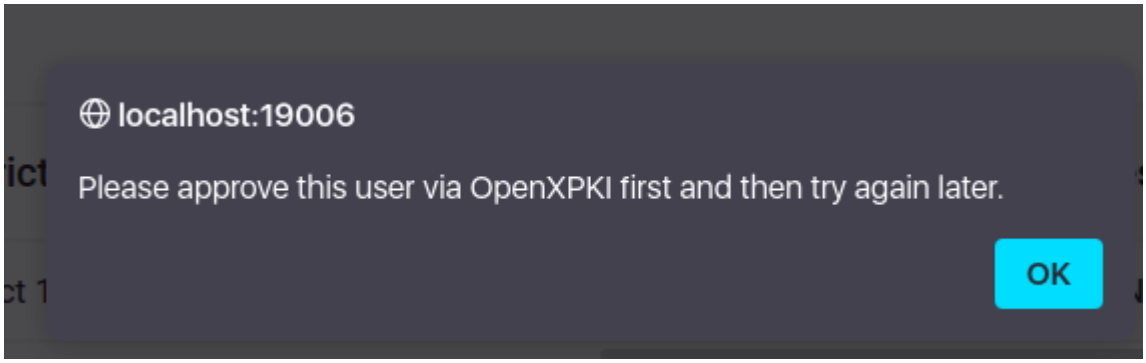
*Gambar 4.16 Web dashboard **Request Authority** yang dibuat menggunakan **Expo.***

Di sini operator harus terlebih dahulu mengecek masing-masing *request* yang masuk. Dengan beberapa informasi yang ditampilkan seperti KTP, dan informasi lainnya seperti pada gambar 4.17, diharapkan dapat memberikan informasi akan kelayakan pengguna sebelum kemudian diterima untuk menghindari penyalahgunaan atau tindakan yang tidak diinginkan lainnya.

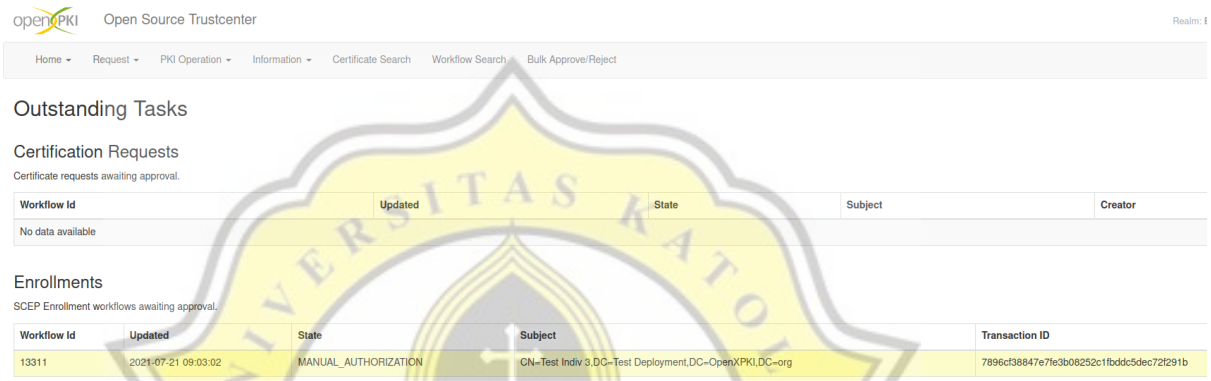


Gambar 4.17 Menu "Need Approval" pada dashboard apabila menerima request pendaftaran.

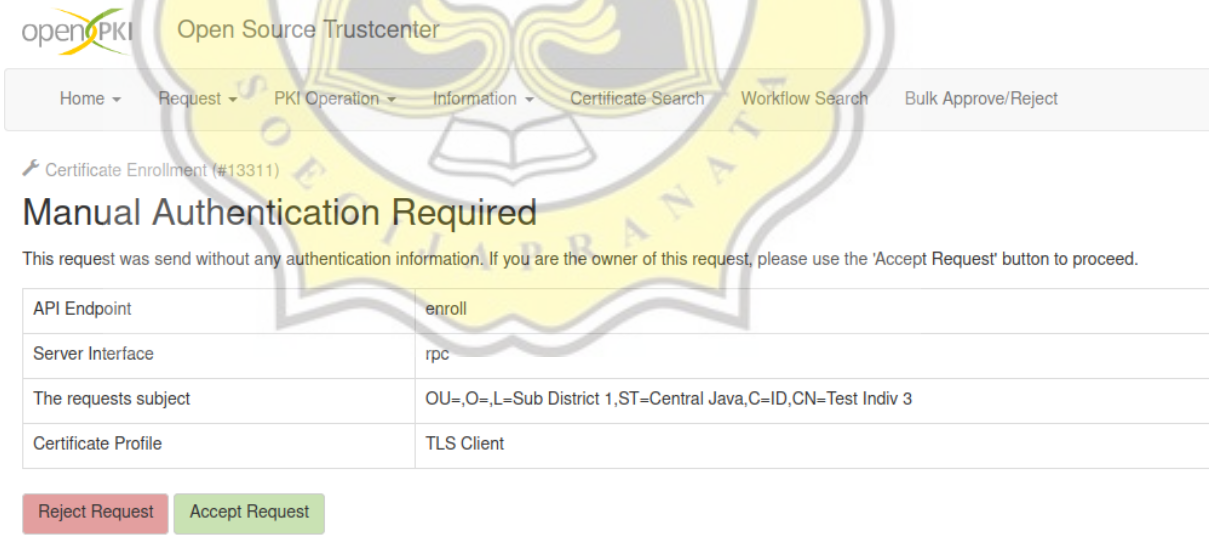
Kemudian, operator harus melakukan tindakan penerimaan *request* melalui *website* OpenxPKI seperti pada gambar 4.19 dan gambar 4.20 sebelum dapat melakukan *approval* melalui *web dashboard request* pendaftaran. Apabila RA operator belum melakukan penerimaan melalui web OpenXPKI, maka akan mucnul pop-up berisikan peringatan seperti pada gambar 4.18. Apabila operator telah menerima *request* melalui OpenxPKI, OpenxPKI akan melakukan proses pembuatan *digital certificate* milik pengguna yang bersangkutan.



Gambar 4.18 Apabila request yang masuk ke OpenXPKI belum disetujui.



Gambar 4.19 Tampilan web OpenXPKI ketika terdapat request.



Gambar 4.20 Tampilan detail request dan juga pilihan untuk menerima atau menolak request.

Apabila proses penerimaan telah sukses, akan muncul pesan seperti pada gambar 4.21.

This workflow has finished with success and can not be restarted

Certificate Enrollment (#13311)

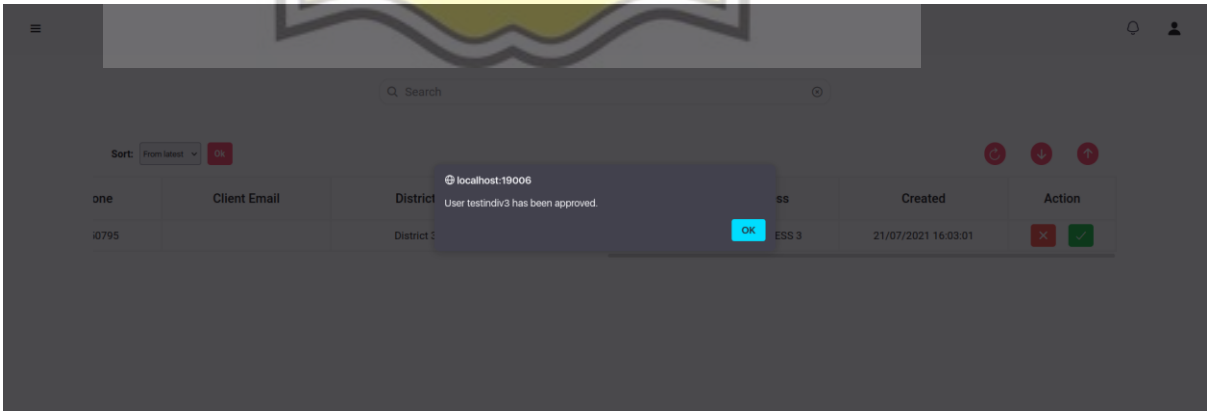
SUCCESS

The certificate was issued and will be send to the client on request.

Certificate	Kb15C_7RtmQ5ZnFhSuVPxrs5i3Y CN=Test Indiv 3,DC=Test Deployment,DC=OpenXPKI,DC=org
API Endpoint	enroll
Server Interface	rpc
Certificate Subject	CN=Test Indiv 3,DC=Test Deployment,DC=OpenXPKI,DC=org
Certificate Profile	TLS Client
Request Mode	initial
Transaction ID	7896cf38847e7fe3b08252c1fbddc5dec72f291b

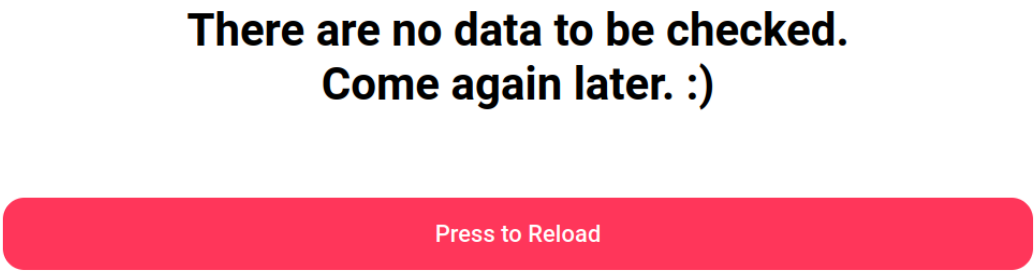
Gambar 4.21 Tampilan apabila request enrollment disetujui.

Kemudian, operator dapat melakukan proses approval terhadap pengguna yang bersangkutan karena *digital certificate* telah diberikan oleh OpenxPKI. Setelah pendaftar di-approve, maka akan dilakukan pembuatan akun ke sistem Hyperledger Iroha (beserta pembuatan *domain* dan juga *asset* apabila pendaftar adalah pendaftar *enterprise account*). Dan apabila CSR yang sebelumnya di-approve oleh OpenXPKI dikirimkan ke *endpoint* "/rpc/enroll/RequestCertificate", maka OpenXPKI akan mengembalikan *certificate chain* yang dapat digunakan disaat dibutuhkan seperti saat hendak melakukan sertifikasi dokumen.



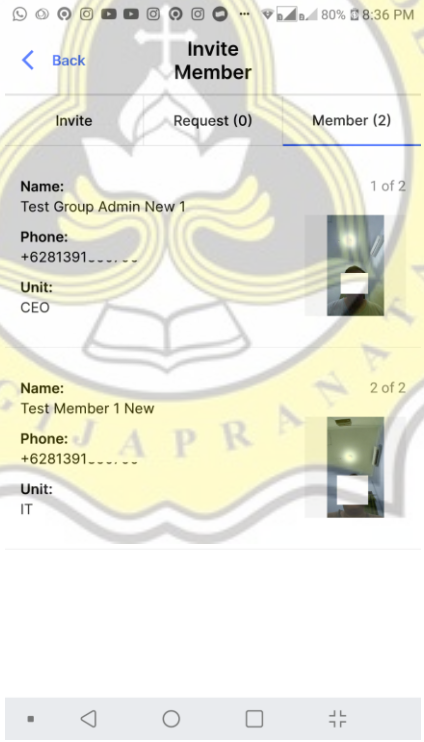
Gambar 4.22 Tampilan apabila melakukan persetujuan melalui dashboard admin operator RA.

Apabila sudah tidak ada *request* yang harus diproses, tampilan *website dashboard* pada menu “Need Approval” akan memberikan hasil seperti pada gambar 4.23.



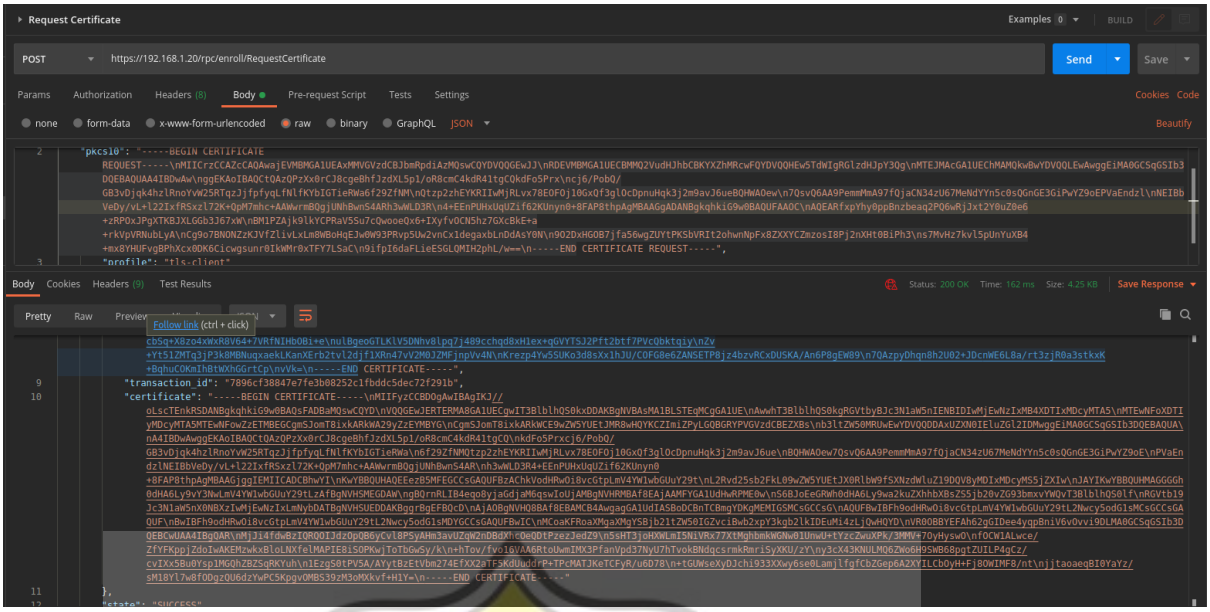
Gambar 4.23 Tidak ada request yang masuk atau semua request telah ditangani oleh RA operator.

Dan pada halaman “Invite Member” apabila request pendaftaran member *enterprise* telah diterima, maka member akan tercatat pada tab “Member” pada akun *enterprise admin* yang bersangkutan seperti pada gambar 4.24.



Gambar 4.24 Tampilan pada daftar member group apabila member telah disetujui oleh RA operator.

Berikut adalah gambar apabila CSR yang telah diterima oleh RA admin dikirimkan melalui *endpoint* “./RequestCertificate”. Dapat dilihat pada gambar 4.25 bahwa sekarang respon JSON yang diberikan berisikan informasi *certificate* dari pengguna yang kemudian dapat digunakan seperti saat melakukan sertifikasi dokumen.

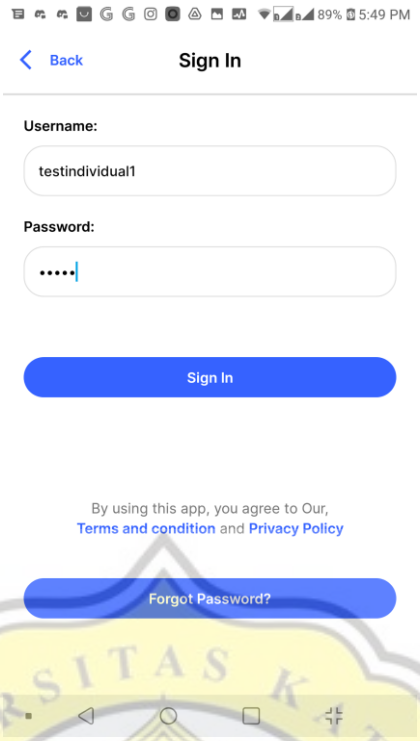


Gambar 4.25 Certificate chain yang diberikan oleh OpenXPKI.

Setelah operator menyetujui *request* pendaftaran, pengguna dapat melakukan proses “Sign in” atau masuk ke aplikasi dan kemudian menggunakan layanan.

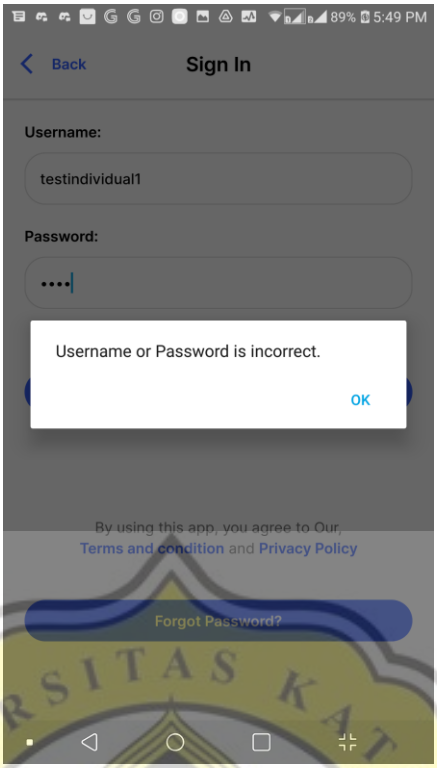
4.1.2 Masuk ke aplikasi (Signing in)

Untuk dapat masuk ke aplikasi, pengguna cukup menekan tombol “Sign in” pada halaman awal aplikasi. Kemudian pengguna akan diminta untuk memasukan informasi akun berupa *username* dan *password* seperti pada gambar 4.26. Apabila seluruh form telah diisi dengan benar, pengguna dapat menekan tombol “Sign In”, kemudian aplikasi *front end* akan mengirimkan data *username* serta *password* yang telah diinputkan pengguna tersebut ke *endpoint* “./signin”.



Gambar 4.26 Tampilan halaman "Sign In"

Pada *endpoint* tersebut, aktivitas intinya adalah mengecek apabila seluruh data telah dimasukan. Apabila sudah, maka akan terjadi pengecekan ke dalam *database* apabila data pengguna yang hendak masuk tersebut ada di dalam sistem dengan bantuan *username* yang diberikan. Kemudian tidak lupa membatasi juga data apa saja yang dibutuhkan untuk membuat proses agar lebih enteng karena data pada *collection* "users" juga memuat banyak data lain seperti gambar dan certificate yang tidak digunakan pada *endpoint* ini. Apabila data tidak ditemukan, maka fungsi akan segera memberikan return berupa *API response* dengan *error code* berstatuskan 401 dengan pesan "*Wrong credentials*" dan tampilan pada aplikasi *front-end* akan seperti pada gambar 4.27.



Gambar 4.27 Tampilan apabila username atau password salah.

Apabila terdapat data, tetapi *field* status akun dari pengguna yang terdapat pada *database* tidak bernilaikan "*verified*", maka sistem akan memberikan respon bahwa akun tersebut sedang dalam review. Apabila statusnya "*verified*" maka hasil yang ditemukan pada *database* tersebut akan digunakan dalam proses autentikasi *password*. Pada aplikasi *back-end* aktivitas yang dilakukan saat proses *sign-in* dapat dilihat pada kode 4.9.

```
const username = req.body.username
const password = req.body.password

if (!username || !password) return next(badRequest())

await Db().catch(next)

const userData = await UserModel.findOne(
    { username },
    {
        password: true,
        contact: true,
        accountStatus: true
    }
```

```
    ).exec().catch(next)

    if (!userData) return next(unauthorized())

    const accountStatus = userData.accountStatus

    if (accountStatus !== "verified") return
next(unauthorized("Your account is still being reviewed.\nPlease
wait until you contacted by admin."))

    const id = userData._id
    const idObj = mongoose.Types.ObjectId(id)
    const passFromDb = userData.password
    const phone = userData.contact.phone
    const authResult = await bcrypt.compare(password, passFromDb)

    if (!authResult) return next(unauthorized())

    const token = generateToken(id)
    const refreshToken = generateRefreshToken(id)

    await pushToken(idObj, refreshToken).catch(next)

    return res.status(200).json({
        message: "Signed in successfully",
        info: {
            phone,
            token,
            refreshToken
        }
    })
```

Kode 4.9 Source code pada endpoint "./signin"

Pada proses autentikasi *password*, karena *password* sebelumnya disimpan dalam *database* menggunakan *library* bcrypt, maka untuk melakukan autentikasi terhadap *password* juga harus menggunakan *library* "bcrypt" dengan membandingkan *password* yang dikirimkan pengguna, dengan *password* yang telah dienkripsi dan disimpan ke dalam *database* seperti dapat dilihat pada kode 4.9. Apabila proses autentikasi tidak berhasil, maka fungsi akan memberikan *return* respon dengan *error code* 401 dengan pesan yang memberi tahu bahwa informasi yang diberikan salah.

Apabila proses berhasil, maka akan dilakukan pembuatan *token* serta *refresh token* JWT, kemudian *refresh token* yang telah dibuat akan disimpan ke dalam *array refresh token* yang ada pada data pengguna bersangkutan di dalam *collection* "users" seperti pada kode 4.10. Baru kemudian sistem akan memberikan respon dengan status 200 berisikan pesan apabila proses "*sign in*" telah berhasil beserta membawa informasi lain juga seperti nomor telepon, *token*, dan *refresh tokennya* seperti pada kode 4.9.

```
const generateToken = (id) => {

    const options = { expiresIn: '10s' }

    const generatedToken = jwt.sign({ id }, JWT_SECRET, options)

    return generatedToken
}

const generateTokenWRefreshToken = async (id, refreshToken) => {

    await Db()

    const dbResult = await UserModel.findOne({ _id: id }, {
refreshToken: true })

    if (!dbResult) return "404"

    const refreshTokenStore = dbResult.refreshToken

    if (!refreshTokenStore.includes(refreshToken)) return "401"

    const newToken = generateToken()

    return newToken
}

const generateRefreshToken = (id) => {

    const options = { expiresIn: '10d' }

    const generatedRefreshToken = jwt.sign({ id }, JWT_SECRET_2,
options)
```

```
    return generatedRefreshToken
}

const pushToken = async (id, refreshToken) => {

    await Db()

    return await UserModel.updateOne(
        { "_id": id },
        { $push: { refreshToken } }
    )
}

const removeToken = async (id, refreshToken) => {

    await Db()

    return await UserModel.updateOne(
        { "_id": id },
        { $pull: { refreshToken } }
    )
}
```

Kode 4.10 Kumpulan fungsi dalam proses pembuatan, penginputan, dan penghapusan token juga refresh token.

Lalu pada aplikasi *front end* walaupun pada API memberikan respon dengan status sukses (200), proses "*sign in*" masih harus membutuhkan verifikasi lagi melalui verifikasi SMS OTP seperti dapat dilihat pada kode 4.11.

```
const signInAsync = async () => {

    if (!username || !password) return SimpleAlert("Username and
Password field should not empty")

    dispatch({ type: "IS_LOADING" })

    const body = { username, password }

    return await request.post('signin', body)
        .then(async (response) => {

            dispatch({ type: "NOT_LOADING" })
```

```
            const responseStatus = response.status
            const json = await response.json()
            const message = json.message
            const info = json.info

            if (responseStatus === 401) SimpleAlert("Username or
Password is incorrect.")
            if (responseStatus === 403) SimpleAlert(message)
            if (responseStatus === 200) {

                const token = info.token
                const refreshToken = info.refreshToken
                const phone = info.phone

                setTempToken({ token, refreshToken })
                setPhone(phone)
                return setOtpModal(true)
            }
        })
        .catch(throwError)
        .finally(() => dispatch({ type: "NOT_LOADING" }))
}
```

Kode 4.11 Fungsi "signInAsync" pada aplikasi frontend.

Apabila pengguna sukses melewati verifikasi OTP, informasi yang sebelumnya diterima seperti token dan refresh token akan disimpan ke dalam perangkat dan kemudian pengguna akan diarahkan ke halaman utama aplikasi. Prosesnya dapat dilihat pada kode 4.12 dan 4.13.

```
const verifyOtp = () => {
    setVerimodal(false)
    setOtpModal(true)
}

const onOtpDone = async () => {

    setOtpModal(false)

    await storeToken()
    await setItem("currentUser", username)
```

```
    dispatch({ type: "SET_CURRENT_USER", payload: username })

    return navigation.dispatch(
        CommonActions.reset({
        index: 1,
        routes: [ { name: "HomeAuthScreen" } ],
    }))
}
```

Kode 4.12 Fungsi "verifyOtp" dan "onOtpDone".

```
const storeToken = async () => {

    await setItem("token", tempToken.token)
    await setItem("refreshToken", tempToken.refreshToken)

    dispatch({ type: "SET_TOKEN", payload: tempToken.token })
    dispatch({ type: "SET_REFRESH_TOKEN", payload:
tempToken.refreshToken })
}
```

Kode 4.13 Fungsi "storeToken" yang akan dipanggil apabila proses verifikasi OTP berhasil.

Dan pada saat aplikasi dijalankan, apabila terdapat *token* dan *refresh token* tersimpan dalam Expo *secure storage* (pengecekan melalui fungsi getItem), maka pengguna akan diarahkan ke halaman utama aplikasi. Prosesnya dapat dilihat pada kode 4.14.

```
const Navigation = () => {
    const [state, dispatch] = useContext(Context)
    useEffect(() => {
        checkIfTokenAvailable()
    }, [])
    const checkIfTokenAvailable = async () => {
        let token = state.user.token
        let refreshToken = state.user.refreshToken
        if (!token)
            token = await getItem("token")
        if (!refreshToken)
            refreshToken = await getItem("refreshToken")
        if (token && refreshToken)
            RootNavigation.dispatch(
                CommonActions.reset({
```

```
                    index: 1,
                    routes: [
                        { name: "HomeAuthScreen" }
                    ],
                })
            )
    }
```

Kode 4.14 Pengecekan apakah terdapat token tersimpan pada perangkat saat aplikasi dijalankan.

Saat mengakses halaman pada aplikasi terkadang harus mengakses *routes* yang memberikan data dari pengguna aplikasi. Pada beberapa *routes* yang memberikan data sensitif seperti data pengguna dapat juga diberikan proses autentikasi yang dikemas menjadi sebuah *middleware* express seperti pada kode 4.15 dan cara memanggilnya adalah dengan menyertakan *middleware* tersebut pada *route* yang hendak diberikan proses autentikasi seperti pada kode 4.16. Hal ini dapat dilakukan dengan mengirimkan *token* dan juga *refresh token* yang tersimpan pada aplikasi *client* saat melakukan API *request* ke *routes* yang memerlukan autentikasi. *Middleware* ini berisikan proses pengecekan terhadap *token* yang dikirimkan melalui API *request* dan memberikan tindakan berdasarkan hasil verifikasi dari *token* tersebut. Apabila *token* JWT telah kadaluarsa, terdapat fungsi yang akan mengecek bilamana terdapat *refresh token* yang dikirimkan melalui API *request*. *Refresh token* tersebut akan digunakan untuk membuat *token* yang baru. Apabila *refresh token* kadaluarsa, tidak ditemukan pada *database*, ataupun tidak dikirimkan melalui API *request*, maka *middleware* ini akan memberikan respon kadaluarsa dan pengguna tidak dapat mengirimkan API *request* ke *route* yang memerlukan proses autentikasi (apabila route yang dituju menggunakan *middleware* "authentication") sampai pengguna mendapatkan token JWT yang baru melalui proses login atau menggunakan *refresh token* yang masih tersimpan pada *database*.

```
const authentication = (req, res, next) => {

   const throwExpired = () => { return next(forbidden("Your
session is expired. Please sign in again.")) }

   const authHeader = req.headers['authorization']
   if (!authHeader)
      return res.status(400).json({
         message: "Unauthorized.",
```

```
            info: {}
        })

    const token = authHeader.split(" ")[1]

    jwt.verify(token, JWT_SECRET, async (err, tokenDecoded) => {

        if (err) {

            const refreshToken = req.body.refreshToken

            if (!refreshToken) return throwExpired()

            let id = ""

            jwt.verify(refreshToken, JWT_SECRET_2, (err,
tokenDecoded) => {
                if(err) return null
                id = tokenDecoded.id
            })

            if(!id) return throwExpired()

            const newToken = await generateTokenWRefreshToken(id,
refreshToken).catch(next)

            if (newToken === "401") return
next(unauthorized("Session ended."))
            if (newToken === "404") return next(notFound("User not
found."))

            req.newToken = newToken

            next()
        } else { next() }
    })
}
```

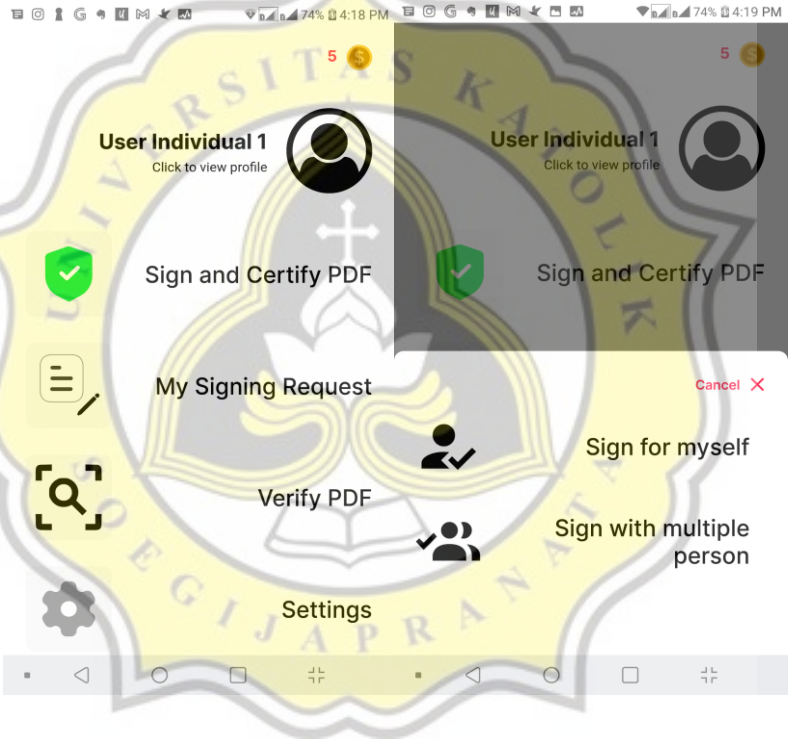
Kode 4.15 Middleware "authentication".

```
const getUser = router.post('/getuser', authentication, async
(req, res, next) => { ... getUser activity })
```

Kode 4.16 Contoh menggunakan middleware "authentication" pada sebuah route.

4.1.3 Penggunaan layanan penandatanganan secara individual (Individual Signing)

Pada aplikasi, untuk dapat melakukan penandatanganan pengguna harus mempunyai saldo. Apabila saldo tidak mencukupi, saat transaksi hendak dikirimkan akan tertolak. Biaya penandatanganan akan dikenakan pada setiap sekali proses penandatanganan. Saldo dari pengguna disimpan sebagai asset pada sistem blockchain. Karena setiap transaksi penandatanganan yang dilakukan, serta aktivitas transfer nominal asset akan dicatat riwayatnya pada sistem blockchain. Apabila pengguna sudah mempunyai saldo, pengguna dapat melakukan transaksi dengan menekan tombol "Sign and Certify PDF", kemudian memilih "Sign for myself" seperti pada gambar 4.28.

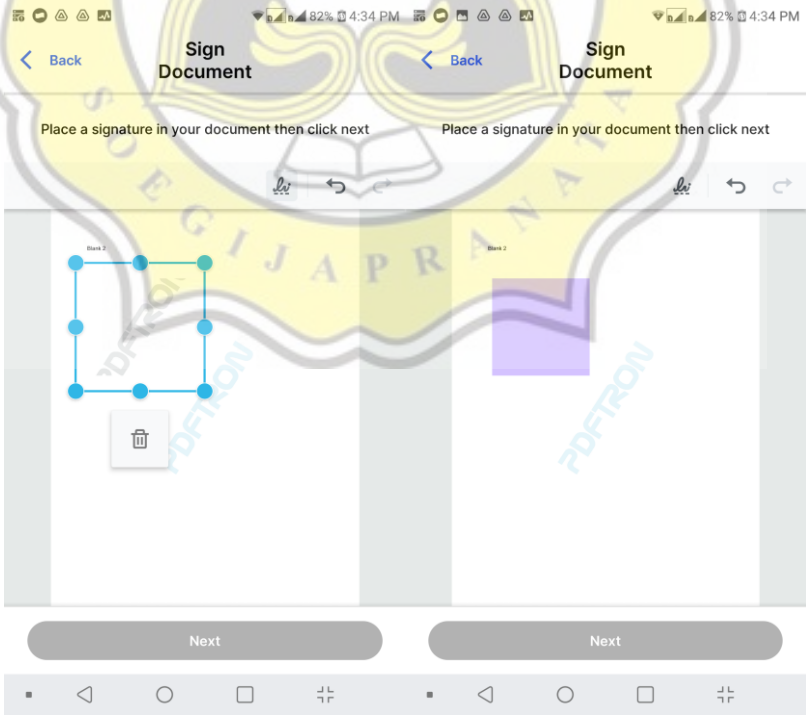


Gambar 4.28: Pilihan menu Sign and Certify PDF pengguna non-enterprise.

Sesudah itu, pengguna akan diarahkan ke halaman "Sign Document" seperti pada gambar 4.29. Pengguna dapat memilih dokumen berformat "PDF" yang hendak ditandatangani dengan menekan tombol "Press to browse".

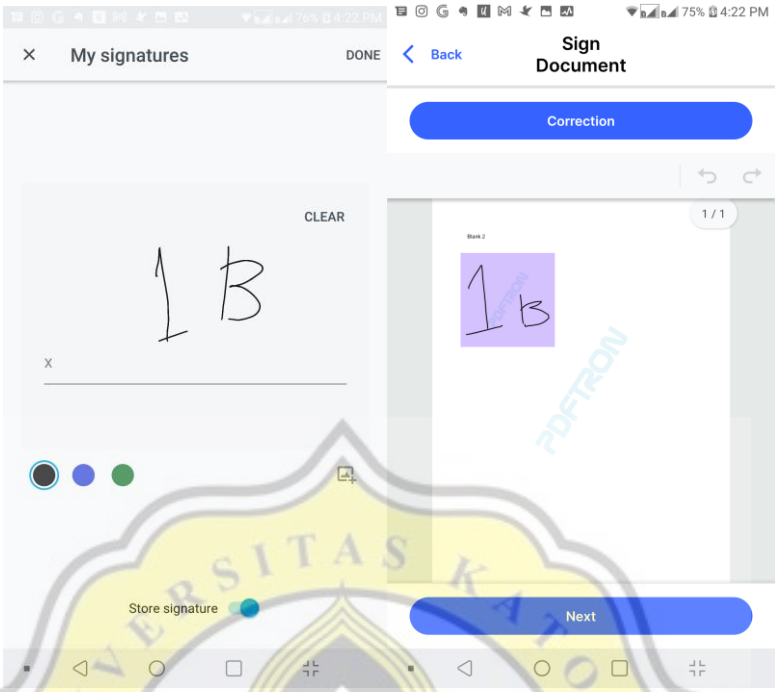
Gambar 4.29 Menu Sign Document.

Sesudah dokumen selesai dipilih, maka pengguna akan diarahkan ke halaman penandatanganan seperti pada gambar 4.29. Untuk meletakan tanda tangan dapat dilakukan seperti pada gambar 4.30.



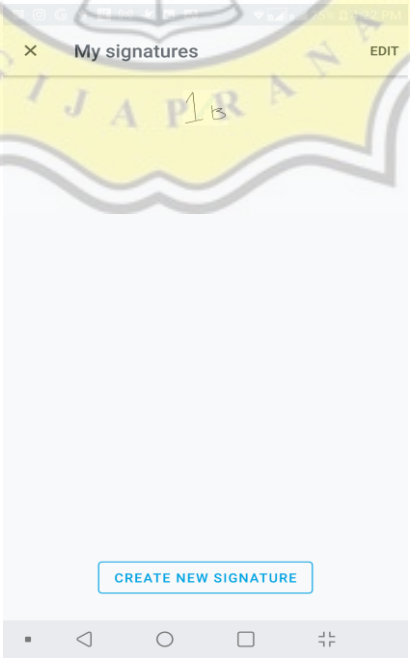
Gambar 4.30 Proses meletakan letak container tanda tangan.

Pada pengguna baru, pengguna harus membuat terlebih dahulu tanda tangannya yang kemudian akan disimpan ke dalam perangkatnya seperti pada gambar 4.31.



Gambar 4.31 Melakukan tanda tangan dengan tanda tangan baru.

Sesudah tanda tangan dibuat dan disimpan, maka pengguna dapat menempelkan tanda tangan yang telah dibuatnya tersebut pada setiap proses penandatanganan seperti pada gambar 4.32. Setelah pengguna menempelkan tanda tangannya, pengguna dapat menekan tombol “Next” untuk lanjut ke proses selanjutnya.



Gambar 4.32 Tampilan tanda tangan yang telah tersimpan.

Pada kode aplikasi *client* di setiap halaman yang menggunakan "React Native PDFTron" terdapat kode yang digunakan untuk menginisialisasi *engine* React Native PDFtron yang dipanggil pada *hook* React "*useEffect*". Apabila mempunyai *license key* PDFTron berformat string, dapat dimasukan sebagai *parameter* pertama pada fungsi "initialize".

```
useEffect(() => {
   RNPdftron.initialize("")
}, [])
```

Kode 4.17 Menjalankan sekali engine PDFTron pada aplikasi client saat halaman dimuat.

Lalu terdapat fungsi "retrieveSignatures" yang dibuat dengan tujuan untuk melakukan pembersihan *thumbnail cache* dari tanda tangan yang tersimpan pada perangkat, dan memastikan agar tanda tangan yang tersimpan dalam perangkat selalu *up-to-date* dengan tanda tangan terakhir yang tersimpan pada *database*. Fungsi ini akan dijalankan setiap PDFTron menerima perubahan terhadap dokumen ("onDocumentLoaded") dan juga saat pengguna aplikasi melakukan koreksi terhadap tanda tangan yang terpasang (dipanggil melalui fungsi "correction"). Hanya saja apabila fungsi ini dipanggil melalui "onDocumentLoaded", apabila nilai *state* "fetched" adalah *true* dan *parameter* "forced" bernilai *false* maka aktivitas sinkronisasi tanda tangan tidak akan berjalan karena fungsi ini tidak dipanggil menggunakan *parameter* "forced" bernilaikan *true*, dan *state* "fetched" bernilaikan *true*. Gunanya agar proses pengambilan tanda tangan hanya terjadi saat pertama kali dokumen dimuat, dan saat pengguna melakukan koreksi terhadap tanda tangan saja (tidak terpanggil saat dokumen mengalami perubahan). Berikut adalah fungsi "retrieveSignature" pada kode 4.18.

```
const retrieveSignatures = async ({ forced }) => {

   if (fetched && !forced) return null

   const thumbsDir = await
viewer.current.getSavedSignatureJpgFolder()
   const thumbs = await RNFetchBlob.fs.ls(thumbsDir)

   if (thumbs.length > 0) {
      thumbs.map(async (item) => {

      const thumbsPath = thumbsDir + "/" + item
```

```
        return await RNFetchBlob.fs.unlink(thumbsPath)
    })
}

    const sigExists = await viewer.current.getSavedSignatures()

    if (sigExists.length > 0) {

        sigExists.map(async (item) => {

            return await RNFetchBlob.fs.unlink(item)
        })
    }

    if (!signatures || signatures.length === 0) return
setFetched(true)

    const sigDir = await viewer.current.getSavedSignatureFolder()

    const latestSignature = signatures[signatures.length - 1]
    const data = latestSignature.data
    const name = latestSignature.name
    const file_path = sigDir + "/" + name

    await RNFetchBlob.fs.writeFile(file_path, data, 'base64')

    return setFetched(true)
}
```

Kode 4.18 Fungsi "retrieveSignatures".

Berikut adalah fungsi yang digunakan untuk melakukan *browse* terhadap file PDF pada aplikasi *front-end*.

```
const pickPdf = async () => {

    dispatch({ type: "IS_LOADING" })

    return await DocumentPick({
        type: "application/pdf"
    })
        .then(res => {
```

```
            if (res.type === "success") {
                setPdfFileUnsigned(res.uri)
                setPdfFile(res.uri)
                setPdfFileName(res.name)
            }
        })
        .catch(throwError)
        .finally(() => dispatch({ type: "NOT_LOADING" }))
}
```

Kode 4.19 Fungsi untuk memunculkan browser file pada ponsel dengan filter format file PDF.

Lalu pada React componentnya seperti pada kode 4.20, tampilan pada halaman penyematan tanda tangan diawali dengan *state* "pdfFile" saat masih kosong (nilai *boolean* "!pdfFile" menjadi *true* karena *state* bernilaikan *string* kosong) yang mengakibatkan hanya ditampilkannya tampilan untuk melakukan pemilihan dokumen yang hendak ditandatangani. Apabila pengguna aplikasi telah memilih dokumen, maka state "pdfFile" akan terisi, dan nilai *boolean* dari "!pdfFile" akan menjadi *false* yang mengakibatkan tampilan pada layar akan berganti ke proses penyematan tanda tangan.

```
<View style={styles.containerView}>
    {
        !pdfFile &&
            <View style={styles.padding}>
                <Text style={styles.textTitle}>Browse for PDF
Files</Text>
                <Spacing v={7} />
                <Button
                    onPress={() => pickPdf()}
                    label="Press to browse"
                    styleButton={styles.actionButton}
                    styleLabel={styles.textButton}
                />
            </View>
    }
    {
        pdfFile.length > 0 &&
            <View style={styles.containerView}>
                <View style={styles.buttonNextContainer}>
                    {!pdfSignatureId && <Text
style={styles.textGuide}>Place a signature in your document then
```

```
click next</Text>}
                {
                   pdfSignatureId.length > 0 &&
                      <Button
                         onPress={correction}
                         label="Correction"
                         styleButton={styles.actionButton}
                         styleLabel={styles.textButton}
                      />
                }
             </View>
             <DocumentView
                readOnly={!pdfSignatureId ? false : true}
                ref={viewer}
                document={pdfFile}
                isBase64String={true}
                hideTopAppNavBar={true}
                bottomToolbarEnabled={false}
                annotationToolbars={signToolbar}
                longPressMenuEnabled={false}
                hideToolbarsOnTap={false}
                onFormFieldValueChanged={onFormFieldValueChanged}
                onAnnotationChanged={onAnnotationChanged}
                 onDocumentLoaded={retrieveSignatures}
                 annotationAuthor={author}
              />
              <View style={styles.buttonNextContainer}>
                 <Button
                    onPress={checkPdf}
                    label="Next"
                    buttonColor={!pdfSignatureId ?
palette.black4 : palette.blue1}
                    styleButton={styles.actionButton}
                    styleLabel={styles.textButton}
                 />
              </View>
           </View>
    }
</View>
```

Kode 4.20 React Component pada halaman "PlaceSignatureScreen".

Terdapat fungsi untuk mendeteksi apabila terdapat perubahan terhadap *form field* dan *annotation*. Pada saat tanda tangan disematkan ke dokumen, akan terdapat perubahan *form field*. Untuk dapat menyimpan perubahan tersebut, dokumen akan disimpan dengan perubahan yang ada dalam format base64.

```
const onFormFieldValueChanged = async ({ fields }) => {

        const signatureField = fields[0].fieldName

        return await viewer.current.saveDocument().then(async
(base64) => {

            setPdfFile(base64.replace(/[\n\r]/g, ""))
            setPdfSignatureId(signatureField)

            const fileLists = await
viewer.current.getSavedSignatures()

            if (fileLists.length > 1) {
                const exceptLatest = fileLists.filter(item =>
item !== fileLists[fileLists.length - 1])
                exceptLatest.map(async (item) => { return await
RNFetchBlob.fs.unlink(item) })
            }
        })
    }

    const onAnnotationChanged = ({ action, annotations }) => {

        if (action === "add") {
            if (editCounter > 0) {
                setEditCounter(editCounter + 1)
            } else {
                setEditCounter(1)
            }
        }

        if (action === "delete") {
            setEditCounter(editCounter - 1)
            if (editCounter - 1 === 0 || editCounter - 1 < 0) {
                setPdfSignatureId("")
            }
```

```
        }
    }
```

Kode 4.21 Fungsi "onFormFieldValueChanged" dan "onAnnotationChanged"

Pada PDFTron React Native dimungkinkan juga untuk mengatur akan *user interface* dari komponen PDFTron dengan memberikan berbagai properti seperti contoh pada kode 4.22 yang digunakan untuk memberi pengaturan *tools* apa saja yang boleh ditampilkan pada toolbar PDFTron.

```
const signToolbar = [
    {
      [Config.CustomToolbarKey.Id]: 'signToolbar',
      [Config.CustomToolbarKey.Name]: 'signToolbar',
      [Config.CustomToolbarKey.Icon]:
Config.ToolbarIcons.FillAndSign,
      [Config.CustomToolbarKey.Items]:
[Config.Tools.formCreateSignatureField, Config.Buttons.undo,
Config.Buttons.redo]}
]
```

Kode 4.22 Konfigurasi toolbar apa saja yang boleh ditampilkan PDFTron.

Lalu terdapat juga fungsi correction yang digunakan apabila terdapat koreksi dari tanda tangan yang telah disematkan seperti pada kode 4.23 agar setiap perubahan pada dokumen dikembalikan seperti semula.

```
const correction = async () => {
    setPdfFile(pdfFileUnsigned)
    setPdfSignatureId("")
    setEditCounter(0)
    return await retrieveSignatures({ forced: true })
}
```

Kode 4.23 Fungsi "correction".

Apabila pengguna aplikasi menekan tombol "Next" maka fungsi "checkPdf" akan terpanggil untuk melakukan pengecekan sebelum dilanjutkan ke halaman selanjutnya (halaman konfirmasi) dengan mengirimkan semua *parameter* yang diperlukan seperti pada kode 4.24.

```
const checkPdf = async () => {
```

```
    if (!pdfSignatureId) return SimpleAlert("Please place a
signature")

    const fileLists = await viewer.current.getSavedSignatures()

    if (fileLists.length === 0) return SimpleAlert("No
signature.")

    const latestFileUri = fileLists[fileLists.length - 1]
    const latestFileUriSplitted = latestFileUri.split("/")
    const latestSignatureName =
latestFileUriSplitted[latestFileUriSplitted.length - 1]

    const latestSignature = await
RNFetchBlob.fs.readFile(latestFileUri, 'base64')

    return navigate("UploadPdfScreen", { pdfFileUnsigned,
pdfFile, pdfSignatureId, pdfFileName, latestSignature,
latestSignatureName })
}
```

Kode 4.24 Fungsi untuk melakukan pengecekan apabila tombol "Next" ditekan.

Kemudian pada halaman konfirmasi terdapat beberapa informasi yang harus dan dapat diisi secara opsional, yaitu judul dari dokumen, tujuan atau deskripsi, serta bilamana pengguna memperbolehkan informasi kontak dan lokasi pengguna untuk disematkan ke dalam dokumen "PDF" dan sistem aplikasi apabila pengguna tidak mengaktifkan "privacy mode" pada halaman konfirmasi (state "privacyMode" bernilai *false*). Pengambilan lokasi dilakukan di awal saat halaman konfirmasi dimuat dengan memanfaatkan *hook* React "useEffect" untuk menjalankan fungsi tersebut (fungsi "attachLocation" seperti pada kode 4.25).

```
const attachLocation = async () => {

    const hasPermission = await
Location.requestPermissionsAsync().catch(throwError)

    if (!hasPermission) return SimpleAlert("Please give this app
permission to use location in order to save location data.")

    Geocoder.init(googleApiToken)
```

```
    const pos = await Location.getCurrentPositionAsync({
accuracy: 1 }).catch(throwError)
    if (!pos) return null

    const lat = pos.coords.latitude
    const lng = pos.coords.longitude

    const jsonAddress = await Geocoder.from(lat, lng).catch(err
=> throwError(err.message))
    if(!jsonAddress) return null

    const formattedAddress =
jsonAddress.results[0].formatted_address

    setFullAddress(formattedAddress)
    return setLocation(pos)
}
```

Kode 4.25 Fungsi "attachLocation".

Sesudah semua informasi diisi dan semua informasi dirasa sudah benar, pengguna dapat menekan tombol "Confirm". Sebelumnya akan terdapat pengecekan terhadap data yang akan dikirimkan. Apabila data lolos validasi, maka pengguna akan diminta untuk melakukan verifikasi OTP SMS terlebih dahulu seperti pada kode 4.26.

```
const sendPrompt = async () => {

    if (
        !pdfFileUnsigned ||
        !pdfFile ||
        !pdfFileName ||
        //...rest of verification
    ) return SimpleAlert("Please fill all of required field.")

    if(!privacyMode && !location) return SimpleAlert("Please wait
until the map is loaded then try again.")

    return setOtpModal(true)
}
```

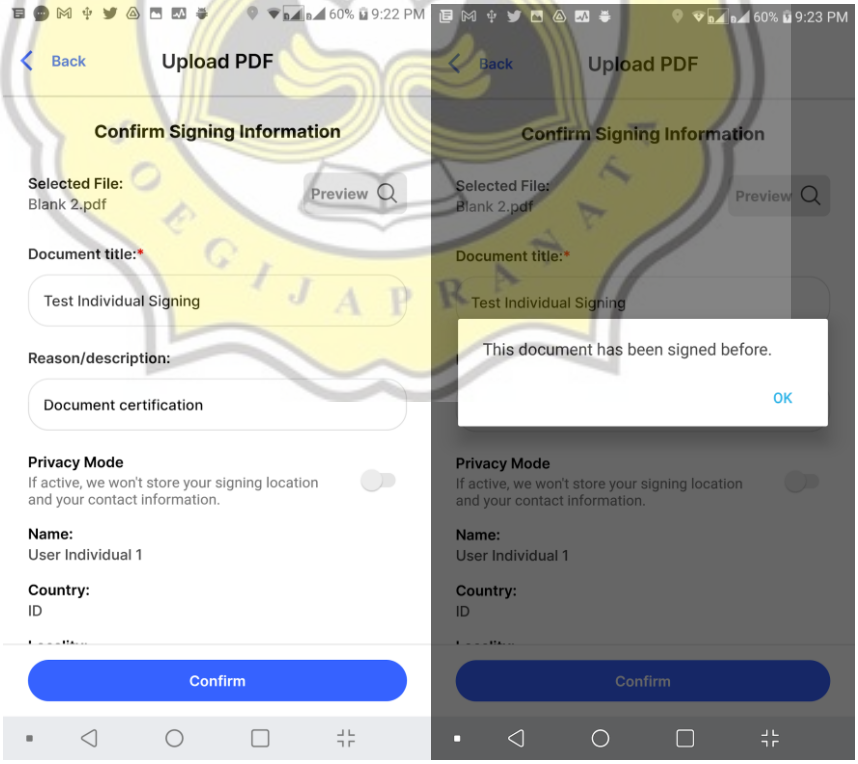
Kode 4.26 Validasi terhadap data yang akan dikirimkan.

Baru kemudian apabila verifikasi berhasil, maka aplikasi akan memanggil fungsi untuk mengirimkan informasi-informasi tersebut ke *endpoint* khusus penandatanganan individual yaitu *endpoint* "/pdfentry" melalui fungsi JavaScript "fetch" seperti pada kode 4.27.

```
const sendPdf = async () => {

   const lat = location ? location.coords.latitude : ""
   const lng = location ? location.coords.longitude : ""

   const latlng = lat ? ". Est lat: " + lat + ", Est lng: " +
lng + "." : ""
   const email = contact.email ? ", " + contact.email : ""

   const signatureProperties = {
      location: privacyMode ? "" : fullAddress + latlng,
      reason: pdfFileDesc,
      contactInfo: privacyMode ? "" : contact.phone + email
   }

   const body = {
      pdf_unsigned_b64: pdfFileUnsigned,
      pdf_signed_b64: pdfFile,
      pdf_signature_id: pdfSignatureId,
      fileName: pdfFileName,
      fileTitle: pdfFileTitle,
      fileDesc: pdfFileDesc,
      owner_id,
      signatureProperties,
      latestSignature,
      latestSignatureName
   }

   dispatch({ type: "IS_LOADING" })

   return await request.post("pdfentry", body)
      .then(async (response) => {

         const responseStatus = response.status
         const json = await response.json()
         const message = json.message
         const info = json.info
```

```
        if (responseStatus !== 200) return SimpleAlert(message)

        const file = info.data
        const now = simpleDateTimeNow()
        const dirs = RNFetchBlob.fs.dirs.DownloadDir
        const file_path = dirs + "/Certified - " + now + " -" +
pdfFileName

        setDownloadPath(file_path)
        setRequestSent(true)

        return await RNFetchBlob.fs.writeFile(file_path, file,
'base64')
      })
      .catch(throwError)
      .finally(() => dispatch({ type: "NOT_LOADING" }))
}
```

Kode 4.27 Contoh sederhana format fungsi pada aplikasi client untuk mengirimkan data tanda tangan.



Gambar 4.33 Halaman konfirmasi dan tampilan apabila dokumen ditemukan pada sistem.

Apabila dokumen yang pengguna kirimkan sebelumnya sudah pernah ditandatangani dikirimkan, dokumen tidak disematkan tanda tangan sama sekali melalui aplikasi dikirimkan, atau pengguna tidak mempunyai cukup saldo, maka permintaan akan ditolak. Hasilnya apabila ditemukan dokumen yang sama akan seperti pada gambar 4.33. Dan proses verifikasinya dapat dilihat pada kode 4.28.

```
const hasSignature = await CheckHasSignature(pdf_buffer)

if (!hasSignature) return next(badRequest("No signature field
found."))

await Db().catch(next)

const pdfExists = await PdfModel.findOne({ "pdf_unsigned.hash" :
pdf_unsigned_hash }, { "pdf_unsigned.hash": true })

if (pdfExists) return next(conflicted("This document has been
signed before."))
```

Kode 4.28 Pengecekan ada tidaknya tanda tangan serta apakah dokumen ditemukan pada database.

Berikut juga pada kode 4.29 adalah kode yang digunakan untuk mengecek ada atau tidaknya tanda tangan pada sebuah dokumen menggunakan library PDFTron.

```
import PDF from '@pdftron/pdfnet-node'
const PDFNet = PDF.PDFNet

const CheckHasSignature = async (pdf_buffer) => {

    const doc = await PDFNet.PDFDoc.createFromBuffer(pdf_buffer)

    doc.initSecurityHandler()

    const hasSignature = await doc.hasSignatures()

    return hasSignature
}

export default CheckHasSignature
```

Kode 4.29 Fungsi "CheckHasSignature".

Untuk dapat melakukan transaksi, akan dilakukan pengecekan terlebih dahulu apakah pengguna mempunyai saldo melalui query "getAccountAssets" pada Hyperledger Iroha seperti pada kode 4.30.

```
const getFundsRequestBody = {
   accountId: username + "@" + domainUsed,
   assetId: assetUsed + "#" + domainUsed
}

const accountAsset = await getAccountAssets(getFundsRequestBody)
const getAssetCode = accountAsset.status

if (getAssetCode !== 200) return next(accountAsset)

const getFundsInfo = accountAsset.info
const balance = getFundsInfo[0].balance

if (balance === "0") return next(paymentRequired())
```

Kode 4.30 Pengecekan apakah pengguna mempunyai jumlah saldo mencukupi.

Berikut adalah fungsi getAccountAsset yang digunakan untuk melakukan *query* informasi *asset* dari suatu *account* pada sistem Hyperledger Iroha.

```
const getAccountAssets = async ({ accountId, assetId }) => {

   if (!accountId || !assetId) return badRequest()

   const pageSize = 100

   return await queries.getAccountAssets(queryOptions, {
      accountId,
      pageSize,
      firstAssetId: assetId,
   })
      .then(info => {
         return {
            status: 200,
            message: "Success getting account asset of '" +
accountId + "'.",
            info,
         }
      })
```

```
        .catch((err) => {
            console.log(err)
            return paymentRequired()
        })
}
```

Kode 4.31 Fungsi untuk melakukan query "getAccountAssets" ke Hyperledger Iroha.

Apabila dokumen belum pernah ditandatangani sebelumnya dan pengguna mempunyai jumlah saldo yang mencukupi, maka proses penandatanganan dan penyematan *digital certificate* milik pengguna akan berjalan, disertai kemudian proses pemotongan saldo (asset) pengguna pada *blockchain* dan pencatatan transaksi ke sistem *blockchain* dengan keterangan atau deskripsi transaksinya berisikan *string* (teks) hasil dari proses *hashing* dokumen PDF yang telah disematkan tanda tangan *digital* beserta dengan sertifikat *digital*-nya. Proses pemotongan saldo pengguna dilakukan dengan mengirimkan saldo pengguna ke "admin Hyperledger Iroha" saat proses transfer asset (pencatatan transaksi). Proses *hashing* dilakukan dengan menggunakan *library* "hasha", dengan inputan berupa Base64 string dari dokumen yang telah selesai disertifikasi, format *encoding* "hex" *string*, dan algoritma yang digunakan adalah SHA-256 seperti pada kode 4.32.

```
const hashBase64 = async (base64) => {

    const hash = await hasha.async(base64, {
        encoding: "hex",
        algorithm: "sha256"
    })

    return hash
}
```

Kode 4.32 Fungsi yang digunakan untuk melakukan hashing dokumen.

Untuk melakukan pembuatan arsip sertifikat berformat PKCS#12 dapat memanfaatkan *library* "node-forge". Dengan memberikan informasi *certificate chain*, *private key*, dan *password* untuk mengunci arsip PKCS#12 yang dibuat seperti pada kode 4.33.

```
const pemToP12 = (certificateChain, privateKey, name, password)
=> {
```

```
    const pki = forge.pki

    var privateKeyForge = pki.privateKeyFromPem(privateKey)
    const p12Asn1 = forge.pkcs12.toPkcs12Asn1(privateKeyForge,
certificateChain, password)
    const p12Der = forge.asn1.toDer(p12Asn1).getBytes()
    const p12b64 = forge.util.encode64(p12Der)

    return p12b64
}
```

Kode 4.33 Fungsi untuk membuat arsip PKCS#12.

Terdapat juga fitur lain pada *library* PDFTron apabila hendak menyematkan *stamp* atau *annotation* lainnya ke dokumen yang dipilih. Kali ini sebagai contoh QR Code akan dibuat dan akan digunakan untuk melakukan verifikasi terhadap transaksi yang telah dilakukan. QR Code akan menyimpan *hash* dari PDF yang belum ditandatangani, yang dapat digunakan untuk mencari data pada *database*. Kemudian, QR Code akan dikonversi menjadi format JPEG agar lebih mudah dibaca pada PDFTron untuk kemudian disematkan ke dalam dokumen seperti pada kode 4.34.

```
const url = BCWEB_ADDRESS + "getpdfid?id=" +
pdf_unsigned_hash_hex
const bcb64 = await QRCode.toDataURL(url)
const bcb64_stripped = bcb64.split(',')[1]
const bcpngbuff = await base64ToBuffer(bcb64_stripped)
const bcbuff = await Jimp.read(bcpngbuff).then((image) => {
    return image.getBufferAsync(Jimp.MIME_JPEG) })
```

Kode 4.34 Pembuatan dan konversi QRCode.

Untuk melakukan konversi sederhana base64 ke dalam format buffer dibuat fungsi seperti pada kode 4.35.

```
const base64ToBuffer = (base64) => {
    return new Buffer.from(base64, 'base64')
}
```

Kode 4.35 Fungsi yang membantu dalam proses pengkonversian string base64 ke bentuk buffer.

Kegiatan pengolahan dokumen PDF sebenarnya terjadi saat menjalankan fungsi “CertifyPDF” seperti pada kode 4.36. Yang mana untuk menggunakannya harus

menyertakan *parameter buffer* dari dokumen yang telah ditandatangani melalui aplikasi *front-end* (*client*), identitas (penanda) *field* dari tanda tangan yang telah dibuat pada aplikasi *front-end*, *buffer* dari arsip PKCS#12 yang telah dibuat sebelumnya, serta *password* dari arsip PKCS#12 tersebut yang sebelumnya telah ditentukan. *Parameter* "signatureProperties" adalah *parameter* opsional yang dapat disematkan untuk memberi informasi seperti lokasi, deskripsi, serta kontak dari pelaku penandatangan. Kemudian terdapat *parameter* untuk menandai bahwa dokumen harus dikunci atau tidaknya. Penguncian dokumen bertujuan agar dokumen tidak dapat diedit lagi setelah dokumen disimpan (melalui *parameter* keenam dengan diset menjadi *true*). Lalu pada *parameter* terakhir diisikan *buffer* dari gambar QR Code yang telah dibuat sebelumnya untuk kemudian disematkan ke dalam dokumen.

```
const certifiedPdf = await CertifyPDF(pdf_buffer,
certfield_name, privp12_buffer, p12Password,
signatureProperties, true, bcbuff)
const certifyError = certifiedPdf.error

if (certifyError) return next({ message: certifyError })

   const pdf_certified_b64 = base64Encode(certifiedPdf.buffer)
   const pdf_certified_hash = await
hashBase64(pdf_certified_b64)
```

Kode 4.36 Menjalankan fungsi "CertifyPDF" kemudian menangkap responnya.

Berikut pada kode 4.37 adalah detail dari fungsi "Certify PDF" yang dipanggil pada kode 4.36.

```
const CertifyPDF = async (pdf_buffer, certfield_name,
privp12_buffer, privp12_pass, signatureProperties, final,
bcbuff) => {

 const main = async () => {

   const doc = await PDFNet.PDFDoc.createFromBuffer(pdf_buffer)

   if (bcbuff) {

     const pageCount = await doc.getPageCount()
     const page = await doc.getPage(pageCount)
     const pageWidth = await page.getPageWidth()
```

```
      const pageHeight = await page.getPageHeight()
      const allPage = await PDFNet.PageSet.createRange(1,
pageCount)
      const matriX = pageWidth - (pageWidth*0.20)
      const matriY = pageHeight - (pageHeight*0.90)

      let img = await PDFNet.Image.createFromMemory2(doc, bcbuff)
      let imgWidth = await img.getImageWidth()
      let imgHeight = await img.getImageHeight()

      //just put regular qr image at end of document
      const builder = await PDFNet.ElementBuilder.create()
      const writer = await PDFNet.ElementWriter.create()
      writer.beginOnPage(page,
PDFNet.ElementWriter.WriteMode.e_overlay)
      let element = await builder.createImageScaled(img, matriX,
matriY, imgWidth/2, imgHeight/2)
      writer.writePlacedElement(element)
      writer.end()

      //stamp with opacity
      const stamper = await
PDFNet.Stamper.create(PDFNet.Stamper.SizeType.e_absolute_size,
pageWidth, pageHeight)
      stamper.setOpacity(0.4)
      stamper.setSize(PDFNet.Stamper.SizeType.e_absolute_size,
imgWidth/2, imgHeight/2)

stamper.setAlignment(PDFNet.Stamper.HorizontalAlignment.e_horizo
ntal_left, PDFNet.Stamper.VerticalAlignment.e_vertical_bottom)
      stamper.setPosition(matriX, matriY)
      stamper.stampImage(doc, img, allPage)
   }

   const location = signatureProperties.location
   const reason = signatureProperties.reason
   const contactInfo = signatureProperties.contactInfo
   const permission =
PDFNet.DigitalSignatureField.DocumentPermissions.e_no_changes_al
lowed

   doc.initSecurityHandler()
```

```
    const certification_sig_field = await
doc.getDigitalSignatureField(certfield_name)

    return await Promise.all([
      await
certification_sig_field.setFieldPermissions(PDFNet.DigitalSignat
ureField.FieldPermissions.e_include, [certfield_name]),
      final && await
certification_sig_field.setDocumentPermissions(permission),
      await
certification_sig_field.signOnNextSaveFromBuffer(privp12_buffer,
privp12_pass),
      await certification_sig_field.setLocation(location),
      await certification_sig_field.setReason(reason),
      await certification_sig_field.setContactInfo(contactInfo)
    ])
      .then(async () => {

        const docBuffer = await
doc.saveMemoryBuffer(PDFNet.SDFDoc.SaveOptions.e_incremental)

        return {
          error: "",
          buffer: docBuffer
        }
      })
      .catch(err => {

        return {
          error: err.toString(),
          buffer: ""
        }
      })
  }

  return await main()
}
```

Kode 4.37 Fungsi "CertifyPDF".

Berikut pada kode 4.38 adalah fungsi yang digunakan untuk melakukan pencatatan transaksi dari pengguna ke sistem *blockchain* Hyperledger Iroha. Informasi

yang dibutuhkan adalah *private key* akun Hyperledger Iroha pengguna, sumber akun yang diisikan dengan nilai akun pengguna, destinasi akun ke akun admin Hyperledger Iroha, informasi *asset ID*, dan deskripsi transaksi berisikan *hash* dari dokumen yang telah melewati proses sertifikasi.

```
const saveToBlockchainResult = await addHashViaTx({
   privateKey: privateKeyWallet,
   srcAccountId: username + "@" + domainUsed,
   destAccountId: IROHA_ADMIN + "@" + IROHA_DOMAIN,
   assetId: assetUsed + "#" + domainUsed,
   description: pdf_certified_hash
})

const saveCode = saveToBlockchainResult.status

if (saveCode !== 200) return
res.status(saveCode).json(saveToBlockchainResult)

const txHash = saveToBlockchainResult.info
```

Kode 4.38 Melakukan proses pencatatan transaksi dalam system blockchain.

Tanda tangan dari pengguna juga akan disimpan berdasarkan apabila ditemukan atau tidaknya data tanda tangan yang sama yang dikirimkan saat melakukan *request* penandatanganan seperti pada kode 4.39.

```
const signatureSaved = userData.signature

await saveUserSignature({ signatureSaved, latestSignature,
latestSignatureName, created, owner_id_obj })
```

Kode 4.39 Menyimpan tanda tangan yang disertakan dalam request.

Berikut pada kode 4.40 adalah rincian aktivitas pada fungsi "saveUserSignature" yang dipanggil pada kode 4.39.

```
const saveUserSignature = async ({ signatureSaved,
latestSignature, latestSignatureName, created, owner_id_obj })
=> {

   await Db()

   if (!latestSignature || !latestSignatureName) return null
```

```
    const userNewSignature = {
        data: latestSignature,
        name: latestSignatureName,
        created
    }

    const modified = {
        time: created,
        modifier_id: owner_id_obj.toString()
    }

    if (signatureSaved && signatureSaved.length > 0) {

        const signatureSavedData = signatureSaved.map(item =>
item.data)
        const foundSameSignature =
signatureSavedData.includes(latestSignature)

        if(foundSameSignature) return null
    }

    return await UserModel.updateOne(
        { _id: owner_id_obj },
        { $push: { modified, signature: userNewSignature } }
    )
}
```

Kode 4.40 Fungsi "saveUserSignature" yang digunakan untuk menyimpan tanda tangan.

Dan berikut pada kode 4.41 adalah kode yang digunakan untuk menyimpan aktivitas ke dalam *database* MongoDb kemudian memberikan respon sukses apabila tidak terdapat masalah.

```
return await PdfModel.create({...data})
    .then(() => {
        return res.status(200).json({
            message: "PDF Certified successfully.",
            info: {
                data: pdf_certified_b64,
                hash: txHash
            }
        })
```

```
    })
    .catch(next)
```

Kode 4.41 Fungsi untuk melakukan input ke database setelah semua aktivitas dilakukan, serta respon API yang memberikan dokumen yang telah disertifikasi.

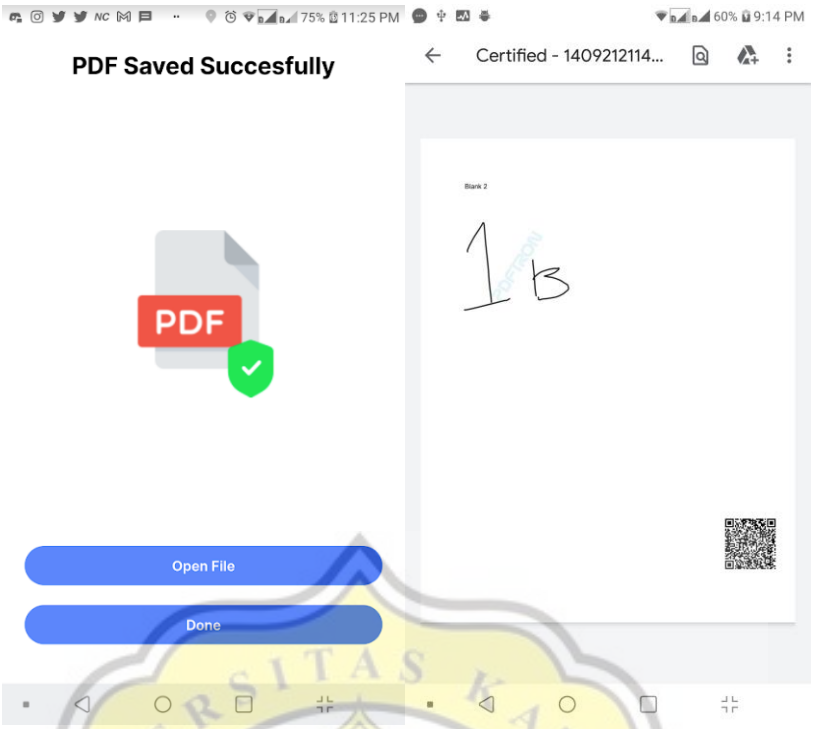
Jangan lupa apabila menggunakan fungsi yang menggunakan *library* PDFTron pada Node.Js sama halnya pada *library* PDFTron pada React Native juga harus menjalankan terlebih dahulu fungsi untuk melakukan inisialisasi engine PDFNet yang dapat dilakukan dengan memanggil fungsi "initialize" yang dapat ditemukan pada class PDFNet, di awal kode, atau dengan membungkus seluruh fungsi yang menggunakan PDFTron kemudian memasukan fungsi yang telah dibungkus tersebut ke dalam *parameter* pertama pada fungsi "runWithCleanup". Apabila mempunyai *license key* dari PDFTron *string license* tersebut dapat dimasukan menjadi *parameter* kedua pada fungsi "runWithCleanup" atau sebagai *parameter* pertama apabila menggunakan fungsi "initialize".

```
return await PDFNet.runWithCleanup(pdf_task).catch(next)
```

Kode 4.42 Fungsi "runWithCleanup".

Setelah semua proses selesai, maka seluruh kegiatan tersebut juga dicatat oleh *database* untuk dapat menyimpan beberapa informasi yang tidak dapat disimpan seluruhnya ke dalam sistem *blockchain* sebagai *record*. Barulah API akan memberikan respon balikan JSON ke *client application* (aplikasi *front-end*) berupa pesan bahwa proses telah selesai, dan juga informasi berupa *string* Base64 yang merupakan hasil *encoding* dari dokumen PDF yang telah ditandatangani dan disertifikasi.

Kemudian pada aplikasi akan tampil *modal* (*pop-up*) apabila proses telah sukses dilakukan, dan pengguna akan diberi opsi apakah pengguna mau membuka dokumen yang telah disertifikasi dengan menekan tombol "Open File", atau menekan tombol "Continue" untuk kembali ke halaman utama seperti pada gambar 4.34. Apabila menekan tombol "Download" maka aplikasi akan melakukan proses *decoding string* Base64 yang didapat dari API menjadi format *blob* PDF yang kemudian dapat disimpan ke dalam perangkat pengguna.



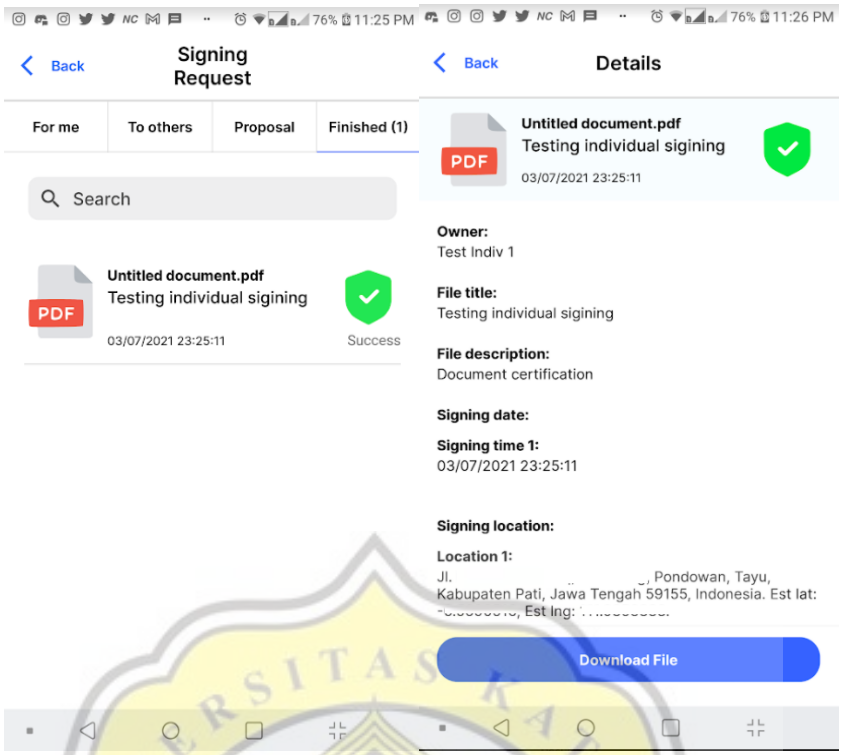
Gambar 4.34 Pop-up apabila transaksi telah selesai dilakukan oleh penandatangan terakhir, dan tampilan dokumen yang telah dibuka.

Apabila dokumen telah tersimpan, maka aplikasi secara otomatis akan membuka dokumen tersebut dengan aplikasi penampil PDF *external* pada perangkat pengguna debgan menjalankan fungsi "openFile" seperti pada kode 4.43.

```
const openFile = async () => { return await
FileViewer.open(downloadPath).catch(throwError) }
```

Kode 4.43 Membuka file yang telah tersimpan pada aplikasi client.

Pengguna masih dapat melihat riwayat transaksinya kemudian mengunduh dokumennya pada menu "My Signing Request" pada tab "Finished" seperti pada gambar 4.35.



Gambar 4.35 Dokumen selesai melewati proses penandatanganan.

4.1.4 Penggunaan layanan penandatanganan secara kelompok (multiple signing)

Di tipe transaksi ini, jumlah penandatanganan harus dilakukan oleh lebih dari satu pengguna, dan masing-masing pengguna harus mempunyai saldo yang mencukupi untuk dapat melakukan penandatanganan, sehingga ini juga dapat menjadi bukti bahwa pengguna melakukan transaksi dengan niat nya sendiri (dengan pengguna melakukan pembayaran terhadap setiap transaksi penandatanganan yang Ia lakukan).

Ada dua cara dalam proses penandatanganan kelompok. Yang pertama adalah pengguna menandatangani terlebih dahulu sebuah dokumen, kemudian pada halaman konfirmasi setelah mengisi informasi yang diperlukan seperti judul dan deskripsi dokumen, pengguna memilih pengguna lain yang dapat menandatangani dokumen dengan menekan tombol “Invite user to sign this document”.

Setelah semua informasi terisi, maka pengguna menekan tombol “Upload”. Maka aplikasi akan mengirimkan seluruh informasi melalui *endpoint* “/pdfentrymulti”. Apabila API menerima permintaan berisikan informasi-informasi yang diperlukan, sistem akan melakukan pengecekan terlebih dahulu apakah terdapat data PDF pada *database*. Bilamana ada apakah pengguna yang bersangkutan telah melakukan penandatanganan. Apakah status dari transaksi telah selesai. Bilamana dokumen tidak

ditemukan, maka sistem akan melakukan aktivitas seperti sebelumnya yaitu menyematkan tanda tangan dan sertifikat pengguna kedalam PDF, namun pertama kali PDF tidak akan dikunci dengan *permission* agar pengguna selanjutnya dapat melakukan penandatanganan. Apabila sudah, maka transaksi akan dicatat ke dalam sistem *blockchain* dan juga sekaligus dengan pengurangan saldo dari pengguna. Kemudian sistem akan mencatat kegiatan tersebut ke dalam *database* tetapi dengan status transaksi dengan status "On process" karena masih ada pengguna lain yang harus menandatangani dokumen tersebut. Pengguna yang berperan sebagai inisiator dapat melihat status transaksinya pada menu "My Signing Request" dan pada tab "To others", namun apabila semua proses terselesaikan pengguna dapat melihat dokumen yang telah selesai pada tab "Finished".

Pengguna lain yang diberi hak dan kewajiban untuk menandatangani dokumen dapat melihat permintaan penandatanganan pada menu "My Signing Request" pada tab "For me", kemudian untuk menandatangani dokumen pengguna tinggal memilih dokumen mana yang ingin ditandatangani, kemudian pengguna akan diarahkan ke halaman penandatanganan. Terdapat perbedaan pada pengguna "*non-initiator*" pada halaman konfirmasi. Pada halaman konfirmasi pengguna tidak dapat menulis atau mengganti kolom judul dokumen, deskripsi dokumen, dan juga siapa saja yang boleh menandatangani dokumen, namun pengguna dapat memilih untuk menyertakan informasi personalnya (lokasi dan kontak) atau tidak ke dalam file PDF dan juga sistem. Kemudian apabila sudah pengguna menekan tombol "Upload" untuk mengirimkan semua informasi tersebut ke *endpoint* "/pdfentrymulti". API akan melakukan pengecekan terlebih dahulu seperti pada kode 4.44. Apakah terdapat tanda tangan pada dokumen? Apakah pengguna yang menandatangani belum melakukan penandatanganan? Apabila terdapat tanda tangan pada dokumen dan pengguna belum melakukan penandatanganan dan berada pada *list* penandatangan? Maka kegiatan penandatanganan yang sama akan dilakukan dengan perbedaan apabila pengguna yang menandatangani adalah pengguna terakhir yang belum menandatangani, maka sistem akan mengunci dokumen tersebut agar tidak dapat ditandatangani ataupun dirubah lagi, barulah kemudian saat pencatatan data pada *database* status dokumen akan diubah menjadi selesai.

```
const pdfTronCallback = async () => {
```

```
    if (
        !pdf_unsigned_hash ||
        !pdf_signature_id
    ) return next(badRequest())

    const signer_id_obj = mongoose.Types.ObjectId(signer_id)
    const hasSignature = await CheckHasSignature(pdf_buffer)

    if (!hasSignature) return next(badRequest("No signature field
found."))
```

Kode 4.44 Proses validasi dokumen yang dikirimkan.

Apabila pada *database* tidak ditemukan *hash* PDF yang belum ditandatangani, maka akan dilakukan sertifikasi dokumen tanpa mengunci *permission* dari *file* PDF yang telah ditandatangani agar penandatangan selanjutnya dapat membuat perubahan pada *file* tersebut (seperti pada kode 4.45).

```
if (!pdfData) {
    const certified = await CertifyPDF(pdf_buffer,
certfield_name, privp12_buffer, p12Password,
signatureProperties, false)
    const certifiedError = certified.error
    const certifiedBuffer = certified.buffer

    if (certifiedError) return next({ message: certifiedError })

    const pdf_certified_b64 = base64Encode(certifiedBuffer)
    const pdf_certified_hash = await
hashBase64(pdf_certified_b64)
```

Kode 4.45 Proses validasi dokumen terhadap database dan tindakan selanjutnya.

Apabila ditemukan pada *database hash* yang sama dari *file* PDF yang belum ditandatangani, dan terdapat properti status dengan nilai 1 yang berarti aktivitas telah selesai dilakukan maka *endpoint* akan mengakhiri *request* dengan memberikan respon dengan status 409 berisikan pesan bahwa dokumen sebelumnya pernah melewati proses penandatanganan. Kemudian terdapat juga pengecekan apakah pengguna yang bersangkutan mempunyai hak akses untuk menandatangani dokumen untuk alasan keamanan. Kemudian apabila telah ditemukan bahwa pengguna telah mengambil peran untuk menandatangani dokumen sebelumnya, maka *endpoint* akan mengakhiri *request*

dengan memberikan respon dengan status 409 dengan berisikan keterangan bahwa pengguna tersebut telah melakukan tanda tangan sebelumnya. Proses pengecekan yang disebutkan dapat dilihat pada kode 4.46.

```
if (pdfData.status === 1) return next(conflicted("PDF already
signed successfully before."))

if (pdfData.status === 0) {

   const certifiedLists = pdfData.pdf_certified
   const allow_sign_pdf = pdfData.allow_sign

   if (!allow_sign_pdf.includes(signer_id)) return
next(forbidden("User don't have permission to sign this
document."))

   certifiedLists.map((item) => {
      const signInfo = item.signingInfo
      if (signInfo.signer_id === signer_id) return
next(conflicted("Already signed."))
   })
```

Kode 4.46 Pengecekan lebih lanjut terhadap dokumen yang ditemukan pada database.

Kemudian terdapat juga pengecekan untuk mengetahui sedang berada di mana tahap penandatanganan yang dilakukan seperti pada kode 4.47. Apabila jumlah pengguna yang belum melakukan penandatanganan hanya tersisa satu, dan pengguna berada pada daftar pengguna yang belum menandatangani, maka tindakan yang dilakukan adalah melakukan tahap penandatanganan terakhir dengan mengunci dokumen dan menyematkan QR Code yang dapat digunakan untuk melakukan verifikasi terhadap transaksi penandatanganan dokumen.

```
   const alreadySignedList = certifiedLists.map(item =>
item.signingInfo.signer_id)
   const notYetSignedList = allow_sign_pdf.filter(item =>
!alreadySignedList.includes(item))

if (
   notYetSignedList.length === 1 &&
   notYetSignedList.includes(signer_id)
) { //Do last signing activity }
```

Kode 4.47 Logika untuk menentukan tindakan penandatanganan tahap terakhir.

Pada proses tahap terakhir, proses penandatanganan dilakukan dengan memberikan nilai *true* sebagai *parameter* keenam, serta *buffer* dari *file* gambar berformat JPG dari QR Code yang hendak disematkan ke dalam PDF seperti pada kode 4.48. *Parameter* nilai *true* diberikan untuk menandai agar dokumen dapat dikunci sehingga tidak ada perubahan yang boleh dilakukan pada dokumen setelah proses sertifikasi dilakukan. Kemudian akan terjadi proses seperti sebelumnya yaitu pencatatan *hash* dari *file* yang telah selesai melewati proses sertifikasi sebagai transaksi pada sistem *blockchain*, baru kemudian file akan dicatat ke dalam *database*.

```
await CertifyPDF(pdf_buffer, certfield_name, privp12_buffer,
p12Password, signatureProperties, true, bcbuff)
```

Kode 4.48 Fungsi sertifikasi dokumen pada tahap terakhir.

Pada *endpoint* ini apabila proses sertifikasi telah selesai, maka akan terjadi penyimpanan tanda tangan seperti pada sebelumnya, dan menyimpan perubahan dokumen pada *database* dengan merubah status menjadi 1, kemudian mencatat perubahan yang telah terjadi, baru kemudian memberikan respon terhadap *client application* berdasarkan pada hasil saat menyimpan data ke dalam *database* (seperti pada kode 4.49).

```
return await PdfModel.updateOne(
   { "pdf_unsigned.hash": pdf_unsigned_hash },
   {
      $set: { status: 1 },
      $push: { pdf_certified, modified }
   })
   .then(() => {
      return res.status(200).json({
         message: "All signing process has been done
successfully.",
         info: {
            data: pdf_certified_b64,
            hash: txHash
         }
      })
   })
   .catch(next)
```

Kode 4.49 Fungsi untuk mencatat perubahan ke database.

Apabila ditemukan hasil bahwa yang belum menandatangani dokumen lebih dari satu orang, dan pengguna terdapat pada daftar penandatangan, nilai *parameter* keenam diset menjadi *false* sebagai penanda bahwa dokumen untuk tidak dikunci terlebih dahulu agar penandatangan selanjutnya dapat menandatangani dokumen tersebut dan sistem dapat menempelkan QR Code pada tahap akhir (seperti pada kode 4.50).

```
if (
   notYetSignedList.length > 1 &&
   notYetSignedList.includes(signer_id)
) {
   const certified = await CertifyPDF(pdf_buffer,
certfield_name, privp12_buffer, p12Password,
signatureProperties, false)
```

Kode 4.50 Logika untuk melakukan proses sertifikasi tanpa mengunci dokumen.

Pada tahap ini, proses yang berjalan hampir sama dengan proses saat pada penandatanganan tahap akhir, hanya saja status dibiarkan agar tetap bernilai 0 (on process) sebagai tanda bahwa proses penandatanganan belum selesai dan dapat dilanjutkan pada tahap berikutnya, serta respon yang diberikan tidak membawa *file* dari dokumen yang telah ditandatangani (sepert pada kode 4.51).

```
return await PdfModel.updateOne(
   { "pdf_unsigned.hash": pdf_unsigned_hash },
   {
      $set: { status: 0 },
      $push: { pdf_certified, modified }
   })
   .then(() => {
      return res.status(200).json({
         message: "Signing process done and has been passed to
the next user on signing list.",
         info: txHash
      })
   })
   .catch(next)
```

Kode 4.51 Fungsi untuk mencatat perubahan ke database dengan status on process.

Apabila semua penandatangan sudah melakukan penandatanganan dokumen, maka pengguna terakhir yang telah menandatangani dapat mengunduh dokumen

tersebut apabila *pop-up* “sukses” telah muncul pada aplikasi *front-end* dengan cara menekan tombol “Download PDF” atau menekan “Continue” apabila pengguna hendak mengunduhnya lain waktu beserta dengan pengguna lainnya juga dapat melihatnya pada menu “My Signing Request” pada tab “Finished”.

Pengguna juga dapat mengirimkan permintaan penandatanganan ke beberapa pengguna tanpa harus menandatangani dokumen tersebut terlebih dahulu, dengan membiarkan dokumen kosong tanpa ditandatangani sama sekali sebelum dikirimkan, tetapi pengguna harus juga menyertakan informasi siapa saja yang harus menandatangani dokumen tersebut pada halaman konfirmasi. Biaya hanya dikenakan pada pengguna yang melakukan penandatanganan, namun pada riwayat di *database* tetap tercatat siapa yang menjadi inisiasi kegiatan penandatanganan. Proses pengecekan request dan penyimpanan request tersebut dapat dilihat pada kode 4.52.

```
if (!signatureProperties) {

   if (
      !fileName ||
      !fileTitle ||
      !owner_id ||
      !allow_sign
   ) return next(badRequest())

   await Db().catch(next)

   return await PdfModel.create({
      pdf_unsigned,
      fileName,
      fileTitle,
      fileDesc,
      owner_id,
      allow_sign,
      status: 0,
      created,
      deleted,
   })
      .then(() => {
         return res.status(200).json({
            message: "Signing request sent.",
            info: {}
         })
```

```
        })
        .catch(next)
}
```

Kode 4.52 Proses menyimpan data request penandatanganan.

Perlu diingat tanda tangan atau sertifikat pengguna tidak akan terpasang apabila yang melakukan inisiasi penandatanganan tidak menandatangani dokumen tersebut, dan *record* pelaku inisiasi hanya akan tercatat ke dalam *database* dan tidak pada sistem blockchain, karena pengguna tidak menandatangani dokumen tersebut dan pengguna tidak diminta atau memberikan biaya apapun.

4.1.5 Penggunaan layanan enterprise (group) signing pada sisi enterprise admin

Pada sisi admin, *admin* harus mempunyai saldo terlebih dahulu agar dapat melakukan transaksi atau kegiatan penandatanganan. Untuk melakukan penandatanganan, pengguna admin memilih menu "Sign and Certify PDF" pada menu utama, kemudian pada *drawer* yang terbuka memilih "Sign with multiple person". Kemudian pengguna akan diarahkan ke halaman pemilihan dokumen. Pengguna akan memilih dokumen yang hendak ditandatangani. Apabila sudah, admin dapat langsung menandatangani dokumen yang Ia inginkan, atau menekan tombol "Next" untuk mengirimkan perintah penandatanganan ke anggotanya di halaman konfirmasi dengan menekan tombol "Set who allowed to sign this document", dan tidak lupa juga untuk mengisi informasi yang harus diisi seperti judul dan deskripsi dokumen. Apabila semua *field* yang diperlukan telah terisi, pengguna dapat menekan tombol "Upload" untuk mengirimkan dokumen beserta keterangannya. Maka permintaan penandatanganan akan dikirimkan ke pengguna yang bersangkutan.

Perlu diingat bahwa pada transaksi penandatanganan enterprise (group), berlaku alur dana (*cash flow*) yang berbeda dari transaksi biasa (*non-enterprise*). Pada transaksi *enterprise*, hanya saldo *admin* yang digunakan sebagai media pembayaran. Pada sistem API akan melakukan pengecekan tipe akun yang melakukan transaksi. Apabila akun yang digunakan pengguna adalah akun tipe *admin*, maka saldo *admin* akan terpotong langsung dan dikirimkan ke akun "admin sistem blockchain" sebagai *record*. Tetapi apabila tipe akun pengguna yang digunakan adalah akun anggota (*member*) *enterprise*, akan terjadi beberapa alur yang harus dilalui di belakang seperti pada kode 4.54. Yaitu

akun *admin enterprise* harus mempunyai saldo yang mencukupi dengan melakukan pengecekan seperti pada kode 4.53.

```
const organizationStripped =
organization.toLowerCase().replace(/\s/g, '')
const adminIrohaId = adminId.id + "@" + organizationStripped
const orgAssetId = organizationStripped + "#" +
organizationStripped

const getFundsRequestBody = {
   accountId: adminIrohaId,
   assetId: orgAssetId
}

const getFunds = await getAccountAssets(getFundsRequestBody)
const getFundsCode = getFunds.status

if (getFundsCode !== 200) return next(getFunds)
```

Kode 4.53 Mengecek apakah admin mempunyai saldo cukup untuk melakukan transaksi enterprise.

Kemudian apabila saldo *admin* mencukupi, sistem akan memindahkan sejumlah saldo yang diperlukan dari akun *admin enterprise* ke anggota yang bersangkutan (yang melakukan tanda tangan) seperti pada kode 4.54, kemudian sistem akan memindahkan saldo dari pengguna ke akun "admin sistem blockchain" sebagai bukti dan *record* pengguna telah melakukan transaksi. Dengan ini pada sistem dapat mudah diketahui bahwa yang melakukan transaksi tidak hanya *admin enterprise* saja (walaupun yang melakukan penandatanganan adalah pengguna anggota non admin) dan juga bukan anggota grup saja tanpa diketahui bilamana transaksi adalah transaksi *enterprise*.

```
let transferAdminAssetToMemberHash = ""

if (accountType === 2) {

   const transferAdminAssetToMember = await
transferAsset(transferAdminAssetToMemberBody)
   const transferAdminAssetToMemberStatus =
transferAdminAssetToMember.status
   const transferAdminAssetToMemberMessage =
transferAdminAssetToMember.message

   if(transferAdminAssetToMemberStatus !== 200) return next({
```

```
message: transferAdminAssetToMemberMessage })

   transferAdminAssetToMemberHash =
transferAdminAssetToMember.info
}
```

Kode 4.54 Perpindahan saldo dari admin group ke membernya.

```
const transferAdminAssetToMemberBody = {
   privateKey: adminId.key,
   srcAccountId: adminIrohaId,
   destAccountId: userIrohaId,
   assetId: orgAssetId,
   amount: 1,
   description: "Add funds from admin to member."
}
```

Kode 4.55 Objek body request pada fungsi "transferAdminAssetToMember"

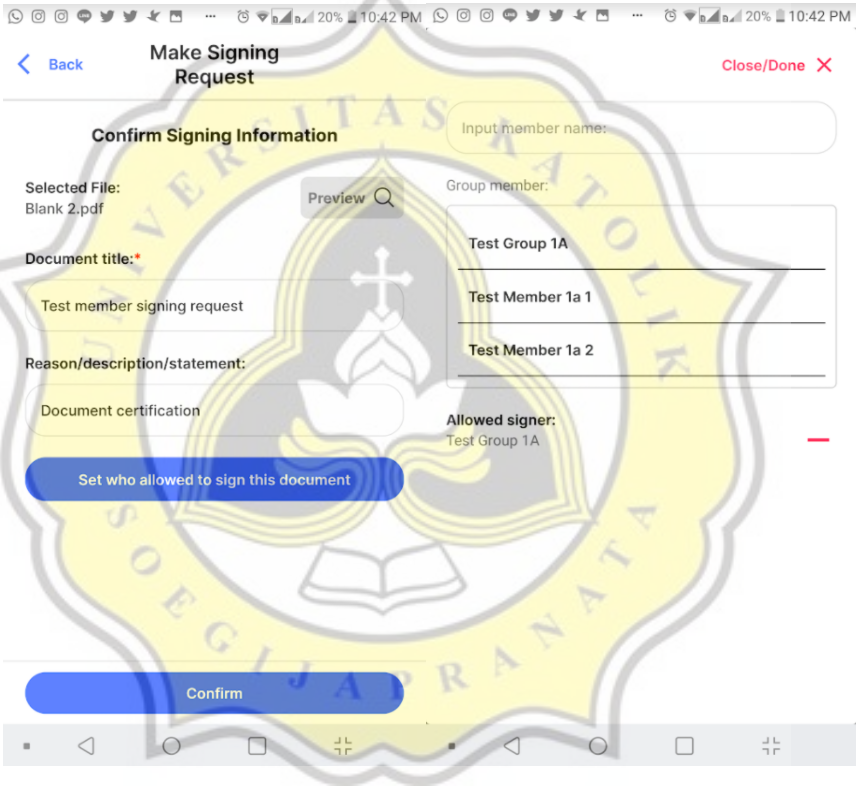
Kemudian apabila proses sertifikasi dan pencatatan pada sistem *blockchain* telah dilakukan, seperti biasa dilakukan juga pencatatan ke dalam *database*. Hanya saja pada transaksi *enterprise* terdapat perbedaan kode status dari dokumen yang telah ditandatangani, yaitu 2 apabila dokumen membutuhkan persetujuan admin (tahap proposal), 3 apabila proses penandatanganan sedang berjalan dan belum selesai, dan 4 apabila proses penandatanganan telah selesai. Status dapat disesuaikan sesuai dengan kebutuhan dan *use-case* yang diinginkan.

Apabila aplikasi *front-end* telah menerima respon sukses dari API, maka pengguna yang terakhir melakukan penandatanganan dapat mengunduh apabila *pop-up* "berhasil" telah muncul pada aplikasi. Pengguna dapat menekan tombol "Download PDF" untuk mengunduh, atau menekan "Continue" untuk kembali ke halaman utama. Dokumen yang telah selesai ditandatangani dan disertifikasi akan muncul pada list pada *tab* "Finished" di menu "My Signing Request".

4.1.6 Penggunaan layanan enterprise (group) signing pada sisi enterprise member

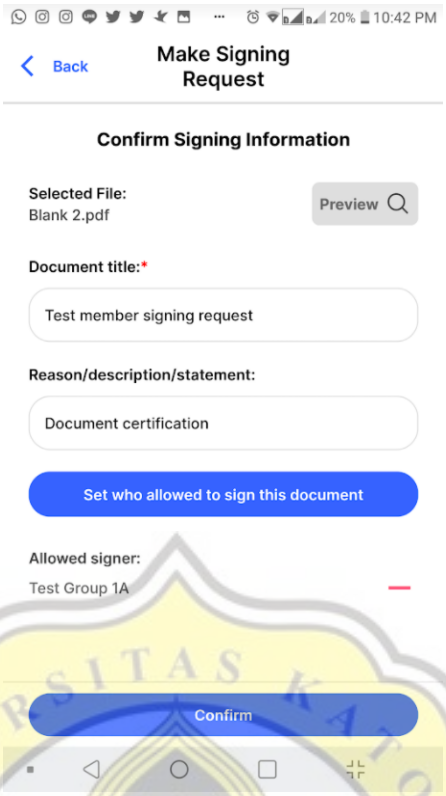
Pada sisi pengguna, pengguna tidak diharuskan mempunyai saldo terlebih dahulu, karena seluruh proses penandatanganan biayanya ditanggung oleh saldo admin. Untuk dapat melakukan penandatanganan dokumen, pengguna harus membuat *request/proposal* terlebih dahulu kepada enterprise admin.

Caranya adalah dengan menekan menu “Sign and Certify PDF” pada halaman utama, kemudian akan muncul *drawer* pilih “Make signing request”. Kemudian pengguna akan diminta untuk memasukan dokumen yang hendak ditandatangani. Apabila sudah, pengguna dapat menekan tombol “Next” untuk melanjutkan ke proses selanjutnya. Selanjutnya pengguna diarahkan pada halaman konfirmasi, di mana pengguna harus menuliskan judul dokumen, deskripsi dokumen, dan juga pengguna wajib memasukan siapa saja yang dapat menandatangani dokumen dengan menekan tombol “Set who allowed to sign this document”, kemudian akan muncul *pop-up* di mana pengguna harus memilih siapa saja (anggota dari grup) yang dapat melakukan penandatanganan terhadap dokumen seperti pada gambar 4.36.



Gambar 4.36 Halaman konfirmasi pada saat pembuatan “Signing Request” (proposal).

Apabila sudah, pengguna dapat menekan tombol “Close/done” untuk menutup pop-up. Maka kemudian pengguna dapat menekan tombol “Confirm” untuk mengirim semua informasi ke *endpoint* “/pdfentrygrp1st”. Kemudian akan dilakukan verifikasi SMS OTP terlebih dahulu pada aplikasi *front-end*, baru kemudian apabila verifikasi telah sukses proses pengiriman data *request* (*proposal*) ke API akan dilakukan.



Gambar 4.37 Halaman konfirmasi pembuatan proposal setelah semua data siap dikirimkan.

Pada sisi API yang terjadi adalah pengecekan bilamana telah terdapat file PDF yang sebelumnya pernah di *upload* atau diproses pada sistem. Apabila belum ada data, input baru, apabila sudah ada mengeset status dokumen sebelumnya menjadi "belum selesai". Apabila sudah ada dan proses penandatanganan telah selesai sebelumnya, maka akan mengembalikan pesan error. Pengecekan tersebut dilakukan seperti pada kode 4.56.

```
const inputRequest = async () => {
    return await PdfModel.findOne(
        { "pdf_unsigned.hash": pdf_unsigned_hash },
        { status: true }
    )
    .then(async (result) => {
        if (!result) return await inputNew()
        if (result.status === 5) return await inputUpdate()

        return next(conflicted("Found already signed PDF data."))
    })
    .catch(next)
}
```

Kode 4.56 Validasi terhadap dokumen proposal yang dikirimkan.

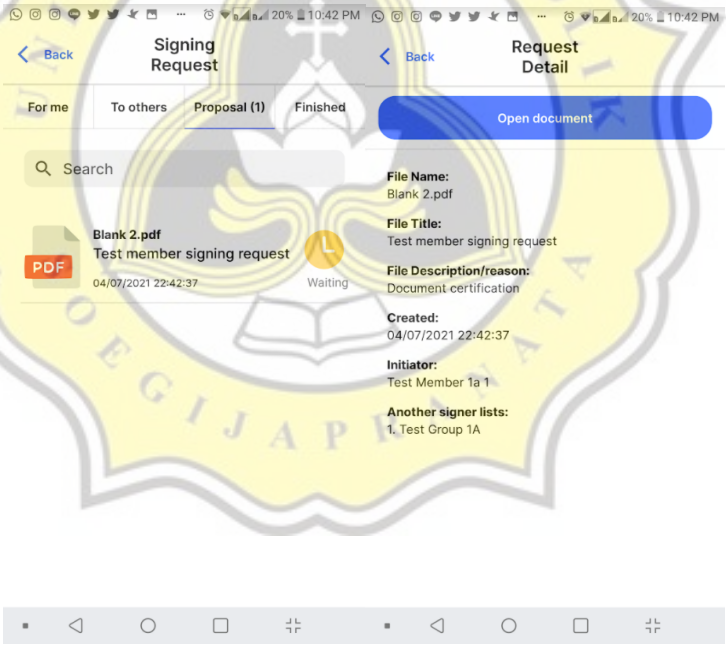
```
const inputNew = async () => {
    return await PdfModel.create({
        pdf_unsigned,
        fileName,
        fileTitle,
        fileDesc,
        owner_id,
        allow_sign,
        status,
        created,
        modified: [modified],
        deleted
    })
    .then(() => {
        return res.status(200).json({
            message: "PDF input succesfully, waiting admin to
approve.",
            info: {}
        })
    })
    .catch(next)
}
```

Kode 4.57 Fungsi apabila dokumen tidak ditemukan pada sistem.

Apabila proses selesai maka akan muncul *pop-up* yang memberi tahu bahwa *signing request* (*proposal*) telah terkirim untuk dapat kemudian ditinjau oleh admin terlebih dahulu seperti pada gambar 4.38.

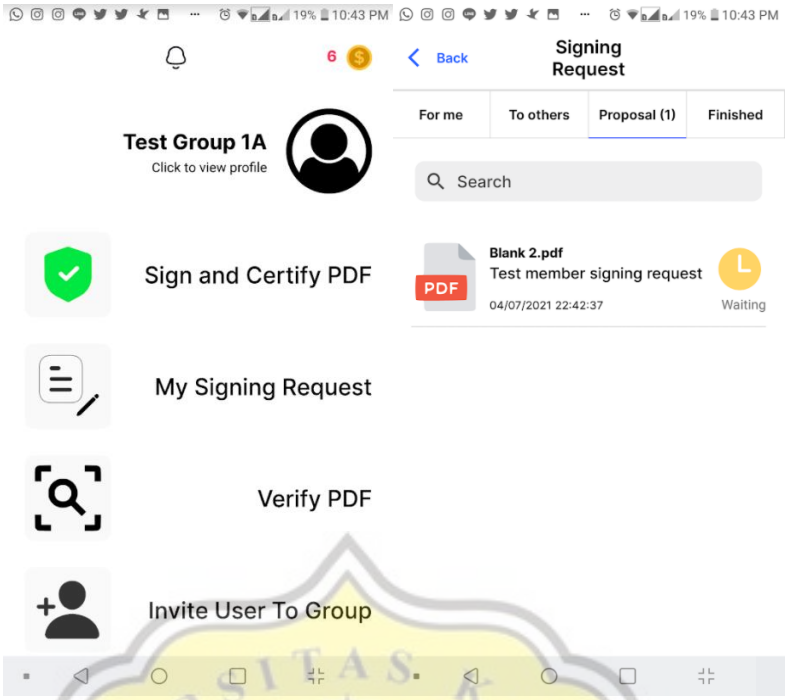
Gambar 4:38 Pop-up (modal) setelah proposal berhasil terkirim.

Request/proposal yang dikirimkan oleh anggota *enterprise* (*group*) dapat dilihat pada menu "My Signing Request" pada tab "Proposal" seperti pada gambar 4.39.



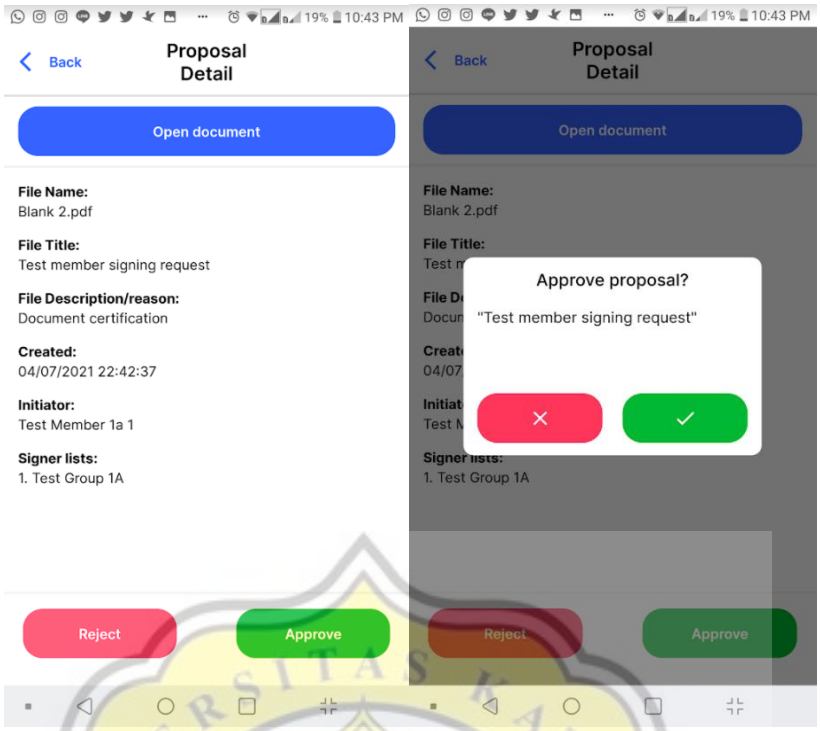
Gambar 4.39 Proposal pada halaman "Signing Request".

Setelah pengguna mengirimkan *request/proposal*, pada akun admin *enterprise* (group) dapat melihat pada menu "My Signing Request" pada *tab* "Proposal".



Gambar 4.40 Tampilan proposal pada akun admin.

Apabila *admin* memilih *request* yang masuk, admin dapat melihat informasi berkaitan dengan dokumen tersebut seperti melihat dokumen secara langsung, melihat siapa saja yang boleh menandatangani, apa judul dan deskripsi dokumen, kapan dokumen dibuat, dan sebagainya. Kemudian *admin* harus menekan dua pilihan untuk menolak atau menerima *proposal/request* dari anggotanya. Apabila *admin* menerima, maka seluruh *request* untuk penandatanganan akan diteruskan ke seluruh *list* pengguna yang bersangkutan (yang dipilih untuk menandatangani dokumen tersebut). Proses tersebut dapat dilihat pada gambar 4.41.

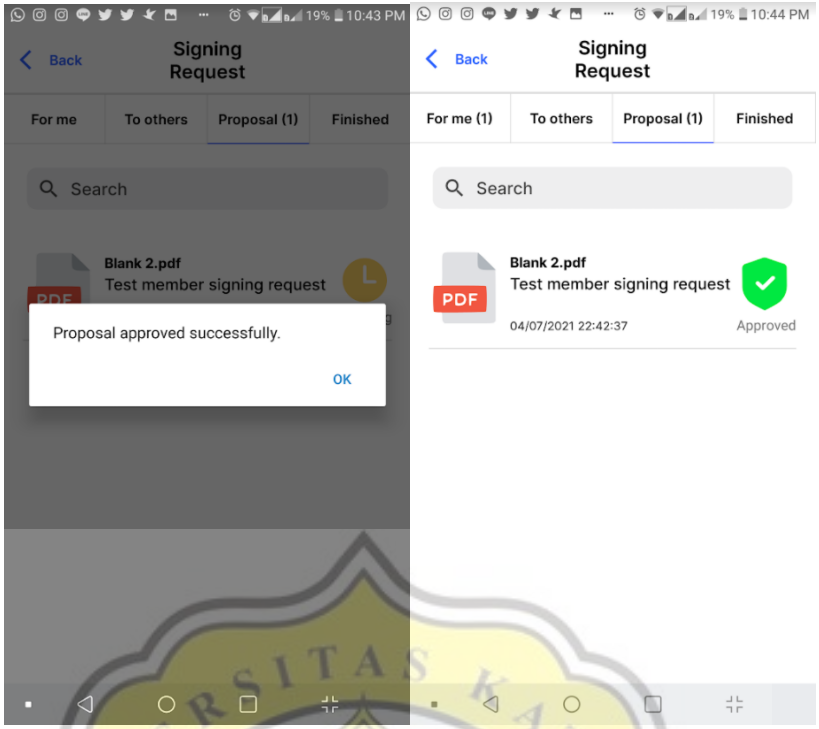


Gambar 4.41 Admin enterprise melakukan konfirmasi terhadap proposal.

Pada API, apabila melakukan perubahan status hanya sesederhana merubah nilai pada kolom status dokumen seperti pada kode 4.58.

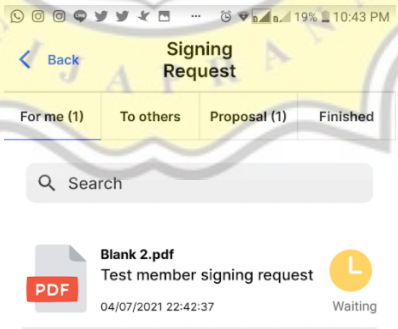
```
const changeStatus = async () => {
   if (reject)
      return await PdfModel.updateOne(
         { "_id": id },
         { $set: { status: 5 } }
      )

   return await PdfModel.updateOne(
      { "_id": id },
      { $set: { status: 3 } }
   )
}
```

Kode 4.58 Melakukan perubahan status berdasarkan keputusan admin enterprise.



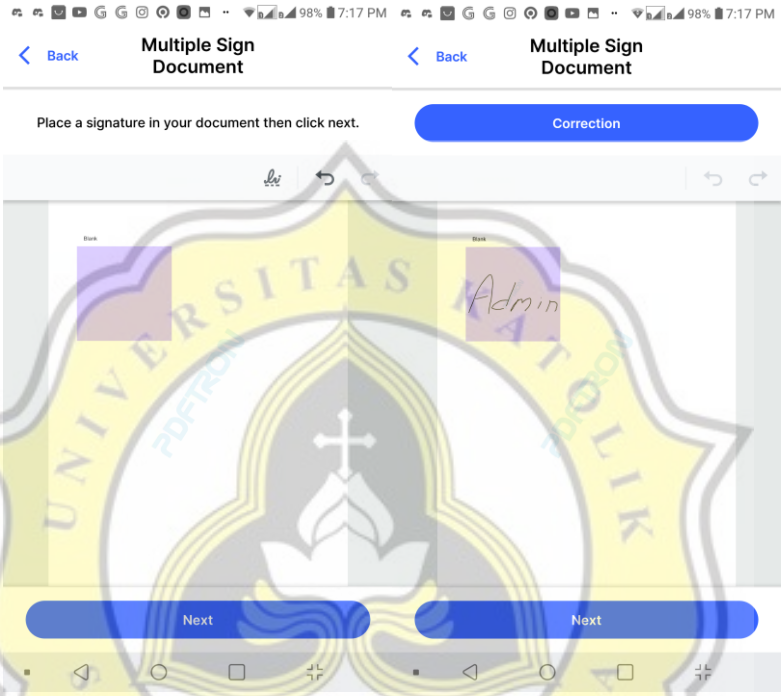
Gambar 4.42 Proposal telah disetujui oleh admin enterprise.

Apabila admin telah melakukan penerimaan izin untuk melakukan penandatanganan terhadap dokumen yang bersangkutan, pengguna yang bersangkutan dapat melihat pada menu "My Signing Request" pada tab "For me" seperti pada gambar 4.43, kemudian menekan dokumen tersebut untuk kemudian melakukan proses penandatanganan dengan memilih dokumen yang hendak ia tandatangani dengan status "Waiting".



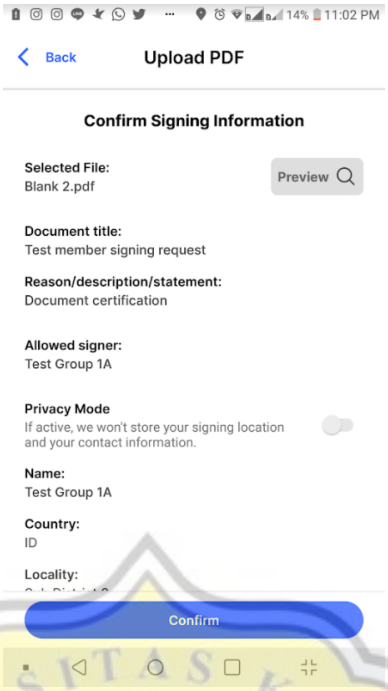
Gambar 4.43 Pengguna yang diminta untuk menandatangani menerima request.

Proses penandatanganan berlangsung seperti sebelumnya yaitu dengan menyematkan tanda tangan, kemudian mengirimkan informasinya ke sistem sampai semua pengguna yang dipilih menandatangani dokumen. Apabila semua penandatangan telah menandatangani dokumen tersebut, maka pengguna yang terakhir menandatangani dapat mengunduh dokumen tersebut, dan pengguna lainnya dapat mengunduh dokumen tersebut pada menu "My Signing Request" pada tab "Finished".



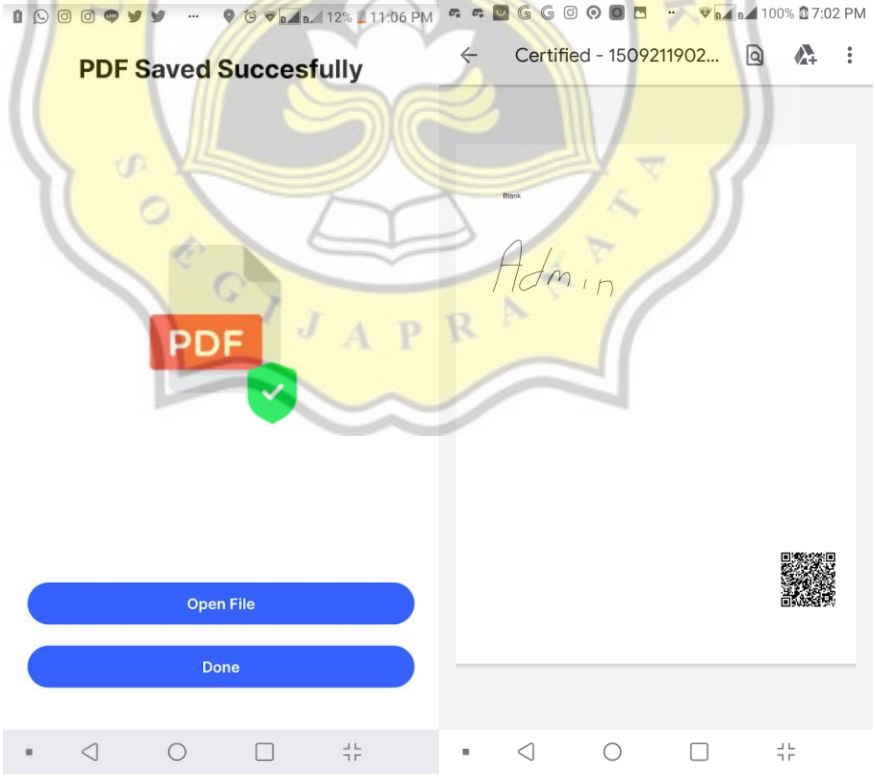
Gambar 4.44 Menandatangani dokumen.

Berikut adalah tampilan halaman konfirmasi pada penandatangan "non-initiator" dapat dilihat seperti pada gambar 4.45 bahwa seluruh informasi yang diperlukan telah terisi kecuali pengaturan *privacy mode* yang masih dapat diatur.



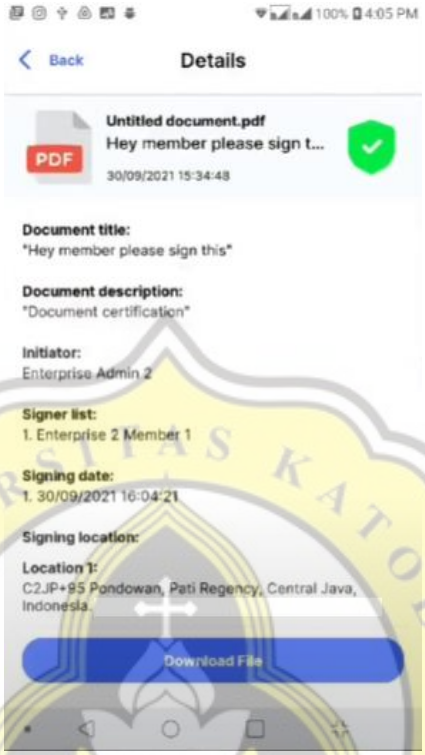
Gambar 4.45 Tampilan halaman konfirmasi pada penandatangan "non-initiator".

Berikut adalah hasil apabila proses penandatanganan telah terlewati dengan sukses dapat dilihat pada gambar 4.46.



Gambar 4.46 Hasil dari proses penandatanganan.

Apabila dokumen diverifikasi, maka pada detailnya sedikit berbeda dengan penandatanganan yang dilakukan oleh individual. Yaitu terdapat detail "*initiator*" yang menunjukan bahwa dokumen ini pertama kali dikirimkan oleh pengguna yang bersangkutan (namanya tertulis sebagai *initiator*) seperti pada gambar 4.47.

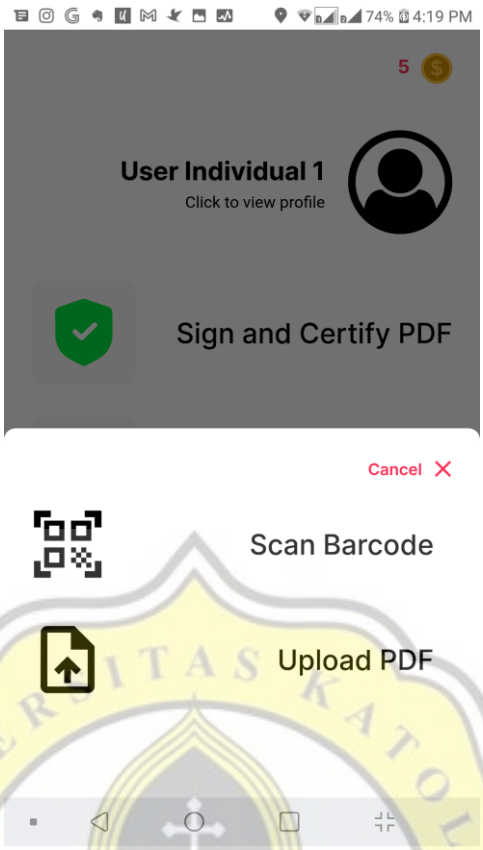


Gambar 4.47 Detail dokumen di menu "My Signing Request" pada tab "Finished".

Alur keuangan (*cash flow*) *enterprise* (*group*) juga berlaku pada transaksi ini. Yaitu pada setiap pengguna yang bukan *admin* dalam melakukan pencatatan transaksi, sebelumnya akan menerima saldo terlebih dahulu dari admin, kemudian pengguna tersebut akan mengirimkan transaksinya beserta dengan saldo sejumlah yang ia terima ke akun "admin sistem blockchain" sebagai *record* pencatatan transaksi yang telah dilakukan oleh masing-masing pengguna.

4.1.7 Verifikasi Transaksi Penandatanganan

Untuk dapat melakukan verifikasi terhadap dokumen, pengguna dapat masuk ke halaman utama dengan memilih menu "Verify PDF" seperti pada gambar 4.48.



Gambar 4.48 Mengakses menu "Verify PDF" pada aplikasi client.

Proses verifikasi transaksi dapat dilakukan dengan beberapa cara.

Yang pertama adalah dengan cara mencari data *hash* dokumen yang belum ditandatangani pada *database*, kemudian mengambil *hash* transaksi yang dihasilkan saat melakukan transaksi melalui Hyperledger Iroha kemudian mencari transaksi tersebut melalui API *query* Hyperledger Iroha (misal "getTransactions" atau "getAccountAssetTransactions"). Cara ini cukup efektif apabila hendak membuat sistem verifikasi terhadap bukti transaksi saja, dan untuk melakukannya pengguna tidak perlu mengunggah *file* yang hendak diverifikasi karena pencarian data pada *database* dilakukan dengan memberikan data berupa *hash string* dari dokumen yang belum ditandatangani, yang mana data tersebut dapat dikemas ke dalam QR Code. Kelemahannya adalah tidak dimungkinkannya untuk melakukan verifikasi terhadap keaslian dari dokumen secara langsung (mengecek ke dalam dokumen secara langsung). Prosesnya dapat dilihat pada gambar 4.54 dan detail dari hasil verifikasinya seperti pada gambar 4.55.

Cara yang kedua, adalah dengan mengunggah dokumen secara langsung baru kemudian dilakukan pengecekan seperti pada gambar 4.49. Cara ini mempunyai

kelebihan apabila seseorang hendak melakukan verifikasi secara lebih detail dan lebih *valid*, karena dengan adanya *file* dokumen yang diunggah dapat dilakukan pengecekan secara lebih lanjut seperti mengecek apakah dokumen tersebut mempunyai tanda tangan *digital* yang terpasang dengan bantuan PDFTron, dan apakah terdapat data dokumen tersebut pada *database*. Bahkan juga dimungkinkan untuk menampilkan data misalkan pengguna mengunggah dokumen yang tidak ditandatangani melalui sistem yang dibuat tetapi mempunya data tanda tangan *digital* dari *platform* lain, atau bahkan juga memunculkan data transaksi apabila mengunggah *file* dokumen sebelum ditandatangani (terdapat data file dokumen setelah ditanda tangani) seperti dapat dilihat hasilnya pada gambar 4.53. Kelebihannya adalah proses verifikasi tidak hanya dilakukan terhadap transaksi, melainkan juga terhadap *file* dokumen itu sendiri apakah ditemukan pada *database* atau tidak.

```
const pdfId = req.body.pdfId
const pdfB64 = req.body.pdfB64

const findHashByTxHash = async ({ txHashesList }) => {

   const result = await getTransactions({ txHashesList })
   if (result.status === 500) {
      const latestHash = txHashesList[txHashesList.length - 1]
      const resultJustLatestHashes = await getTransactions({
txHashesList: [latestHash] })
      if (resultJustLatestHashes.status === 500) return false
      return true
   }
   return true
}

if (pdfB64) {

   buffer = base64ToBuffer(pdfB64)
   hasSignature = await CheckHasSignature(buffer)
   pdfHash = await hashBase64(pdfB64)
} else {
   pdfHash = hexDecode(pdfId)
}

await Db().catch(next)
```

```
const pdfData = await getPdfData(pdfHash).catch(next)

if (!pdfData && !hasSignature) return next(notFound("This
document does not have any signature."))

if (!pdfData && hasSignature) {

   const signatureInfo = await
PrintSignaturesInfo(buffer).catch(next)
   return res.status(200).json({
      message: "This document has signature, but it's not signed
in this platform.",
      info: signatureInfo
   })
}

const txHashesList = pdf_certified.map(item => item.txHash)
const bcVerifyResult = await findHashByTxHash({ txHashesList })

if (pdfB64 && pdfData && !hasSignature) return
res.status(200).json({
   message: "This document has been signed before.",
   info: {
      pdfData: pdfDataMutated,
      signerData,
      bcVerifyResult,
      txHashesList
   }
})

return res.status(200).json({
   message: "Success getting data.",
   info: {
      pdfData: pdfDataMutated,
      signerData,
      bcVerifyResult,
      txHashesList
   }
})
```

Kode 4.59 Proses verifikasi dengan dua metode.

```
const getTransactions = async ({ txHashesList }) => {
```

```
    if (!txHashesList) return badRequest()

    return await queries.getTransactions(queryOptions, {
txHashesList })
    .then(info => {

      return {
        status: 200,
        message: "Success getting transaction by hashes.",
        info
      }
    })
    .catch(e => { return fatalError(e) })
}
```

Kode 4.60 Fungsi yang menangani query "getTransactions" ke Hyperledger Iroha.

Alternatif cara lain untuk menemukan transaksi yang telah sukses dilakukan pada Hyperledger Iroha selain menggunakan *query* "getTransactions", dapat juga dilakukan dengan memanfaatkan *query* "getAccountAssetTransactions". Namun terdapat beberapa usaha lebih yang harus dilakukan untuk mendapatkan transaksi yang diinginkan dengan melakukan pemecahan dari objek respon yang diberikan oleh Hyperledger Iroha, melakukan *mapping* terhadap seluruh deskripsi transaksi yang telah dilakukan pengguna yang bersangkutan, kemudian mencari apakah terdapat deskripsi transaksi (*hash* dari dokumen yang telah ditandatangani) yang sama (yang hendak diverifikasikan).

```
const latestSignerAssetTransactions = await
getAccountAssetTransactions({ accountId, assetId })
const bcVerifyCode = latestSignerAssetTransactions.status
let transactionsList = null
let reducedPayload = null
let commandsList = null
let txDesc = null
let bcVerifyResult = null

if (bcVerifyCode === 200) {
   transactionsList =
latestSignerAssetTransactions.info[0].transactionsList
   reducedPayload = transactionsList.map((item, index) => {
```

```
        return item.payload.reducedPayload
    })
}

if (reducedPayload) {
    commandsList = reducedPayload.map((item, index) => {
        return item.commandsList[0].transferAsset
    })
}

if (commandsList) {
    txDesc = commandsList.map((item, index) => {
        return item.description
    })
}

if (txDesc) {
    bcVerifyResult = txDesc.includes(latestHash)
}
```

Kode 4.61 Melakukan verifikasi transaksi dengan query "getAccountAssetTransactions".

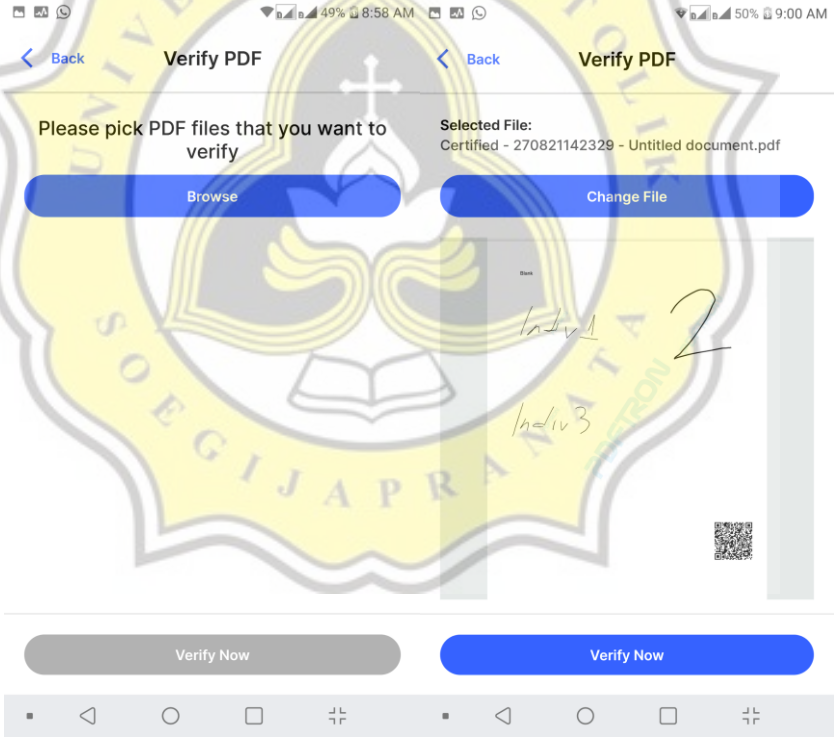
Pada kode di atas, apabila transaksi ditemukan pada akun yang bersangkutan maka variabel "bcVerifyResult" akan berisi dengan nilai *boolean true* yang berarti bahwa deskripsi transaksi (*hash* dari PDF yang telah ditandatangani) ditemukan pada daftar transaksi yang telah dilakukan oleh akun yang bersangkutan.

```
const getAccountAssetTransactions = async ({
    accountId,
    assetId
}) => {

    if (!accountId || !assetId) return badRequest()

    const pageSize = 999

    return await
queries.getAccountAssetTransactions(queryOptions, {
        accountId,
        assetId,
        pageSize,
        firstTxHash: null,
```

```
      ordering: {}
   })
   .then(info => {
      return {
         status: 200,
         message: "Success getting asset history of '" +
accountId + "'.",
         info
      }
   })
   .catch(e => { return fatalError(e) })
}
```

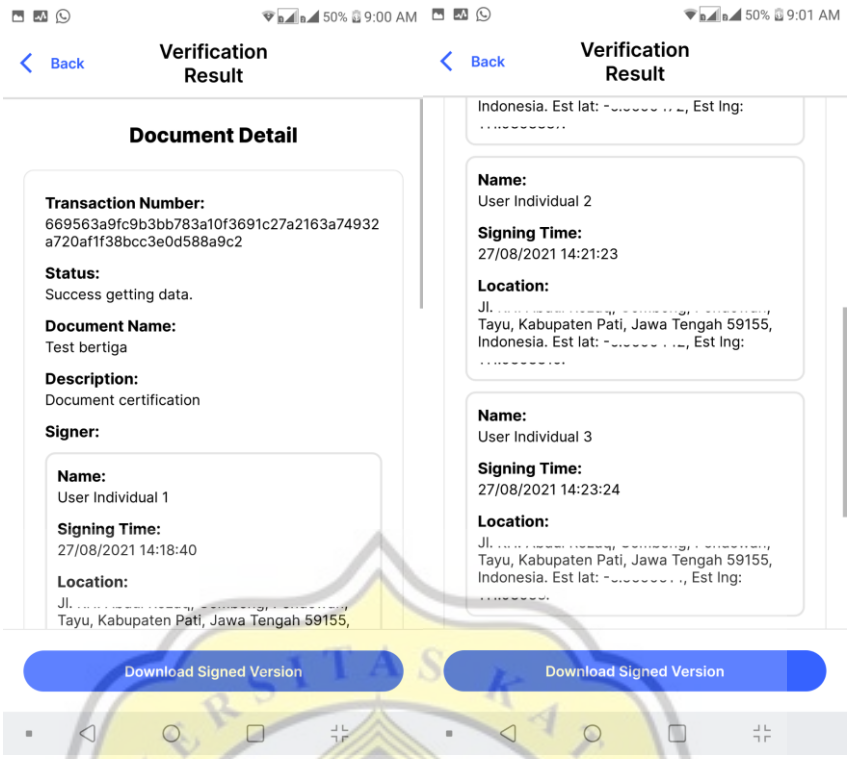
Kode 4.62 Query "getAccountTransactions" ke Hyperledger Iroha.

Berikut pada gambar 4.49 adalah contoh gambar melakukan verifikasi terhadap dokumen melalui *upload file* dokumen yang telah melewati proses penandatanganan.

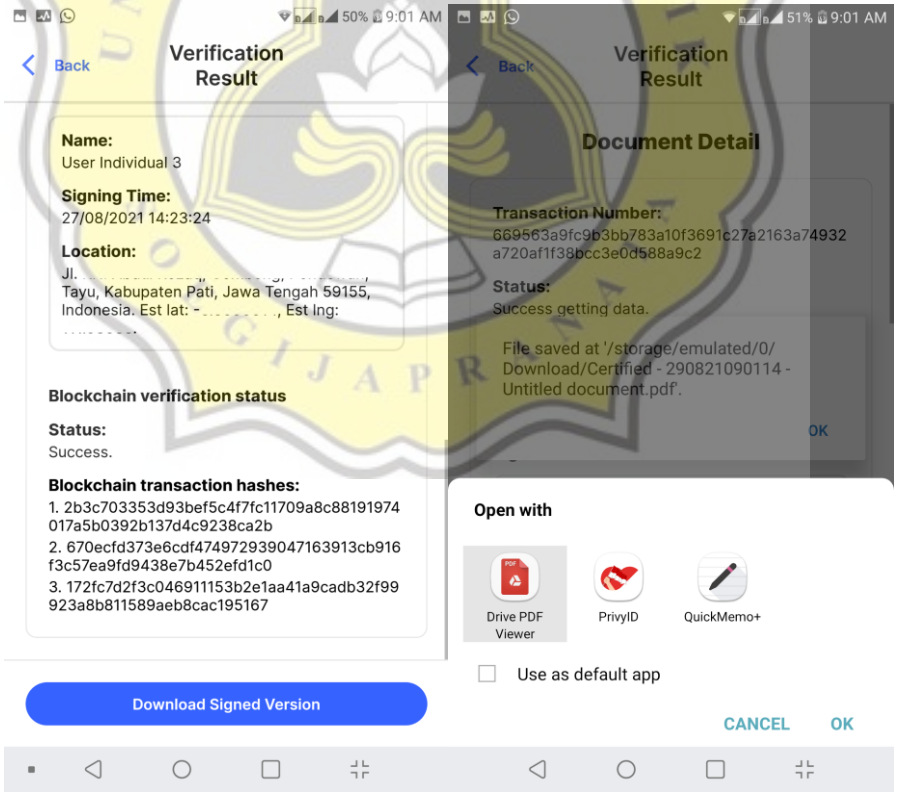


Gambar 4.49 Verifikasi manual melalui file.

Berikut adalah hasil dari melakukan verifikasi melalui *upload file*. Pada *upload file* dimungkinkan juga untuk melakukan *download file* dokumen *final* dari proses penandatanganan seperti pada gambar 4.50 dan 4.51.

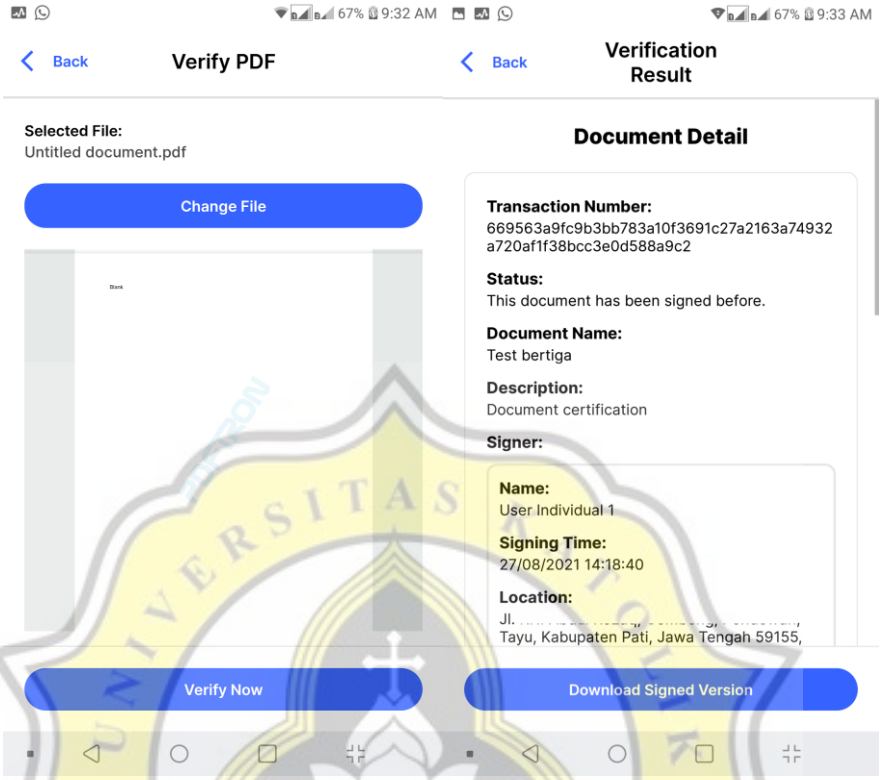


Gambar 4.50 Detail verifikasi dokumen melalui upload.



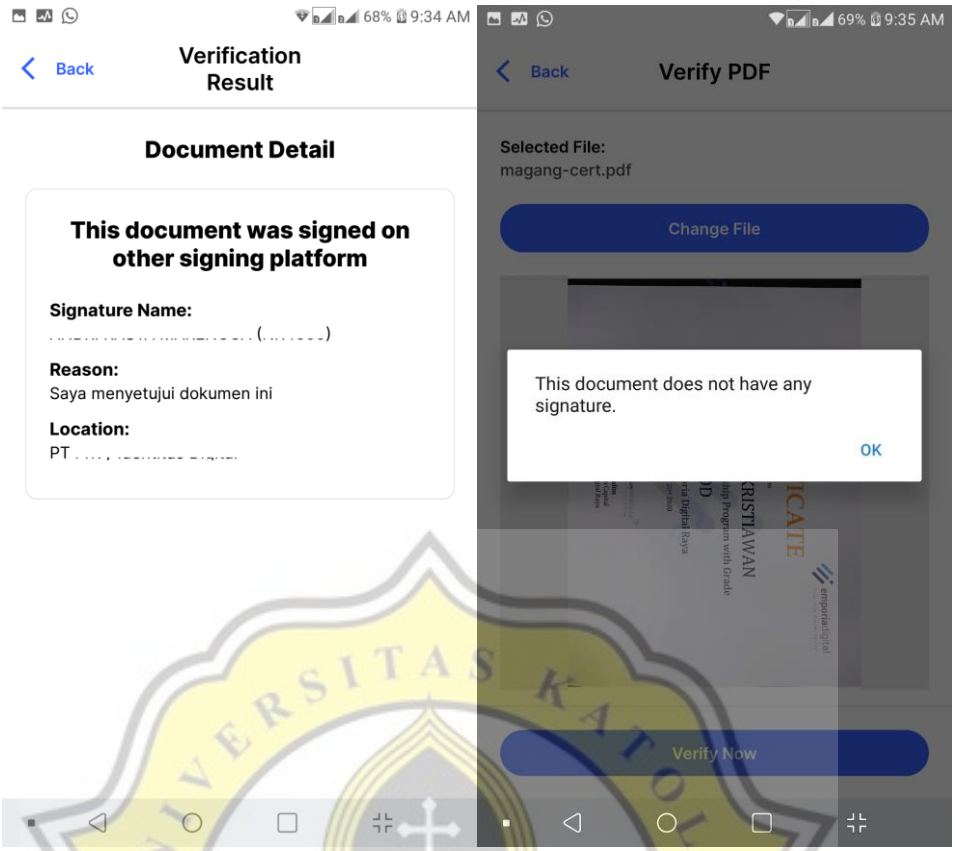
Gambar 4.51 Menyimpan dokumen yang telah melewati proses penandatanganan.

Melalui verifikasi upload file dimungkinkan juga untuk mengupload dokumen asli “sebelum melewati proses penandatanganan” yang hasilnya dapat dilihat pada gambar 4.52.



Gambar 4.52 Verifikasi dokumen asli “sebelum ditandatangani”.

Dan berikut pada gambar 4.53 adalah hasil apabila dokumen yang ditandatangani melalui platform lain dan juga tidak mempunyai tanda tangan sama sekali diverifikasikan.



Gambar 4.53 Hasil dokumen yang ditandatangani melalui platform lain, dan tanpa tanda tangan.

Berikut pada gambar 4.54 adalah contoh melakukan verifikasi menggunakan metode scan QR Code yang disematkan pada dokumen yang telah melewati proses penandatanganan.

Gambar 4.54 Verifikasi transaksi penandatanganan melalui scan QR Code.

Dan berikut pada gambar 4.55 adalah hasil dari verifikasi melalui QR Code. Berbeda dengan verifikasi melalui *file upload*, Tidak terdapat pilihan untuk mengunduh dokumen *final*.

Gambar 4.55 Detail transaksi penandatanganan melalui scan QR Code.

4.1.8 Signing Out (Log-out)

Untuk melakukan proses "signing out" yang dapat dilakukan adalah menghapus data user, *token* JWT, dan *refresh token* JWT yang tersimpan pada aplikasi *client*. *Refresh token* yang disimpan pada aplikasi tersebut juga dihapus dari *database* agar tidak dapat digunakan lagi dalam proses autentikasi pengguna. ID pengguna yang hendak melakukan sign out serta *refresh token* pengguna yang tersimpan pada aplikasi *client* dikirimkan melalui API *request* ke *endpoint* "./signout". Kemudian verifikasi terhadap *refresh token* dilakukan menggunakan *JWT Secret* yang tersimpan pada sistem *back-end*.

Berikut adalah proses sign-out pada aplikasi front-end dapat dilihat pada kode 4.63.

```
const getRefreshToken = async () => {
```

```
    let refreshToken = state.user.refreshToken

    if (!refreshToken) refreshToken = await
getItem("refreshToken")

    return refreshToken
}

const signOut = async () => {

    dispatch({ type: "IS_LOADING" })

    const refreshToken = await getRefreshToken()
    const body = { id }

    return await request.post("signout", body, refreshToken)
    .catch(throwError)
    .finally(async () => {
      await setItem("token", "")
      await setItem("refreshToken", "")
      await setItem("currentUser", "")
      dispatch({ type: "DELETE_SESSION" })
      dispatch({ type: "NOT_LOADING" })
      return navigation.dispatch(
        CommonActions.reset({
          index: 1,
          routes: [
            { name: "HomeUnauthScreen" }
          ],
        })
      )
    })
}
```

Kode 4.63 Fungsi yang digunakan saat melakukan sign out pada aplikasi client.

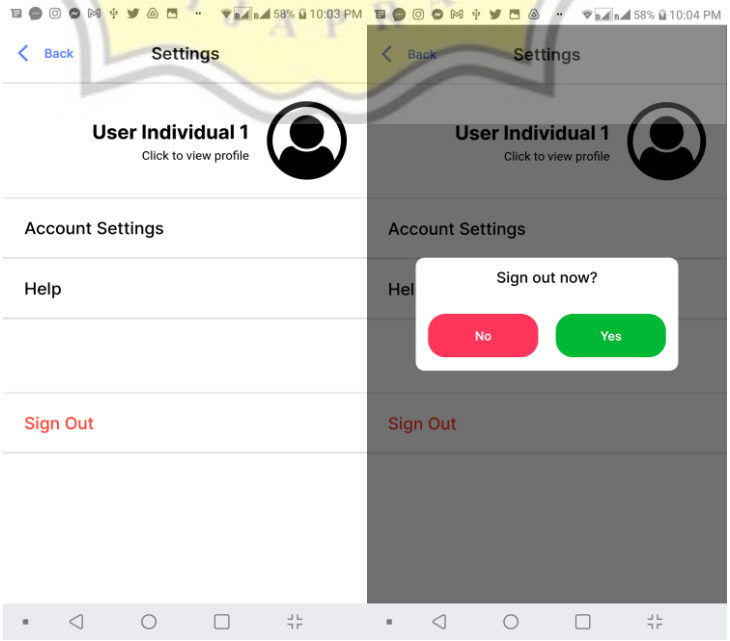
Dan berikut adalah proses sign-out pada aplikasi back-end dapat dilihat pada kode 4.64.

```
const signOut = router.post('/signout', async (req, res, next)
=> {

    const authHeader = req.headers['authorization']
```

```
    const id = req.body.id
    let _id = ""

    if (!authHeader) return next(badRequest())

    const token = authHeader.split(" ")[1]

    if (!id) {
        jwt.verify(token, JWT_SECRET_2, (err, tokenDecoded) => {
            if (err) return next(badRequest())
            _id = mongoose.Types.ObjectId(tokenDecoded.id)
        })
    } else {
        _id = mongoose.Types.ObjectId(id)
    }

    await removeToken(_id, token).catch(next)

    return res.status(200).json({
        message: "User signed out successfully.",
        info: {}
    })
})
```

Kode 4.64 Proses pada endpoint "./signout" pada aplikasi backend.

Berikut pada gambar 4.56 adalah tampilan apabila pada aplikasi front-end di menu setting tombol "Sign Out" ditekan.



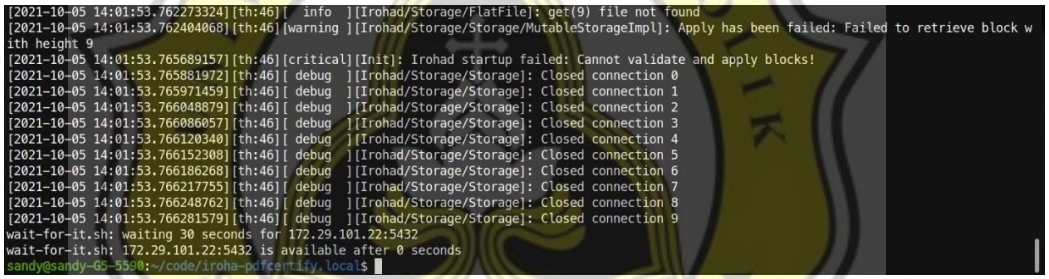
Gambar 4.56 Melakukan sign out pada aplikasi client.

4.4 Pengujian keamanan Hyperledger Iroha

Pada penelitian ini dilakukan juga pengujian keamanan terhadap Hyperledger Iroha sebagai arsitektur *blockchain* yang digunakan dalam pengerjaan ini. Hal ini untuk mengetahui bagaimana Hyperledger Iroha versi 1.2.1 dapat menangani apabila terjadi serangan terhadap salah satu atau beberapa *peer*.

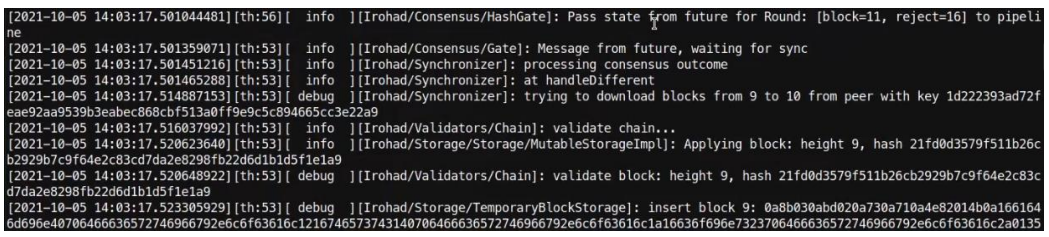
4.4.1 Simulasi apabila sebuah peer kehilangan block dengan posisi urutan block yang tidak berurutan

Apabila sebuah *peer* kehilangan *block* dengan posisi *block* yang hilang tersebut adalah tidak terurut, pada saat *peer* tersebut menerima *block* baru, *peer* tersebut masih dapat menerima *block* yang baru. Tetapi apabila pada saat *peer* tersebut dimulai ulang (*restart*), maka *peer* tersebut tidak akan dapat berjalan seperti yang diharapkan (terjadi *error*) seperti pada gambar 4.57. Agar *peer* tersebut dapat berfungsi dengan baik kembali, kondisi *block* pada direktori "blockstore" harus diurutkan kembali.



Gambar 4.57 Tampilan log ketika error karena peer tidak berurutan.

Maka apabila *block* pada direktori "blockstore" sudah dalam kondisi berurutan, akan terjadi proses sinkronisasi pada peer tersebut dengan melakukan *download* block yang hilang dari *peer* yang masih mempunyai *block* paling lengkap seperti dapat dilihat pada gambar 4.58. Maka kondisi *block* dari *peer* yang kehilangan *block* tersebut dapat menjadi normal kembali.



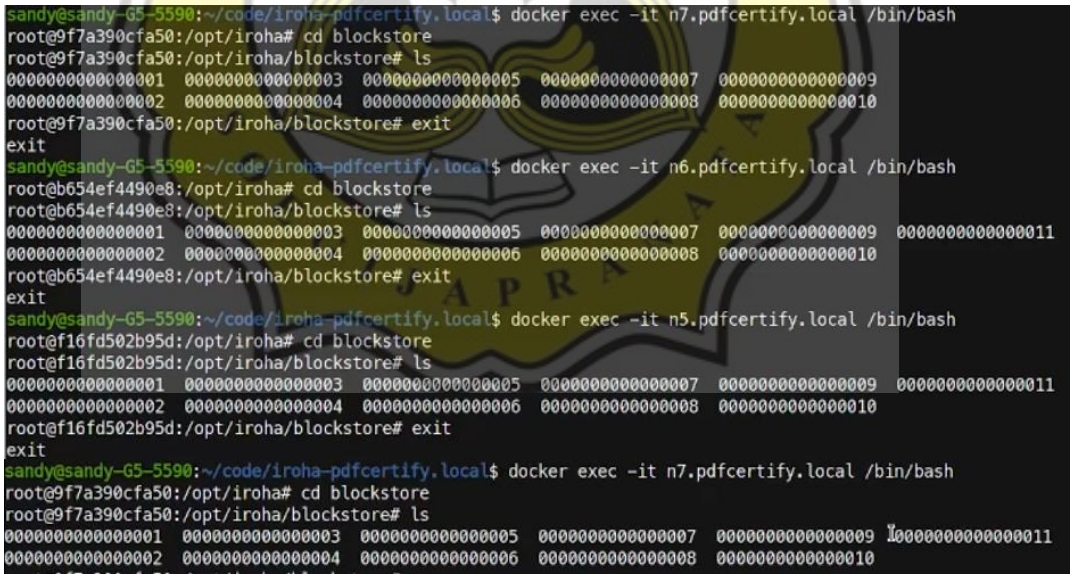
Gambar 4.58 Tampilan log saat melakukan sinkronisasi blockstore.

4.4.2 Simulasi apabila salah satu block dalam direktori “blockstore” sebuah peer diubah

Apabila pada sebuah block store yang ada pada *peer* dihapus salah satu *block*, Hyperledger Iroha tidak langsung melakukan pembenahan pada *peer* tersebut. Aktivitas sinkronisasi akan terjadi apabila peer tersebut di-restart. Hasil dari aktivitas sinkronisasi tersebut adalah *peer* yang kehilangan *block* akan mendapatkan *block* yang hilang kembali pada direktori “blockstore” *peer* tersebut.

4.4.3 Simulasi apabila terdapat beberapa peer yang mempunyai kondisi direktori “blockstore” yang berbeda

Apabila perubahan dilakukan ke hampir semua *peer*, maka saat *peer* dimulai ulang proses sinkronisasi *block* pada masing-masing *peer* akan dilakukan dengan mengambil sejumlah *block* dari *peer* yang masih mempunyai jumlah *block* lengkap dan berurutan. Apabila perubahan terjadi di banyak *peer*, maka proses sinkronisasi akan memakan cukup banyak waktu (mungkin juga tergantung dengan spesifikasi komputer) agar semua *block* pada *peer* dapat dalam keadaan sama pada setiap peer seperti seharusnya.



Gambar 4.59 Kondisi blockstore saat pertama kali dijalankan dan sedang dalam proses sinkronisasi.

Pada gambar di atas dapat dilihat bahwa peer nomor 7 saat pertama kali diakses belum mendapatkan block urutan ke 11 tetapi pada akhirnya saat proses sinkronisasi pada peer tersebut selesai maka block urutan ke 11 telah kembali.

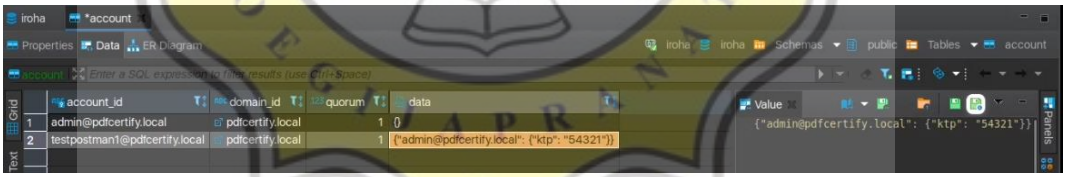
4.4.4 Simulasi apabila semua data pada "blockstore" peer dalam jaringan blockchain disamakan kondisinya

Untuk melakukan simulasi "*worst case scenario"* terhadap *blockstore* pada masing-masing *peer*, dilakukan percobaan untuk menghapus satu *block* yang sama dalam *block store* pada masing-masing *peer* yang ada pada jaringan *blockchain*. Hasilnya karena kondisi semua *block* pada direktori "*blockstore"* di seluruh *peer* adalah sama, maka kondisi yang didapatkan adalah seperti tidak terjadi apa-apa, dan akibatnya *block* yang hilang tersebut telah hilang selamanya. Dan saat melakukan query pada setiap peer yang ada di sistem terhadap block dengan nilai "height" yang hilang atau dengan hash transaksi yang hilang tidak akan mendapatkan hasil.

4.4.5 Simulasi perubahan data pada *"World State View"*

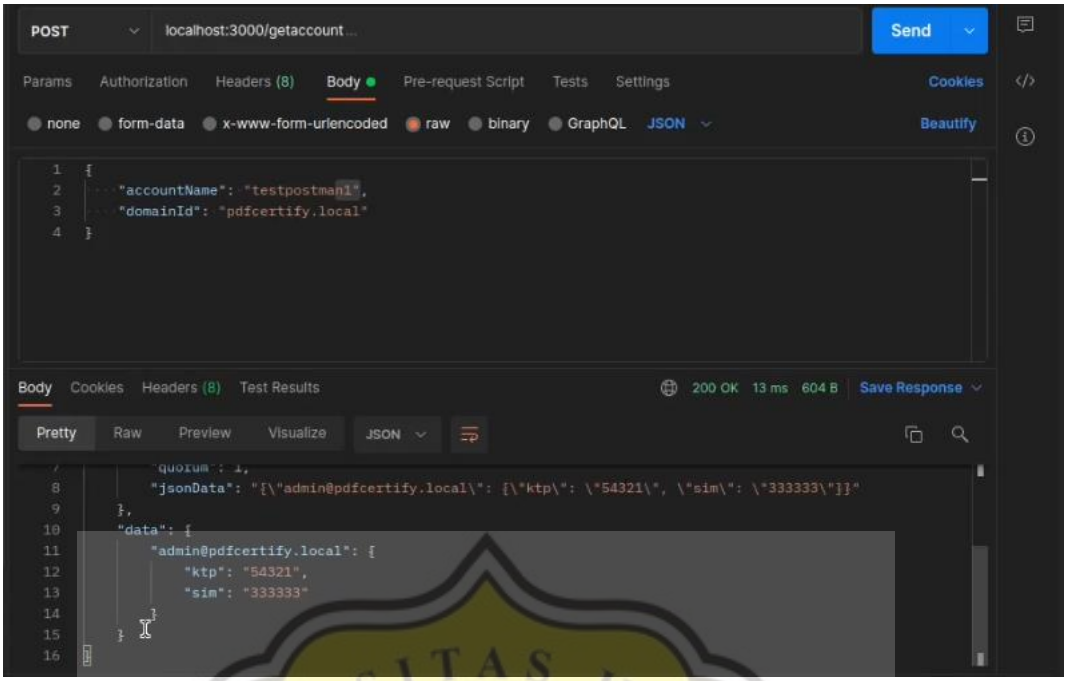
Pada simulasi ini, diasumsikan bahwa *attacker* telah mendapat akses ke *database*. Percobaan pertama adalah dengan melakukan perubahan "account details" pada sebuah akun pada sistem *blockchain* melalui *World State View* (WSV) atau pada Hyperledger Iroha adalah *database* pada masing-masing peer. Hasilnya adalah data tersebut berubah pada saat diakses melalui aktivitas query "Get Account Details".

Berikut pada gambar 4.60, data akun detail "ktp" dirubah yang mana sebelumnya adalah "12345" menjadi "54321". Perubahan hanya dilakukan pada salah satu WSV di sebuah *peer*.



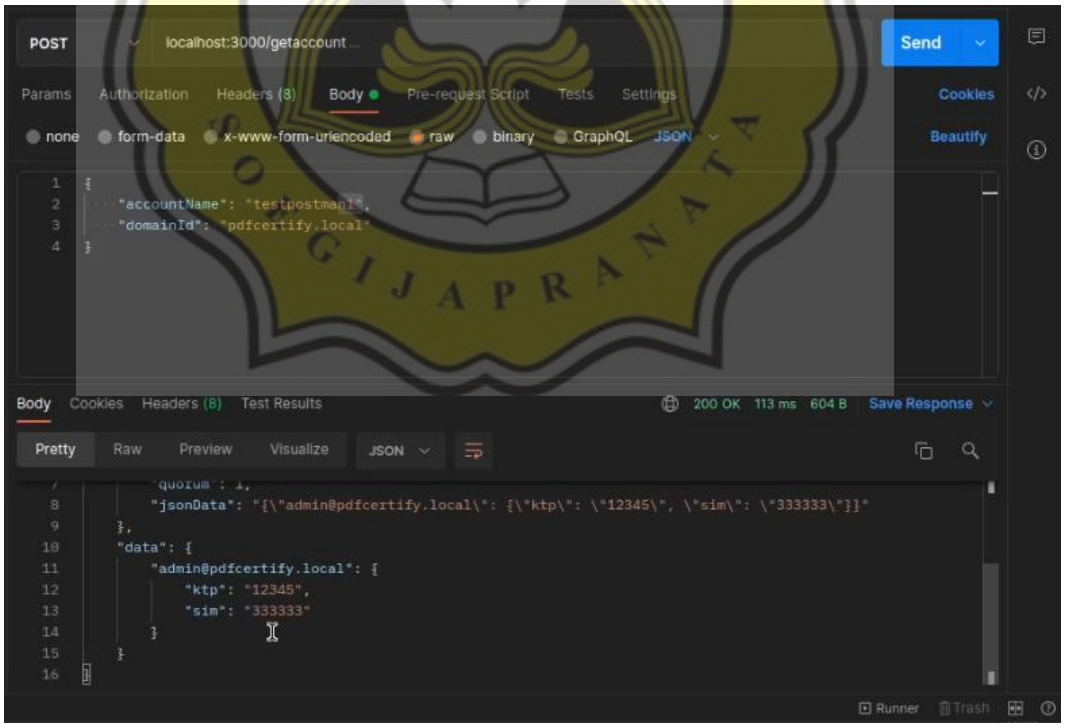
Gambar 4.60 Mengganti data detail akun pada WSV salah satu peer.

Dan berikut adalah hasil apabila *query* dilakukan terhadap *peer* yang dirubah tersebut. Dapat dilihat pada gambar 4.61 data data "ktp" adalah "54321" sesuai dengan perubahan yang dilakukan.



Gambar 4.61 Hasil query "getAccount" ke peer yang bermasalah (data WSV diubah secara paksa).

Dan berikut pada gambar 4.62 adalah hasil apabila *query* dilakukan terhadap *peer* yang belum menerima "serangan" dalam bentuk perubahan data. Dapat dilihat bahwa *field* "ktp" masih bernilaikan dengan informasi semula yaitu "12345".



Gambar 4.62 Hasil query "getAccount" ke peer yang data WSV tidak diubah secara paksa.

Percobaan kedua adalah *worst case scenario* apabila *attacker* menghapus data vital seperti data salah satu akun pada sistem *blockchain*. Maka informasi akun pada

peer tersebut sama sekali tidak dapat diakses melalui *query* dan *peer* tersebut dapat disimpulkan telah menjadi *peer* yang bermasalah (*corrupted*) dan untuk dapat mengakses data yang hilang harus menggunakan *peer* lain yang masih aman (belum bermasalah). Pada perbincangan dengan komunitas Hyperledger Iroha melalui Telegram, salah satu maintainer menjelaskan bahwa terdapat cara untuk mengatasi masalah ini yaitu pada saat WSV mengalami "*rebuild*", dan juga dijelaskan bahwa pada versi *release* selanjutnya akan terdapat penambahan fitur "*tasks*" yang dapat mengecek konsistensinya dengan *block store*.

4.5 Temuan permasalahan pada Hyperledger Iroha

Saat pertama kali menjalankan Hyperledger Iroha, terkadang apabila melakukan transaksi berupa "*commands*" akan ada *chance* mendapati kondisi di mana proses transaksi berjalan seolah lama. Tetapi apabila dilihat pada *log container* yang sedang memproses transaksi, akan ditemukan ternyata terjadi kesalahan yang mengakibatkan transaksi tidak dapat dilanjutkan (seperti pada gambar 4.63).

Gambar 4.63 Bug yang mengakibatkan peer harus dimulai ulang.

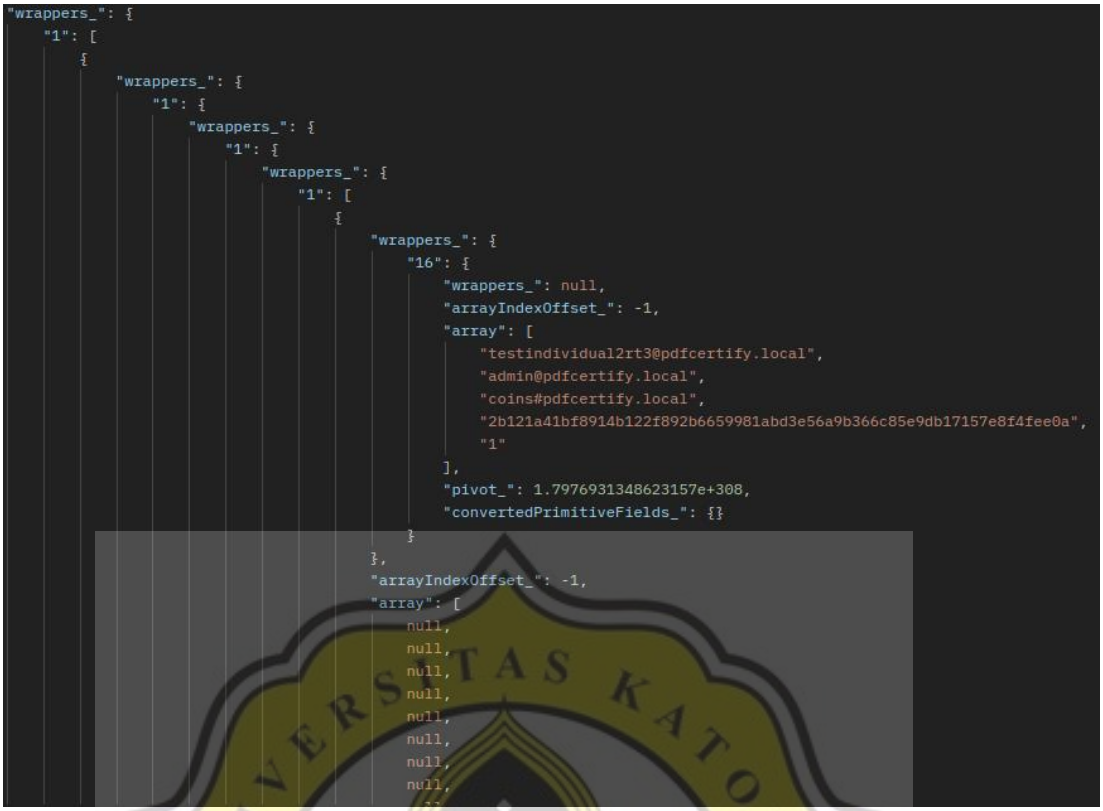
Solusi yang dapat dilakukan apabila mendapat masalah seperti gambar di atas adalah dengan cara melakukan *peer restart* pada Docker yang dapat dilakukan seperti pada kode 4.65.

```
sudo docker-compose down
sudo docker-compose up -d
```

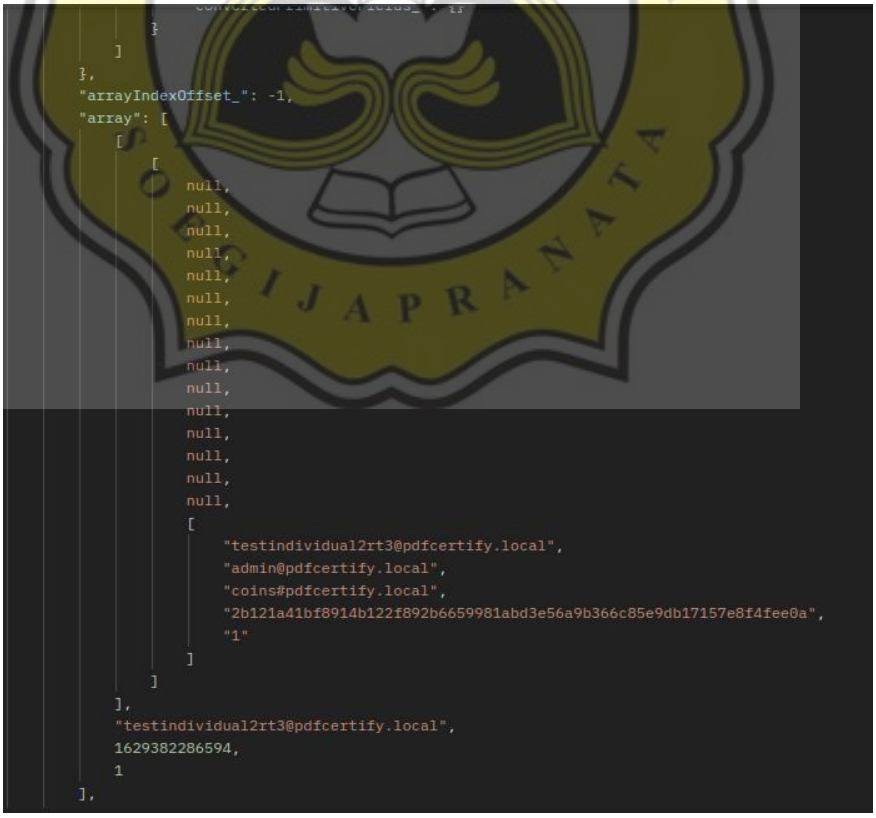
Kode 4.65 Melakukan pembuatan ulang container Hyperledger Iroha.

Selain itu, Hyperledger Iroha belum menyediakan API yang dapat memberikan hasil yang diinginkan secara mudah. Misalnya belum terdapat API untuk dapat melakukan *request query* terhadap seluruh *domain* yang ada apa sistem *blockchain*. Alhasil, apabila hendak melakukan pencarian terhadap seluruh *domain*, yang harus dilakukan adalah mengecek masing-masing block yang ada pada sistem yang mana cukup susah dilakukan dan pastinya akan memakan waktu cukup lama untuk mendapatkan hasil. Dan terkadang misalkan hendak melakukan *query* terhadap transaksi yang pernah dilakukan, misal apabila hendak mencari deskripsi transaksi seperti yang diinginkan, terdapat tiga cara yaitu dengan melakukan *query* "getTransactions" dengan memberikan *hash* transaksi yang pernah dilakukan (dengan mencatat *hash* transaksi yang pernah dilakukan ke dalam *database* kemudian menggunakannya untuk melakukan query*)*, dan pilihan kedua adalah dengan mencari masing-masing transaksi yang pernah dilakukan oleh pengguna yang bersangkutan (yang hendak dicari informasi transaksinya) melalui API "getAccountAssetTransactions" (perlu bantuan *database* untuk mencatat siapa pengguna yang melakukan transaksi tersebut). Pada API *query* "getTransactions" hasil yang diberikan secara teknis adalah format transaksi yang ada pada sistem Hyperledger Iroha yang kemudian dapat diolah kembali. Tetapi bagi pengguna yang baru saja mempelajari Hyperledger Iroha, perlu waktu untuk mempelajari format dari objek transaksi yang dikembalikan. Hal ini mungkin dapat diatasi dengan menyediakan dokumentasi *web* yang lebih baik terhadap *code specific library* Hyperledger Iroha agar pengguna baru dapat mempelajari dengan lebih mudah.

Berikut pada gambar 4.64 dan gambar 4.65 dalah hasil dari *query* "getTransactions" dengan menyertakan *hash* transaksi Hyperledger Iroha.

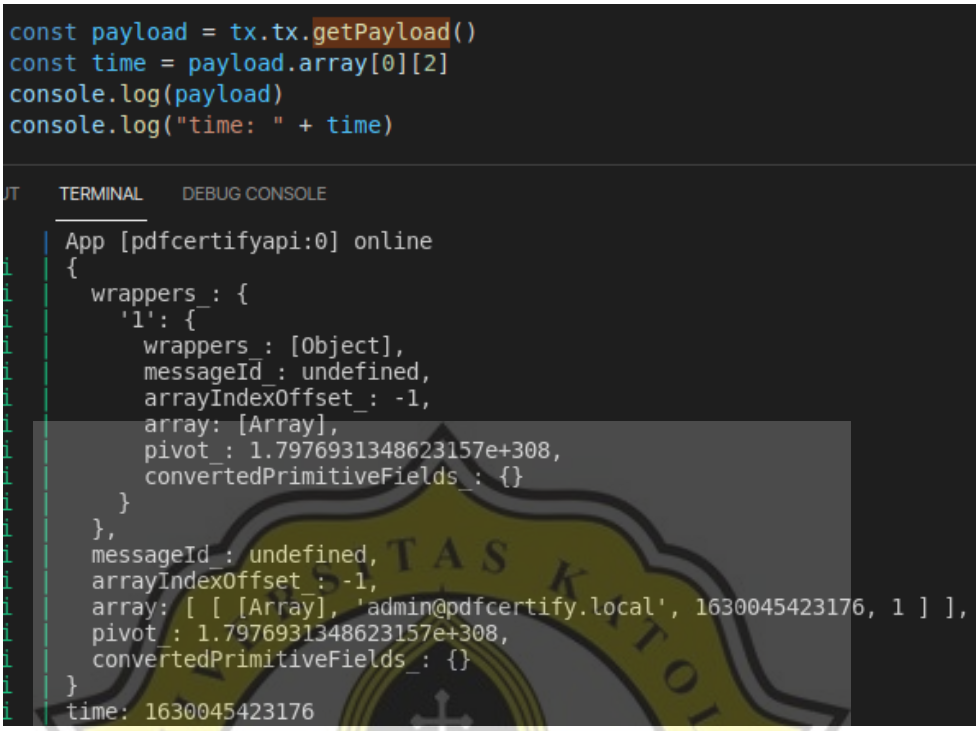


Gambar 4.64 Contoh payload yang diberikan pada API query "getTransactions".



Gambar 4.65 Lanjutan contoh payload yang masih berlanjut.

Berikut adalah contoh mendapatkan data *timestamp* pada transaksi dapat lihat pada gambar 4.66.



Gambar 4.66 Console-logging payload transaksi dan mengambil created time pada payload..

Cara yang kedua dengan memanfaatkan API *query* "getAccountAssetTransactions" memberikan hasil informasi yang lebih mudah dibaca seperti pada gambar 4.67, namun cara ini juga bukanlah pendekatan terbaik karena apabila transaksi dari akun tersebut ada banyak, maka proses yang dilakukan seharusnya menjadi lebih lambat bergantung dari sistem yang digunakan dan juga seberapa banyak transaksi yang dilakukan oleh pengguna tersebut.

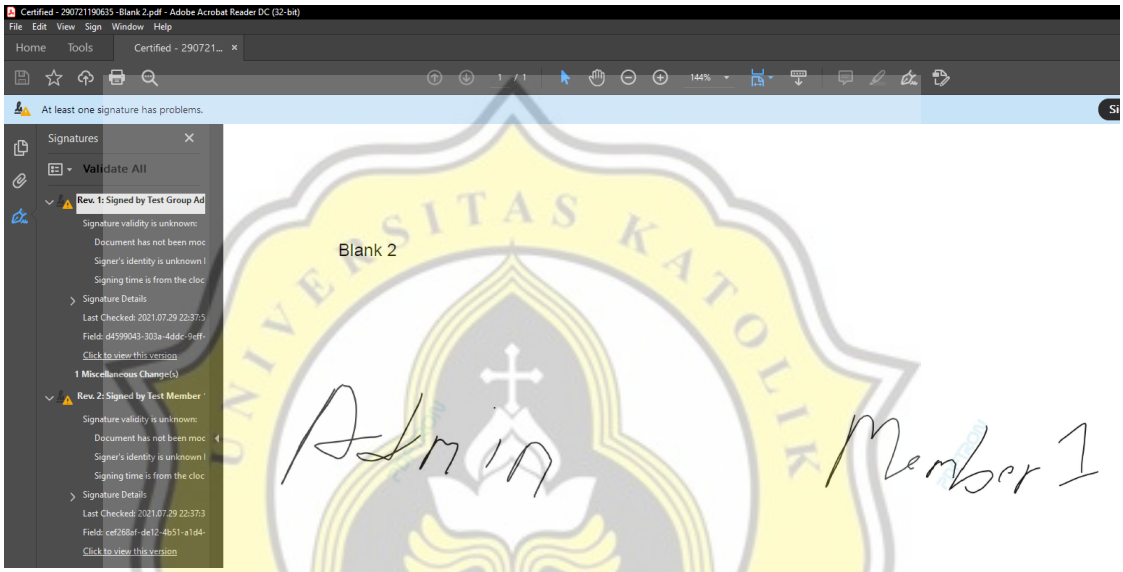
Gambar 4.67 Contoh format informasi balikan query "getAccountAssetTransactions"

Masalah selanjutnya adalah kurangnya dokumentasi pada *code specific library*. Terkadang butuh waktu untuk mengetahui bagaimana cara mengembangkan dan menggunakan Hyperledger Iroha secara lebih mendalam karena bagi *programmer* yang masih awam dan juga pemula, mereka harus mempelajari konsep dasar cara kerja Hyperledger Iroha, mempelajari apa saja yang dapat dilakukan dengan *library language specific* Hyperledger Iroha dengan cara menjelajahi dan mempelajari *code* pada direktori *library* yang mereka gunakan (misal pada JavaScript *library* apabila menggunakan Node.js, *library* dapat ditemukan pada direktori "node_modules/iroha-helpers") dan baru kemudian melakukan eksperimen sesuai dengan kebutuhan mereka. Untungnya bagi pengembang JavaScript pada *repository library* "iroha-helpers" di GitHub terdapat contoh yang cukup untuk melakukan aktivitas secara sederhana seperti cara mengirim *request command* dan *query* yang dapat ditemukan pada *file* README.MD yang dapat ditemukan dalam direktori awal *library* "iroha-helpers" dan contoh kode penggunaan pada folder "examples". Dokumentasi *code specific* pada Hyperledger Iroha diharapkan dapat dikembangkan dan dibenahi agar pengembang baru bisa dengan mudah memulai menggunakan Hyperledger Iroha dengan dasar bahasa pemrograman yang hendak digunakan untuk membangun program.

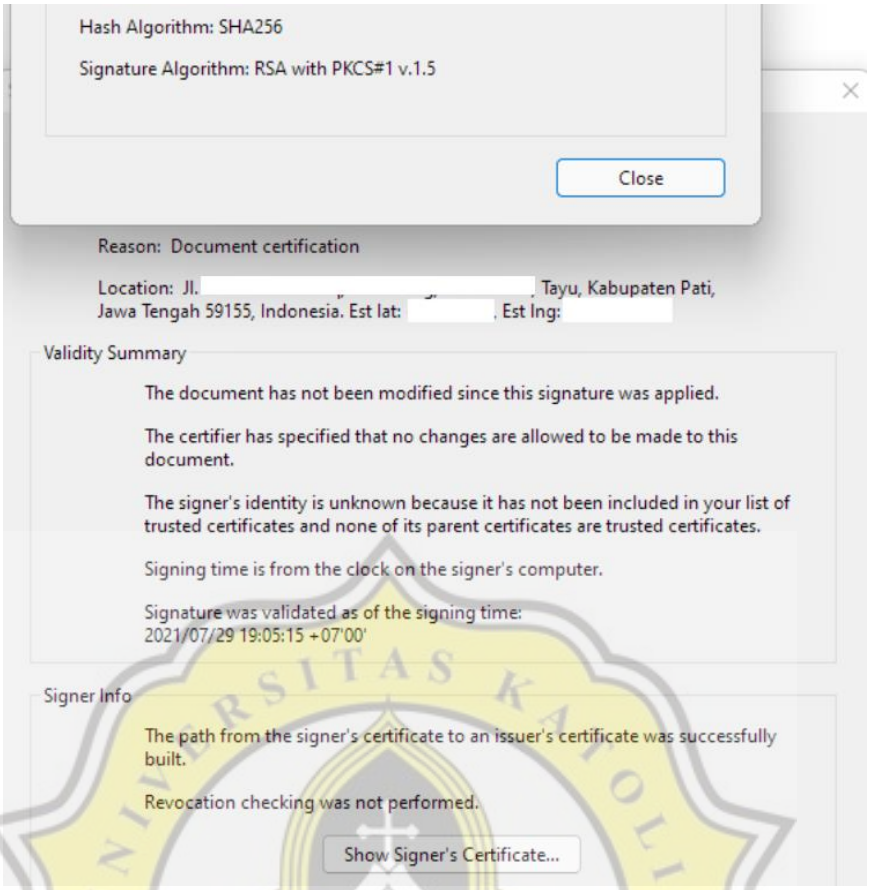
4.6 Pengujian keamanan dokumen yang telah disertifikasi

Pengujian dilakukan secara sederhana dengan melakukan perubahan pada dokumen yang telah selesai melewati proses penyematan tanda tangan beserta sertifikat *digital* dari masing-masing penandatangan.

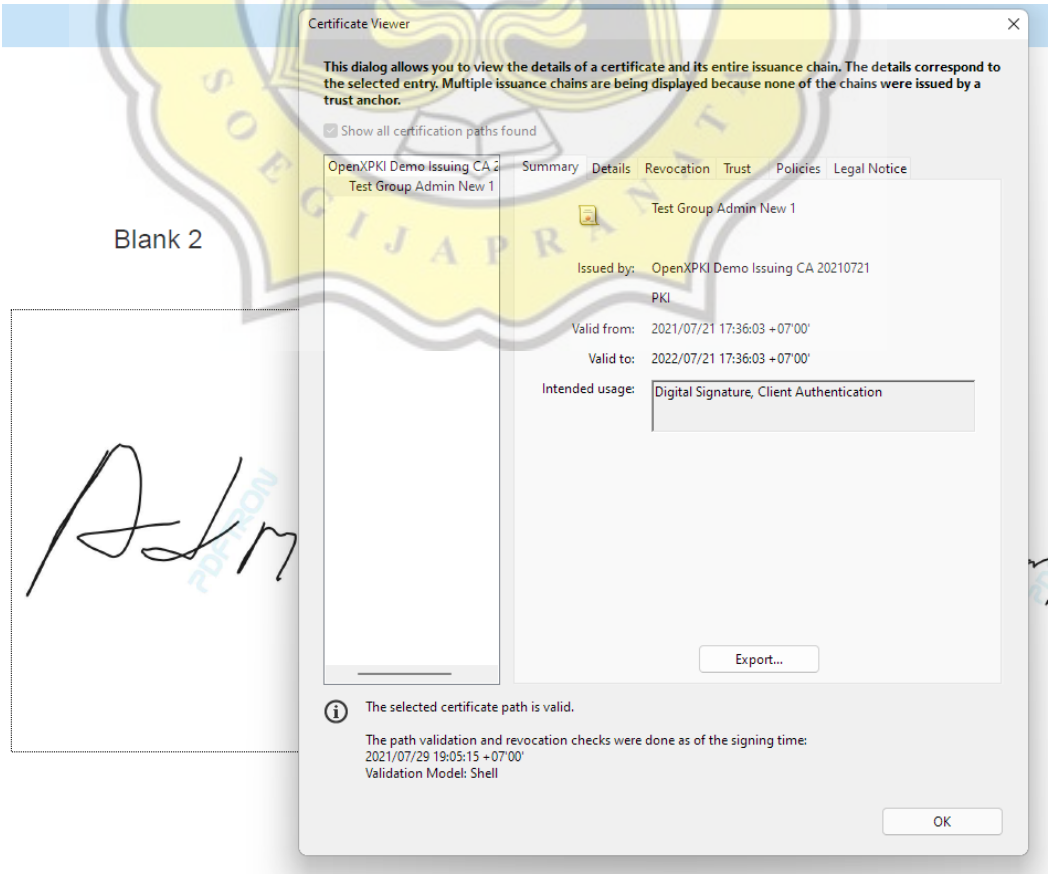
Pada dokumen yang telah melewati proses transaksi penandatanganan, dapat dilihat detail informasi dari penandatangan beserta detail lain seperti deskripsi, *location*, dan informasi lainnya seperti informasi yang berkaitan dengan sertifikat *digital* pengguna.



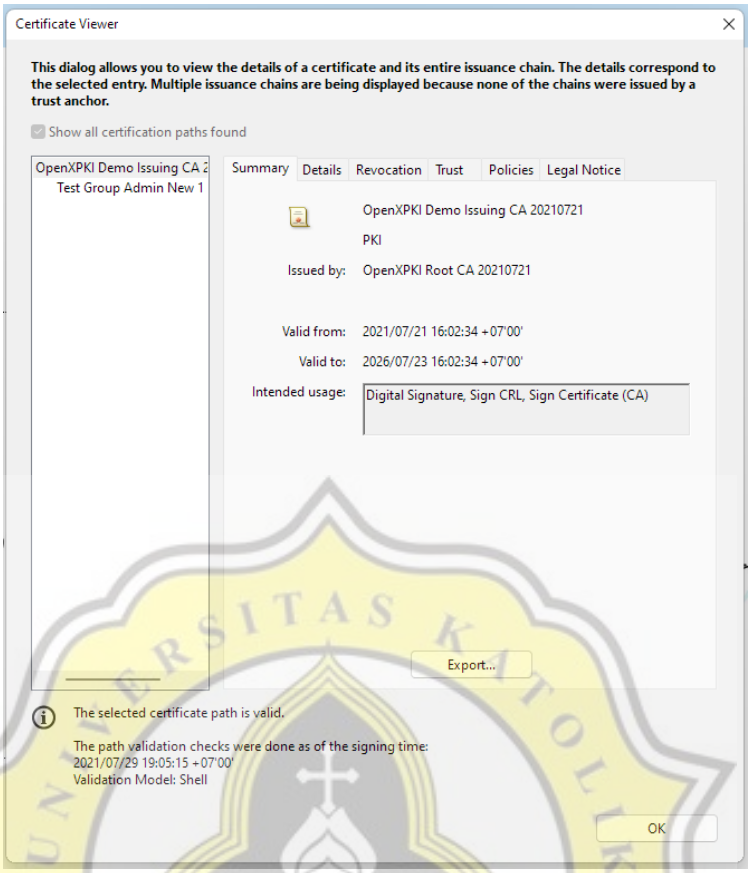
Gambar 4.68 Dokumen telah selesai melewati proses transaksi.



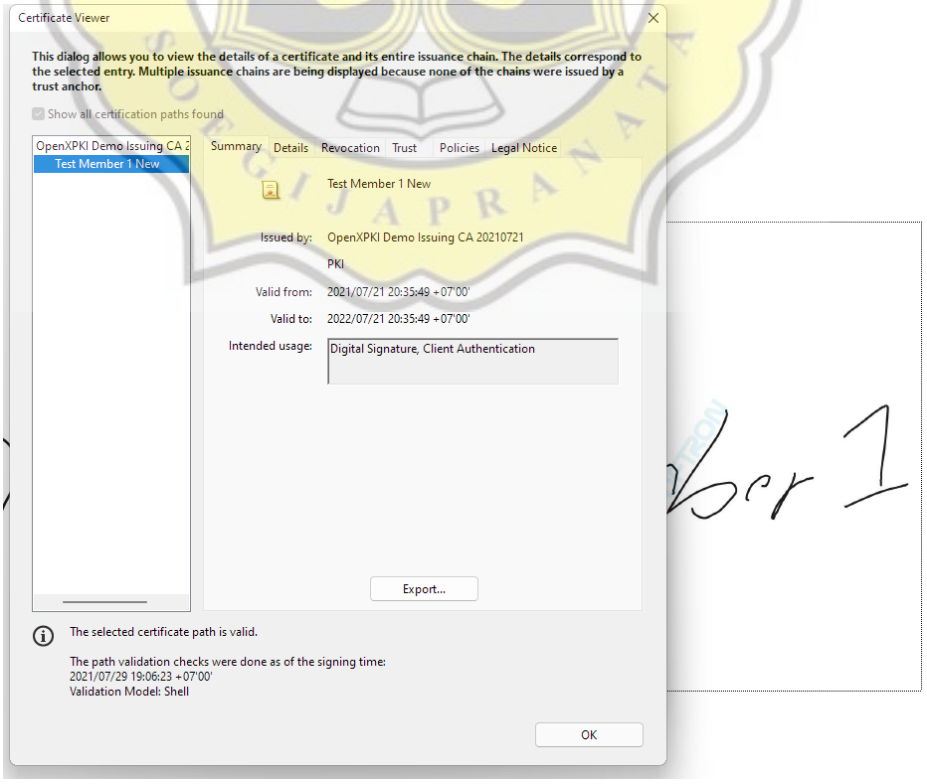
Gambar 4.69 Detail dari salah satu tanda tangan apabila diklik.



Gambar 4.70 Detail sertifikat penandatangan pertama.

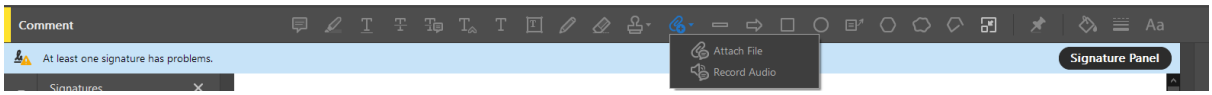


Gambar 4.71 Detail dari intermediate certificate.



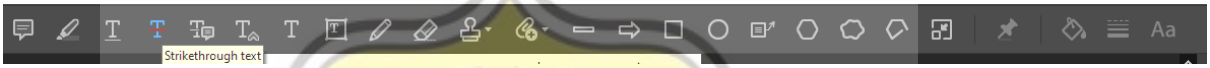
Gambar 4.72 Detail sertifikat dari penandatangan kedua.

Dokumen tersebut juga akan dikunci dengan permission “no change allowed” sehingga perubahan tidak akan dapat dilakukan pada dokumen tersebut. Hal ini mengakibatkan *annotation tools* menjadi tidak dapat difungsikan seperti pada gambar 4.73.



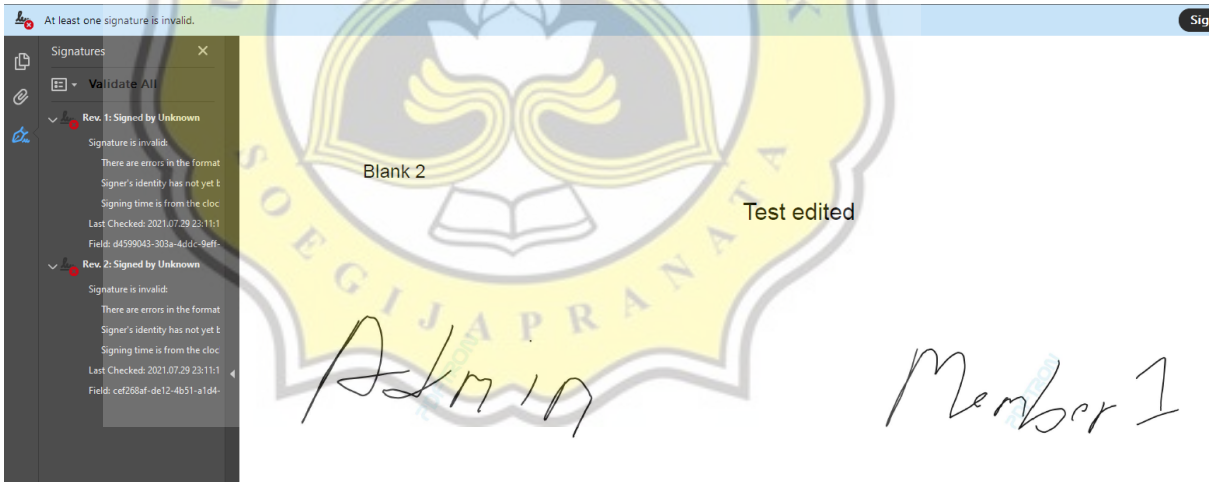
Gambar 4.73 Annotation tools menjadi tidak berfungsi karena permission.

Berbeda dengan dokumen yang tidak dikunci dengan permission apapun, fungsi *annotation* masih dapat difungsikan seperti pada gambar 4.74.



Gambar 4.74 Perbandingan dengan PDF yang tidak diamankan dengan permission sama sekali.

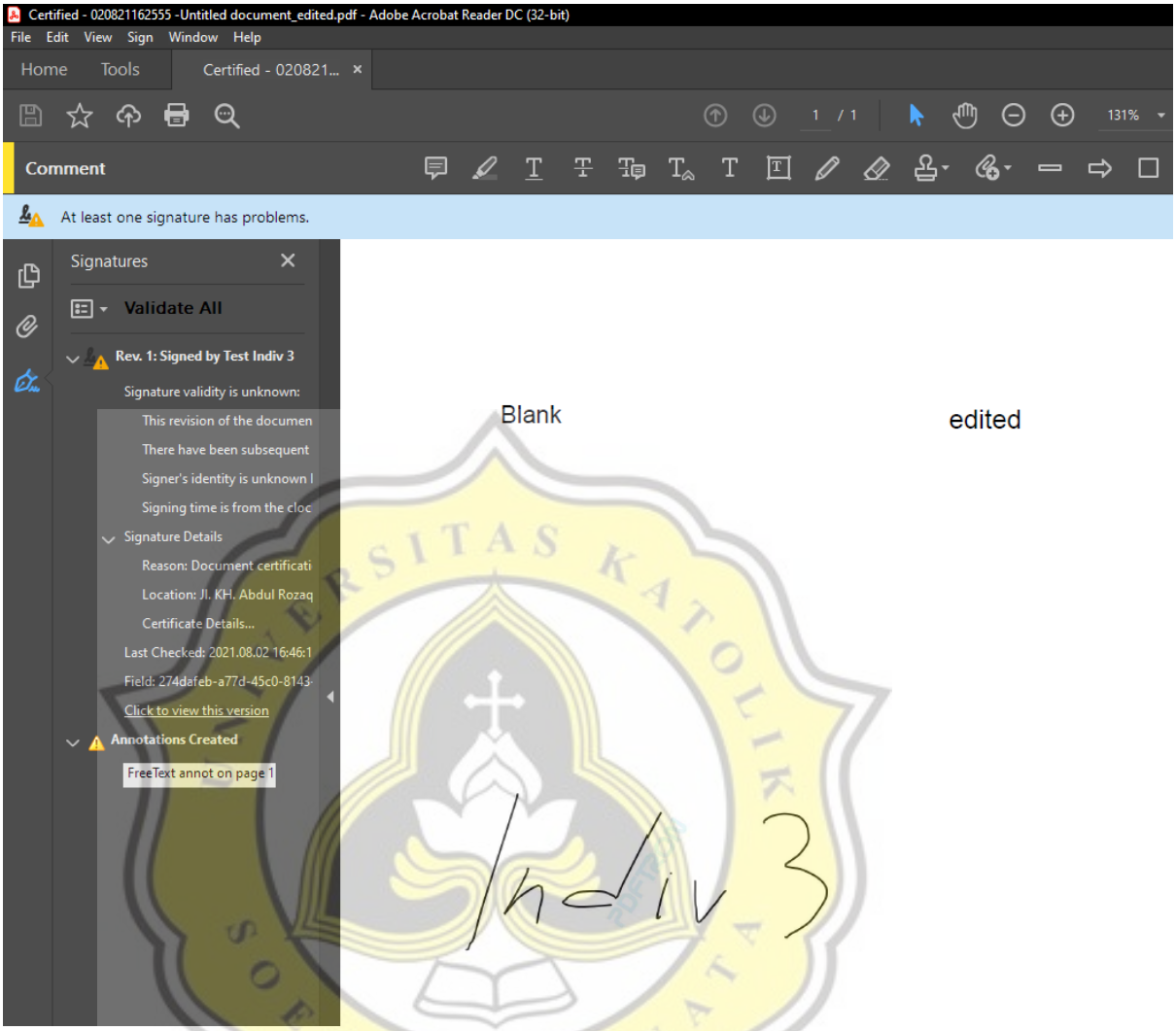
Pada gambar 4.75 terlihat bahwa sertifikat pada tanda tangan menjadi tidak valid lagi karena perubahan dokumen. Yang mana merupakan pelanggaran terhadap opsi “no change allowed”.



Gambar 4.75 Sertifikat yang menjadi invalid karena perubahan pada dokumen.

Apabila dokumen tidak dikunci dengan *permission*, menu *annotation/comment* dapat dimunculkan pada Adobe Reader seperti pada gambar 4.74. Dan juga apabila seseorang membuat perubahan pada *file* PDF tersebut kemudian menyimpannya, maka sertifikat dan tanda tangan yang menempel tidak menjadi *invalid* melainkan pengguna yang membuka dokumen tersebut akan melihat peringatan bahwa terdapat perubahan yang telah dilakukan misal pada gambar 4.76 adalah perubahan dengan menambahkan

annotation berupa teks. Perlu diingat bahwa tampilan tiap program penampil dokumen PDF mungkin akan memberikan tampilan hasil yang berbeda-beda pada tiap kasus.



Gambar 4.76 Hasil apabila dokumen tidak dikunci dengan permission.