# Role-based Auth for http4s microservices

https://github.com/sherpair/weather4s
https://gitter.im/sherpair/weather4s

Lucio Biondi

# Monolithic Architecture

# Microservices

# Our Auth implementation

- Plain *FP* Scala
- Simple microservice architecture
- No Auth service providers

# OWASP

https://github.com/OWASP/CheatSheetSeries

# Security recommendation

1. HTTPS, TLS +1.2
2. *POST* requests for authentication
3. Server-side data validation
4. Zero-Trust policy

# Access Control Workflow

1. User registration
2. Authentication
3. Authorisation

# User registration

```
{
  "accountId" : "jsmith",
  "firstName" : "John",
  "lastName" : "Smith",
  "email" : "john.smith@gmail.com",
  "secret" : [97,80,97,115,115,119,111,114,100]
}
```

POST /auth/signup

```
final case class SignupRequest(
  accountId: String,
  firstName: String,
  lastName: String,
  email: String,
  secret: Array[Byte]
)
```

```
CREATE TYPE ROLE AS ENUM ('Master', 'Member');

CREATE TABLE IF NOT EXISTS users (
          id BIGSERIAL PRIMARY KEY,
  account_id      TEXT NOT NULL UNIQUE,
  first_name      TEXT NOT NULL,
   last_name      TEXT NOT NULL,
       email      TEXT NOT NULL UNIQUE,
      secret      TEXT NOT NULL,
      active      BOOL DEFAULT false NOT NULL,
        role      ROLE DEFAULT 'Member' NOT NULL,
  created_at TIMESTAMP NOT NULL
);
```

# User registration (with *activation token*)

```
CREATE TYPE KIND AS ENUM ('Activation', 'EMail', 'Refresh');

CREATE TABLE IF NOT EXISTS tokens (
           id BIGSERIAL PRIMARY KEY,
     token_id      TEXT NOT NULL UNIQUE,
    member_id    BIGINT NOT NULL REFERENCES users(id),
         kind      KIND NOT NULL,
  expiry_date TIMESTAMP NOT NULL,
   created_at TIMESTAMP NOT NULL
);
```

```
def sendToken(user: User, kind: Kind): Unit = {
  val expiryDate = Instant.now.plusSeconds(duration.toSeconds)
  val tokenId = SecureRandomId.Strong.generate
  val token = tokenOps.insert(
    Token(tokenId, user.id, kind, expiryDate)
  )

  Postman.sendEmail(token, user, kind)
}
```

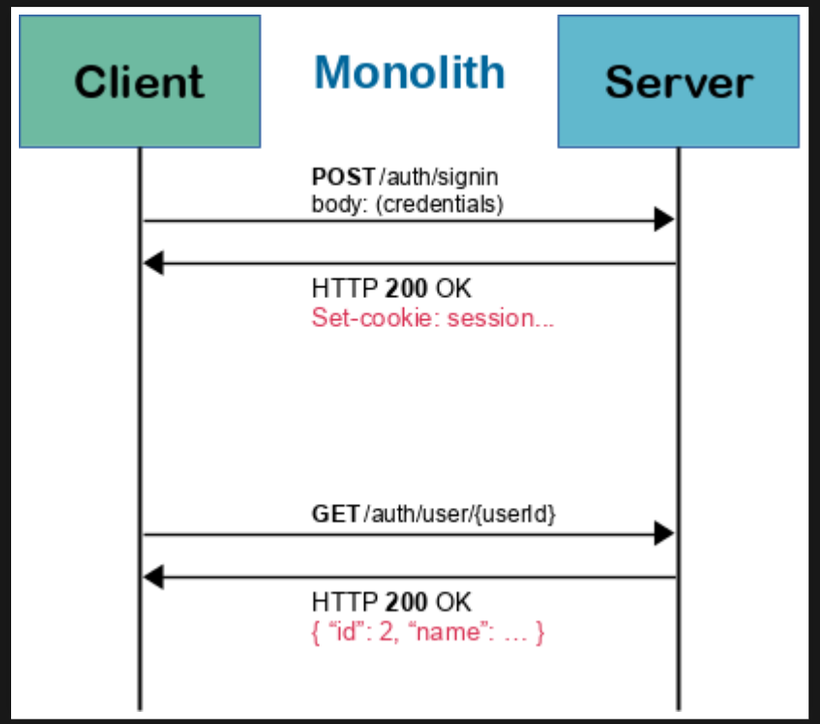
# Activation endpoint

https://sherpair.io/auth/account-activation/460025e13b1d224a049d9883a8f9572b6f3f4424088d9e734d18a0

# User authentication (*monolith*)

```
{
  "accountId" : "jsmith",
  "secret" : [97,80,97,115,115,119,111,114,100]
}
```

```
final case class SigninRequest(
  accountId: String,
  secret: Array[Byte]
 )
```




Centralized and fully *stateful*!

# *Stateful* Auth for Microservices ?

- No session
- No access to Auth service's DB
- No own "*User*" DB

# What about a *stateless* approach ?

- Authorisation for microservices ?
- *Fully* stateless ?
- What level of security can we aim to ?

# JWTs (*stateless*) & refresh tokens (*stateful!*)

*Easy approach* : JWTs & refresh tokens only

- won't give you a complete level of security
- can still be "okish", depending on your use case
- user, after a logout, might be not really logged out yet for a few minutes

*More complex approach*

- in addition to JWTs and refresh tokens
- a mechanism, not necessarily stateful, to inform the microservices that the user has been logged out

# User authentication (*Auth microservice*)

```
{
"accountId" : "jsmith",
"secret" : [97,80,97,115,115,119,111,114,100]
}
```

```
val routes: HttpRoutes[F] = HttpRoutes.of[F] {
  case request @ POST -> Root / "auth" / "signin" => signin(request)
}

def signin(request: Request[F]): F[Response[F]] =
  request.decode[SigninRequest] { sR =>
  userOps.find(sR.accountId)
    .flatMap(_.fold(notFound(accountId))(checkPw(_, sR)))
  }

def checkPw(user: User, sR: SigninRequest): F[Response[F]] =
  SCrypt.checkpwBool[F](sR.secret, PasswordHash[SCrypt](user.secret))
    .ifM(signinResponse(user), notFound(user.accountId))

def notFound(id: String): F[Response[F]] =
  NotFound(s"User(${id}) is not known")

def signinResponse(user: User): F[Response[F]] =
  if (user.active) tokenOps.addTokensToResponse(user, NoContent())
  else Forbidden("Inactive")

```

```scala
object KVStore extends IOApp {
  def run(args: List[String]): IO[ExitCode] = app

  val app: IO[ExitCode] =
    Ref.of[IO, Map[String, String]](Map.empty) >>= { ref =>
      val dsl = new Http4sDsl[F] {}; import dsl._

      def add(k: String, v: String): IO[Response[IO]] =
        ref.update(_.updated(k, v)) *> NoContent()

      def delete(k: String): IO[Response[IO]] =
        ref.update(_.removed(k)) *> NoContent()

      def find(k: String): IO[Response[IO]] =
        ref.get >>= { _.get(k).fold(NotFound())(Ok(_)) }

      val getRoute = HttpRoutes.of[IO] {
        case GET -> Root / "kv" / k => find(k) }

      val modRoutes = HttpRoutes.of[IO] {
        case DELETE -> Root / "kv" / k => delete(k)
        case    PUT -> Root / "kv" / k / v => add(k, v) }

      httpServer(getRoute <+> modRoutes)
    }

  def httpServer(routes: HttpRoutes[IO]): IO[ExitCode] =
    BlazeServerBuilder[IO]
      .withHttpApp(routes.orNotFound)
      .serve.compile.lastOrError
}
```

```scala
object KVStore extends IOApp {
  def run(args: List[String]): IO[ExitCode] = app[IO]

  def app[F[_]: CE: Timer]: F[ExitCode] =
    Ref.of[F, Map[String, String]](Map.empty) >>= { ref =>
      val dsl = new Http4sDsl[F] {}; import dsl._

      def add(k: String, v: String): F[Response[F]] =
        ref.update(_.updated(k, v)) *> NoContent()

      def delete(k: String): F[Response[F]] =
        ref.update(_.removed(k)) *> NoContent()

      def find(k: String): F[Response[F]] =
        ref.get >>= { _.get(k).fold(NotFound())(Ok(_)) }

      val getRoute = HttpRoutes.of[F] {
        case GET -> Root / "kv" / k => find(k) }

      val modRoutes = HttpRoutes.of[F] {
        case DELETE -> Root / "kv" / k => delete(k)
        case    PUT -> Root / "kv" / k / v => add(k, v) }

        http(getRoute <+> modRoutes)
      }

  def http[F[_]: CE: Timer](routes: HttpRoutes[F]): F[ExitCode] =
    BlazeServerBuilder[F]
      .withHttpApp(routes.orNotFound)
      .serve.compile.lastOrError
}
```

```scala
object KVStore {
  implicit val timer = IO.timer(global)
  implicit val contextShift = IO.contextShift(global)
  def main(args: List[String]): Unit = app.unsafeRun...
  val app: IO[Unit] =
```

```scala
import cats.effect.{ConcurrentEffect => CE}
```

# *Tagless Final* encoding

# User authentication (Auth microservice)

Origin of the *stateful* refresh token.

```scala
import tsec.common.{SecureRandomId => SR}

class TokenOps[F[_]](auth: Authenticator[F])(
    implicit C: AuthConfig, TR: TokenRepo[F], S: Sync[F]
) {

  def addTokensToResponse(
      user: User, responseF: F[Response[F]]
  ): F[Response[F]] =
    create(user, Kind.Refresh, C.token.refreshLife) >>= {
      auth.addTokensToResponse(responseF, user, _)
    }

  private def create(
      user: User, kind: Kind, duration: FiniteDuration
  ): F[Token] =
    for {
      now <- S.delay(Instant.now())
      expiryDate = now.plusSeconds(duration.toSeconds)
      token <- TR.insert(
        Token(SR.Strong.generate, user.id, kind, expiryDate)
      )
    }
    yield token
}
```

# The Authenticator class (Auth microservice)

Adds JWT & refresh token to the response *if* the credentials were validated.

```scala
class Authenticator[F[_]: Sync](
    jwtAlgorithm: JwtRSAAlgorithm, privateKey: PrivateKey) {

  def addTokensToResponse(
      responseF: F[Response[F]], user: User, refreshToken: Token
  ): F[Response[F]] =
    for {
      response <- responseF
      jwt <- Sync[F].delay(jwtEncode(user))
      cookie <- cookieWithRefreshTk(refreshToken)
    }
    yield response.addCookie(cookie).putHeaders(
        Authorization(Credentials.Token(AuthScheme.Bearer, jwt)))

  private def cookieWithRefreshTk(rt: Token): F[ResponseCookie] =
    Sync[F].delay(ResponseCookie(
      "refresh-token", rt.tokenId,
      expires = Some(HttpDate.unsafeFromInstant(rt.expiryDate)),
      httpOnly = true, secure = true
    ))

  private def jwtEncode(user: User): String = {
    val now = JwtTime.nowSeconds
    Jwt.encode(JwtClaim(
      user.claimContent, Claims.issO, user.id.toString.some,
      Claims.audO, (now + C.authToken.duration.toSeconds).some,
      now.some, now.some
    ), privateKey, jwtAlgorithm)
  }
}
```

# User authorisation (*other microservices*)

*Non*-public endpoints are
protected by Authoriser.

```scala
object Routes {
  def apply[F[_]: CE](c: Countries): F[HttpRoutes[F]] =
    for {
      authoriser <- Authoriser[F]("/sherpair.io/auth", RS256)
      routes <- CE[F].delay {
        authoriser(new CountryApp[F](countries).routes)
      }
    }
    yield routes
}
```

# User authorisation (other microservices)

*Non*-public endpoints are defined by using AuthedRoutes (from http4s).

Endpoints expecting from the JWT's claims some content, here of UserFromClaim type, may add a parameter, named here uFC, after an as keyword.

```
1
2  class CountryApp[F[_]: CE](c: Countries) extends Http4sDsl[F] {
3    implicit val countryEncoder: EntityEncoder[F, Country] =
4      jsonEncoderOf[F, Country]
5
6    val routes = AuthedRoutes.of[UserFromClaim, F] {
7      case GET -> Root / "geo" / "country" / id as _ =>
8        findCountry(id) >>= { _.fold(NotFound(id))(Ok(_)) }
9
10     case GET -> Root / "geo" / "health" as uFC =>
11       masterOnly(uFC, healthCheck)
12   }
13
14   def findCountry(id: String): F[Option[Country]] = ...
15
16   def healthCheck: F[Response[F]] = ...
17
18   def masterOnly(uFC: UserFromClaim, f: => F[Response[F]]) =
19     if (uFC.role == Master) f else Forbidden("Unauthorized")
20 }
```

The user's *role* is filtered at route-level by a supporting function, here named masterOnly.

# User authorisation (other microservices)

Authoriser is just an alias for AuthMiddleware (from http4s).

```
type Authoriser[F[_]] = AuthMiddleware[F, UserFromClaim]
type EitherUC = Either[String, UserFromClaim]

AuthMiddleware(Kleisli(req => validateJwt(req)), onFailure)
```

Authoriser decodes and validates the JWT.
If OK the request is authorised.

```
 val onFailure =
   Kleisli(req => OptionT.liftF(Forbidden(req.context)))

 def validateJwt(req: Request[F]): F[EitherUC] =
   retrieveBearerToken(req).fold(noJwt)(decodeJwt(_))

 def decodeJwt(token: String): F[EitherUC] =
   JwtCirce.decodeAll(token, publicKey, jwtAlgorithm)
     .map(tokenAsTuple => validateClaim(tokenAsTuple._2))
     .recover { case _: JwtException => notAuthorized }
     .liftTo[F]

 def validateClaim(claim: JwtClaim): EitherUC =
   if (claim.isValid(Claims.iss, issuer))
     decode[UserFromClaim](claim.content).fold {
     _ => notAuthorized, content => content.asRight[String]
   }
   else notAuthorized

 val noJwt = "Missing token".asLeft[UserFromClaim].pure[F]
 val notAuthorized = "Not authorized".asLeft[UserFromClaim]

```

# Questions ?

# Thanks!