Proyecto hsSSO. Infraestructura 100% Haskell para Single-Sign-On

Licenciatura en Computación

Perfil: Computación confiable

Instituto de Computación

Facultad de Ingeniería

Universidad de la República

Federico Yemurenko

federico.yemurenko@internet.com.uy

Supervisor: Alberto Pardo

pardo@fing.edu.uy

Octubre de 2021

RESUMEN

Este reporte presenta el proyecto desarrollado como Actividad Integradora para la obtención del título de Licenciado en Computación, perfil Computación confiable. Se implementó una infraestructura 100% Haskell para Single Sign On (SSO) sobre el protocolo OpenID Connect, formada por dos aplicaciones web implementadas a los efectos (hsColor y hsMsgs) y una pieza de software desarrollada por un tercero (Broch) como servidor de autenticación. Todo ello se desarrolló utilizando exclusivamente el lenguaje de programación funcional y tecnología Haskell.

DESCRIPTORES

Programación funcional. Haskell. Programación web. Ingeniería de software. Seguridad. Single-Sign-On. OpenId Connect. BROCH

Contenido

1. Presentación del proyecto

Como trabajo final de grado para la Licenciatura en Computación, se propuso desarrollar una infraestructura 100% Haskell para Single-Sign-On (SSO) sobre el protocolo OpenID Connect.

Se trata de un proyecto de programación donde se utiliza exclusivamente el lenguaje funcional Haskell y donde se diseña e implementa una solución operativa para un problema real.

El problema planteado para realizar la experiencia de programación surge de una necesidad encontrada en el ejercicio de la profesión en diversas empresas e instituciones: montar una infraestructura de SSO para integrar las aplicaciones existentes, con el objetivo de mejorar la seguridad y la experiencia del usuario. Las aplicaciones industriales no presentan mayor dificultad; en general incluyen módulos de integración con servidores de SSO sobre una variedad de protocolos, que sólo requieren ser configuradas. Las aplicaciones "in-house", desarrolladas a medida desde cero o en base a paquetes open-source personalizados, requieren desarrollos y adaptaciones para incluir un cliente de autenticación de usuarios, integrado al core de la aplicación y que funcione contra el servidor utilizado. Este es el aspecto del problema que se trata en el proyecto.

OpenID Connect (OIDC) es un protocolo para SSO ampliamente utilizado. Amazon, Google, Microsoft, PayPal, etc. lo utilizan para resolver la autenticación de usuarios en diversas plataformas y aplicaciones. Existen buena documentación y diversas implementaciones [17].

1.1. Objetivos

El objetivo principal del proyecto fue experimentar con la capacidad de Haskell y la tecnología asociada, para el desarrollo de sistemas para el "real world".

Después de haber estudiado los fundamentos de la programación funcional y de haber utilizado las primeras versiones de Haskell en un ambiente puramente académico en la segunda mitad de los años 90, me interesaba particularmente comprobar por la vía de la experiencia:

- Cómo habían evolucionado la disciplina y el lenguaje a partir de los "claims" iniciales que sustentaron el desarrollo de la programación funcional.
- Si Haskell es actualmente una opción o herramienta más para el desarrollo de sistemas industriales con los atributos requeridos por la ingeniería de software, al menos en determinadas áreas de aplicación.

Además, como objetivos específicos interesaba:

- Profundizar el conocimiento adquirido y ejercitado en diversos cursos la carrera, donde sólo se hizo programación funcional "in the small" para resolver ejercicios o pequeños proyectos.
- Realizar la experiencia de integrar la solución utilizando software existente, desarrollado por terceros, con software propio desarrollado desde cero.
- Conocer técnicas y herramientas para el desarrollo de aplicaciones web en Haskell.
- Profundizar el conocimiento en el problema del SSO, a nivel teórico y práctico.

1.2. Alcance

Desarrollar una infraestructura SSO sencilla pero completa utilizando exclusivamente el lenguaje de programación funcional Haskell requería:

- Instalar y poner a correr en la web un servidor Identity Provider (IdP)

Se utilizó BROCH, la única implementación Haskell para el protocolo OpenId Connect (OIDC) que se encontró luego de un relevamiento. Se aplicaron configuraciones y personalizaciones al código fuente de este sistema.

- Tener al menos dos aplicaciones para ilustrar el problema y para realizar la experiencia de integración, donde la autenticación de usuarios y el manejo de sesiones se delegan en el IdP.

  Se diseñaron e implementaron desde cero dos aplicaciones web sencillas, con sendos módulos internos de autenticación de usuarios. Se les implementó un cliente OIDC para su integración al SSO, manejando la autenticación en BROCH.

Además, se analizó la experiencia desde el punto de vista del proceso de desarrollo soportado por la tecnología Haskell para producir aplicaciones que puedan ser utilizadas en ambientes de producción.

1.3. Organización

El proyecto se organizó de la siguiente manera:

- **Etapa 1. Actividades preparatorias.**

  Planificación. Relevamiento y selección de bibliografía y software de base. Instalación del ambiente de trabajo.

- **Etapa 2. Refresh.**

  Repaso y actualización de Haskell y programación funcional a nivel básico y avanzado. Se leyeron los libros [10], [11] y [12] resolviendo los ejercicios de cada capítulo.

  Revisión y estudio de otros temas como HTTP, HTML, programación web, OIDC, fundamentos de ingeniería de software y diseño de sistemas.

  Relevamiento y entrenamiento en el uso de las de bibliotecas Haskell para desarrollo aplicaciones web (“Haskell Web Frameworks”).

- **Etapa 3. Desarrollo de las “aplicaciones testigo”.**

  Diseño e implementación de un módulo para autenticación de usuarios con funcionalidades de registro, login y logout.

  Diseño e implementación de dos aplicaciones web configurables, que utilizan el módulo de autenticación de usuarios desarrollado.

- **Etapa 4. Desarrollo de la infraestructura SSO.**

  Relevamiento, selección y puesta en funcionamiento de un servidor de autenticación de usuarios para el protocolo OIDC. Configuración y personalización.

  Implementación de un cliente OIDC para las aplicaciones desarrolladas.

  Pruebas generales de funcionamiento.

- **Etapa 5. Cierre de proyecto.**

  Elaboración del informe final del proyecto con sus conclusiones. Presentación del proyecto y los productos implementados.

Al inicio de la etapa 3 se comenzó con un régimen de reuniones quincenales con el supervisor del proyecto, para seguimiento y consulta.

1.4. Esfuerzo

Sumadas a las muchas horas dedicadas en la etapa inicial para el estudio de la bibliografía, al reaprendizaje de Haskell, a la actualización en otras tecnologías, técnicas, estándares, etc. necesarias para el desarrollo del proyecto, se dedicaron más de 220 horas para las etapas de preparación y desarrollo.

Además:

- Las aplicaciones implementadas comprenden más de 4500 líneas de código Haskell con comentarios.
- Para las reuniones de seguimiento y su preparación se dedicaron más de 18 horas.
- Para escribir el informe final y preparar la presentación se dedicaron más de 30 horas.

1.5. Entregables

Los productos entregables del proyecto son:

- El presente informe técnico.

  Proyecto_hsSSO-InformeFinal-FedericoYemurenko-v1_3.pdf

  Incluye instrucciones para montar el ambiente de trabajo e instalar y correr las aplicaciones

- La presentación del proyecto.

  Proyecto_hsSSO-Presentacion_FedericoYemurenko-20211110.pdf

- El código fuente de las aplicaciones desarrolladas y de BROCH
  - hsAuth.zip
  - hsColor.zip
  - hsMsgs.zip
  - broch-master.zip

2. El problema

En las infraestructuras corporativas, los usuarios, para su trabajo diario, tienen que utilizar diversos sistemas de información: eMail, CRM, ERP, sistemas específicos, etc. En general, estos sistemas requieren que los usuarios deban “ingresar al sistema” (login) presentado una “credencial”, que no sólo los identifica, sino que también les otorga acceso a determinadas funcionalidades, según los roles y perfiles asignados por los administradores de las distintas aplicaciones, para los diferentes usuarios. La forma más simple de credencial de usuario es un username y una password.

Pero manejar credenciales que pueden ser potencialmente diferentes en cada aplicación, no es efectivo ni amigable e implica riesgos de seguridad importantes. Además, desde el punto de vista del área de TI, la administración “distribuida” de usuarios, la protección de passwords u otros datos sensibles, de permisos, de roles, etc. se complejiza notablemente y se dificulta establecer políticas generales.

El problema no sólo aplica al “login de usuarios” por lo que se ha generalizado como el problema del acceso a recursos protegidos en una red. Los protocolos y las soluciones más generales lo presentan desde ese punto de vista, distinguiendo entre:

- Autorización: dar acceso a un recurso o a una funcionalidad.
- Autenticación: validar la identidad del usuario que pretende acceder a un recurso.

La parte del problema que se trata en este proyecto es la autenticación de usuarios.

2.1. Autenticación de usuarios

La solución ampliamente adoptada, dirigida fundamentalmente por la mejora de la seguridad y la gestión, es utilizar un sistema central especializado (en adelante "auth server") en el que las aplicaciones delegan la autenticación y la gestión de los usuarios.

Las diferentes aplicaciones y el auth server forman un ecosistema regulado por un protocolo específico. En ese ambiente:

- Los usuarios deben registrase en el auth server, con sus metadatos y las credenciales que se le requieran.
- Cuando un usuario pretende hacer login en una aplicación, esta obtiene las credenciales y las envía al auth server para validar si son las de un usuario registrado o redirige al usuario al auth server para que presente allí sus credenciales.

Las funciones básicas del auth server son:

- Un servicio de directorio: un repositorio donde se registran los usuarios y donde se mantienen sus credenciales y metadadatos: nombres, información de contacto, roles, permisos, etc.
- Un servicio de autenticación de usuarios: el proceso para la validación de credenciales.

El protocolo establece cómo es el directorio central (que datos puede tener; como están estructurados; etc.); cómo se accede y se obtienen los metadadatos de usuario; como es el proceso para la autenticación de usuarios; cómo se integran las aplicaciones al auth server; cómo se conectan; etc.

El muy conocido Lightweight Directory Access Protocol (LDAP) es un protocolo para servicios de directorio de propósito general; algunas implementaciones son Active Directory (Microsoft), openLDAP, Directory Server (Red Hat) y apacheDS.

El auth server reemplaza o complementa a los subsistemas internos de gestión de usuarios de cada aplicación, que son nativos, heterogéneos y suelen no ser módulos especializados en el manejo de información sensible, como son passwords, credenciales, datos personales, etc. Muchas aplicaciones soportan un *stack* configurable de métodos de autenticación y permiten habilitar uno o varios, dependiendo de cada instalación particular. Entre otros:

- autenticación local, contra un repositorio interno de usuarios (usualmente siempre está presente y es el método por default)
- autenticación contra un auth server externo sobre diferentes protocolos (LDAP, OIDC, SAML, Shibboleth, Kerberos, etc)
- autenticación por la IP de usuario
- autenticación por certificados X.509

2.2. Single-Sign-On (SSO)

El SSO es una mejora a los mecanismos de autenticación de un auth server externo. Con el SSO no sólo se atiende a las cuestiones de seguridad al permitir delegar la autenticación y la gestión de información sensible en un sistema central especializado, sino que también se mejora la

experiencia del usuario.

La característica distintiva de una solución de SSO es que el usuario inicia sesión una sola vez, obteniendo acceso (login) a todos los sistemas integrados, sin que se le solicite iniciar sesión, presentando credenciales nuevamente, cada vez que pretende ingresar a uno de ellos. Del mismo modo, cuando un usuario cierra la sesión en una de las aplicaciones, se produce un logout en todas las aplicaciones en las que había ingresado.

Las sesiones son globales y su inicio y cierre afectan a todas las aplicaciones integradas. Por lo tanto, una solución de SSO debe soportar las funciones básicas de un auth server más un mecanismo para el manejo de sesiones globales.

2.3. Infraestructura para SSO

Los componentes de una infraestructura de SSO básica son:

- El servidor de autenticación ("Identity Provider", IdP) que provee:
    - Repositorio que mantiene la base de datos de credenciales y metadatos.
    - Servicio de autenticación con manejo de sesiones globales
    - Mecanismos para el registro de clientes y registro de usuarios
- Los clientes o aplicaciones que requieren autenticación ("Service Providers", SPs), que interactúan con el IdP sobre HTTP.
- El protocolo de SSO que especifica la interacción entre los clientes y el servidor, atendiendo a la seguridad e interoperabilidad del ecosistema.

2.4. Ejemplos de SSO

Google

Numerosas aplicaciones y portales, los propios de Google y otros de terceros, utilizan los servicios de Google para la autenticación de usuarios.

Si se tiene una cuenta en Google, con esa misma cuenta se accede a Gmail, Google Drive, Meet, a su carpeta de fotos en la nube, Youtube, etc. Además, si se loggeó en cualquiera de ellas, no tendrá que volver a hacer login para ingresar a las otras.

Del mismo modo, si se está suscrito a un medio de prensa digital, por ejemplo, que ofrece la opción "Ingresar con Google", accederá directamente al contenido reservado para los suscriptores sin pasar por un login en el portal; tan sólo por tener el Gmail abierto en el mismo browser.

ANII

La Agencia Nacional de Investigación e Innovación (ANII) tiene una infraestructura de SSO donde están integradas muchas de las plataformas digitales de ANII.

Los usuarios registrados en ANII, en https://sso.anii.org.uy/, utilizan el mismo username y password para ingresar en cualquiera de las siguientes aplicaciones con sesiones globales:

- REDI (https://redi.anii.org.uy/) el repositorio digital de Acceso Abierto
- FOCO (https://foco.timbo.org.uy/) el buscador de publicaciones
- CV UY (https://cvuy.anii.org.uy/) para la gestión y publicación del curriculum on-line

Por ejemplo, puede comprobarse que si se ingresa a REDI no será necesario volver a presentar credenciales para ingresar a FOCO. Y que un logout en FOCO también cierra la sesión que se había iniciado en REDI.

Sin embargo, el sistema de gestión de postulaciones (https://postulaciones.anii.org.uy/) no está integrado al SSO. Como consecuencia, se requiere tener un usuario local en este sistema, que podrá ser diferente al registrado para ingresar a REDI, FOCO o CV UY. Y que, a pesar de tener una sesión iniciada en este sistema, se requerirá presentar credenciales para iniciar sesión en las otras aplicaciones.

ANII utiliza el Gluu Server (https://gluu.org/) como IdP.

2.5. OpenID Connect (OIDC)

OpenID Connect (OIDC) ([17]) es un protocolo especializado en la autenticación de usuarios de aplicaciones web, basado en el protocolo OAuth2 ([22]) para la autorización del acceso a recursos protegidos.

Permite montar una infraestructura de SSO segura e interoperable sobre HTTP, donde un servidor externo (OpenID Provider) verifica la identidad de los usuarios para darle acceso a cualquier aplicación integrada al sistema. También permite mantener y compartir la información básica del perfil del usuario.

Para especificarlo se tomó como base el protocolo general de autorización OAuth 2.0 y se lo adaptó para el caso particular de la autenticación de usuarios en un ecosistema de aplicaciones web con manejo de sesiones globales.

El protocolo define de forma precisa el flujo de mensajes, el formato de los datos, los parámetros, cómo y dónde se intercambian los datos dentro de los mensajes HTTP, el manejo de errores, etc.

Los aspectos contemplados por OpenID Connect 1.0 son:

- El registro y autenticación de los clientes (Relying Party, RP).

  https://openid.net/specs/openid-connect-registration-1_0.html
- La estructura del repositorio de usuarios (“claims”)

  https://openid.net/specs/openid-connect-core-1_0.html#Claims
- El mecanismo para la autenticación de usuarios, en tres variantes

  https://openid.net/specs/openid-connect-core-1_0.html#Authentication
- Aspectos de seguridad y privacidad

  https://openid.net/specs/openid-connect-core-1_0.html#Security

  https://openid.net/specs/openid-connect-core-1_0.html#Privacy

El protocolo no especifica:

- El proceso para el registro de usuarios (ABM en el repositorio de usuarios).
- Que credenciales se le requiere al usuario para autenticarse ni como se validan
- Como se manejan las sesiones

Authorization Code Flow

El mecanismo básico implementado en este proyecto para la autenticación de usuarios es la variante "Authorization Code Flow". Presentado de forma muy general, es un proceso de tres pasos sobre HTTP:

1. **PASO 1. Authentication Request**

   El cliente hace una redirección al "Authorization Endpoint" del auth server para:

   - que el usuario presente sus credenciales

     ó

   - determinar si el usuario ya está autenticado

   Si el usuario es autenticado exitosamente, porque las credenciales son válidas o porque ya hay una sesión global, el auth server envía al cliente un "código de autorización" (Access Code)

2. **PASO 2. Authorization Request**

   El cliente envía al "Token Endpoint" del auth server, el Access Code junto con su "client id" y "client secret" obtenidos en el proceso de registro del cliente.

   Si son válidos y consistentes, el auth server responde con un "código de acceso" (Access Token) al recurso protegido, que es la identidad del usuario (nombre, apellido, etc.) y un ID Token que identifica al usuario autenticado en el cliente.

   Se crea la sesión global en el SSO para el usuario.

3. **PASO 3. UserInfo Request**

   El cliente envía el Access Token al "UserInfo Endpoint" del auth server, para obtener como respuesta, por body y en formato JSON, los metadatos del usuario.

   Con esto el cliente puede iniciar la sesión local en la aplicación y dar acceso al usuario.

En el website de la OpenID Foundation (https://openid.net/), home de la comunidad dedicada al desarrollo y la promoción de OpenID, existe abundante información y documentación sobre el protocolo: especificaciones, bibliotecas, productos, políticas, guías para la implementación, etc. [17], [18], [19]

3. Implementación de las aplicaciones

A los efectos de ilustrar el problema y para mostrar la forma de integrar una aplicación a una infraestructura SSO existente, se implementaron dos "aplicaciones testigo": *hsColor* y *hsMsgs*

A continuación, se presentan de forma general las aplicaciones implementadas, haciendo énfasis en los aspectos de diseño.

3.1. Consideraciones de diseño

El principal objetivo para el diseño de las aplicaciones fue que estuviera bien identificado el módulo que hace la autenticación del usuario. Se buscaba que fuera sencillo y con mínimo impacto en el resto de la aplicación su reemplazo o modificación para la integración a una infraestructura de SSO.

Además, dado que el desarrollo se haría en el marco de un proceso de reaprendizaje del lenguaje Haskell, se buscaba soportar un plan de desarrollo gradual donde inicialmente se

implementarían soluciones sencillas para luego pasar a otras más sofisticadas, a medida que se iba adquiriendo mayor expertise, sin que fuera necesario reescribir todo de nuevo. Por ejemplo, debería ser posible comenzar con una interfase de usuario por std IO para luego pasar a una interfase web sobre HTTP. O comenzar con una implementación del estado de la aplicación en memoria para luego pasar a una base de datos.

Para ello se aplicó una técnica de diseño llamada “ports and adapters”, presentada brevemente en [11]. Ayuda a obtener un diseño modular donde se separa el core de la aplicación de las entidades externas. Permite agregar, modificar, reemplazar y combinar componentes sin afectar al core de la aplicación. Facilita el mantenimiento y la evolución de la aplicación.

Los “port” son interfases a las funciones del core. Los “adapters” son componentes que implementan la interacción entre las entidades externas (usuarios, bases de datos, la memoria, el filesytem, un servidor de mail, etc.) y el core a través de los ports.

En Java, los ports se implementan usando interfaces; en Haskel, con clases.

Como se verá más adelante en la presentación de las aplicaciones, esta técnica permitió atender a los objetivos planteados.

3.2. hsAuth

hsAuth es una aplicación para el registro y la autenticación de usuarios. En el plan de desarrollo, se implementó para tener:

- el módulo interno de gestión de usuarios de las aplicaciones que formarían el ecosistema SSO
- un servidor de autenticación externo, donde las aplicaciones testigo podrían delegar el registro y la autenticación de sus usuarios

Las “aplicaciones testigo” deberían poder implementarse extendiendo este desarrollo. Esto es, concebirlas como un módulo de gestión de usuarios más ciertas funcionalidades específicas.

Por lo tanto, *hsAuth* se diseñó para:

- ser reutilizable
- ser fácilmente extensible con nuevas funcionalidades
- facilitar que la funcionalidad de autenticación nativa pueda ser reemplazada por la autenticación en un servidor externo, a los efectos ilustrar claramente el proceso de su integración a una infraestructura SSO

Las funciones del core de hsAuth son:

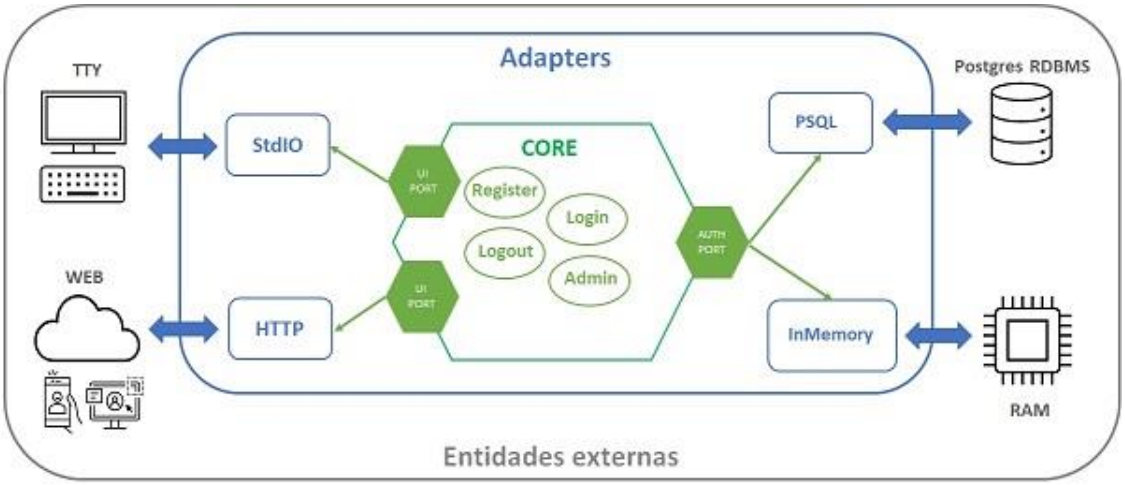
- registro de usuarios
- login
- logout
- funciones administrativas
    - usuarios loggeados
    - usuarios registrados

La aplicación corre leyendo / actualizando un estado interno dado por los usuarios registrados y los usuarios loggeados, dirigida por un usuario ejecutando las funciones del core a través de una interfase de usuario (UI)

Arquitectura

Aplicando la técnica de diseño “ports and adapters” se llegó a la arquitectura del siguiente diagrama.

Tiene una UI que inicialmente es por std IO que luego se cambió por un UI para la web. Tiene un componente para que inicialmente se implementen los repositorios de usuarios y de sesiones como un environment en memoria y luego se le agregaría un componente contra una base de datos postgreSQL para hacer persistente el repositorio de usuarios.



Nota: Para acotar el esfuerzo y avanzar más rápidamente al objetivo final, se resolvió no implementar la interfase para una base de datos. Todos los repositorios se implementaron como un environment en memoria.

Modularización

En el proyecto de implementación en Haskell, el diseño se reflejó en la siguiente estructura de módulos bajo `hsAuth/src/`:

```
Core
      Auth        -- Modulo interno para manejo de usuarios
            Auth              -- Credenciales
            User              -- Usuario: Credenciales y Metadatos
            Session           -- Sesión del usuario loggeado
            RepoUsers         -- Repo de usuarios registrados (register)
            RepoSessions      -- Repo de sesiones (login y logout)
            Admin             -- Admin de usuarios (listLoggedUsers y
                              -- listRegisteredUsers)
Adapter
      InMemory    -- Implementaciones como environment en memoria
            Auth              -- Repositorio de usuarios
            Session           -- Repositorio de sesiones
      PostrgreSQL -- Implementacion como estado persistente en DB
```

```
      StdIO       -- UI por std IO
            Auth        -- Acceso al auth por std IO
            Admin       -- Acceso al admin por std IO
      HTTP        -- Interfaces para la web
                  API         -- REST API para las funciones del
                              -- Core.Auth por HTTP
                  Client      -- HTTP client para auth
                  Server      -- Para funcionar como auth server
                  Backend     -- UI para la web
                  Web         -- css, imagenes, etc
APP   -- Definición y configuración de la aplicación
      App
      AppConfig
```

Haskell soportó perfectamente esa arquitectura por lo que fue posible separar de forma muy clara el core de la aplicación, de las entidades externas: las diferentes opciones de UI y posibles variantes para implementar los repositorios.

Notas de implementación

hsAuth está implementada en base a dos "repositorios" que forman el estado de la aplicación:

- un repositorio de usuarios registrados (credenciales y metadatos)
- un repositorio de sesiones de usuarios loggeados

En el módulo `APP.App` se encuentra la definición de la aplicación. Siguiendo las ideas presentadas en [11], la aplicación se modeló como una mónada que lee/actualiza el estado interno y que hace I/O:

```
newtype App a = App {
    unApp :: ReaderT AppState IO a
} deriving (MonadReader AppState, MonadIO)
runApp :: AppState -> App a -> IO a
runApp state = flip runReaderT state . unApp
```

El estado interno se define como el estado del repositorio de usuarios más el estado del repositorio de sesiones

```
type AppState = (TVar MT.State, TVar ST.State)
appInitialState = (MT.initialState, ST.initialState)
```

donde MT y ST tienen las definiciones de los tipos con los que se implementan los repositorios en el adapter InMemory para manejar un environment en memoria:

```
-- Interfaces entre core y la implementacion del repositorio
-- de usuarios como environment en memoria
import qualified Adapter.InMemory.AuthTypes as MT
import qualified Adapter.InMemory.AuthImpl as M
```

```
-- Interfaces entre core y la implementacion del repositorio
-- de sesiones como environment en memoria
import qualified Adapter.InMemory.SessionTypes as ST
import qualified Adapter.InMemory.SessionImpl as S
```

Ports

En los repositorios se definen las funciones del core por lo que se allí se especifican las interfases para acceder esas funciones.

Por ejemplo, en el módulo Core.Auth.RepoSessions está definido el repositorio de las sesiones activas de los usuarios loggeados. Provee las funciones de core *login* y *logout*.

También en base a [11], se lo especificó como la clase de una mónada con los métodos necesarios para definir las funciones específicas del core. Son las "primitivas" que definen la interfase entre esta parte del core y las entidades externas.

```
class Monad m => SessionRepo m where
    -- Crea una sesion para un usuario
    newSession :: UserId -> m SessionId
    -- Termina la sesion de un usuario
    killSession :: SessionId -> m ()
    -- Cual es el usuario de una sesion dada;
    findUserBySessionId :: SessionId -> m (Maybe UserId)
    -- Si el usuario esta loggeado, tiene una session activa;
    findSessionIdByUser :: UserId -> m (Maybe SessionId)
    -- Los usuarios que tienen una sesion activa
    loggedUsers :: m [UserId]
```

Las funciones específicas del core se definen en el mismo módulo, utilizando exclusivamente los métodos de la clase.

Por ejemplo

```
-- *** login
-- Crea una sesion para el usuario, si esta registrado y su
-- mail validado y no esta loggeado
login :: (UserRepo m, SessionRepo m) =>
         Auth -> m (Either LoginError SessionId)
login auth = runExceptT $ do
    result <- lift $ findUserByAuth auth
    case result of
        Nothing ->
            throwError LoginErrorInvalidAuth
        Just (_, False) ->
            throwError LoginErrorEmailNotVerified
        Just (uid, True) -> do
```

```
            session <- lift $ findSessionIdByUser uid
            case session of
                Nothing ->
                    lift $ newSession uid
                Just sid ->
                    throwError LoginErrorUserAlreadyLogged
```

De este modo, las funciones del core se definen de forma independiente de cualquier implementación de repositorio. Se concretan en una implementación particular cuando en el módulo `APP.App` se instancia la clase para construir la aplicación que se pondrá a correr.

Adapters

Los componentes que permiten la interacción del core con una u otra entidad externa deben proveer una implementación para cada uno de los métodos de la clase.

Ejemplo. Los módulos `Adapter.InMemory.AuthTypes` y `Adapter.InMemory.AuthImpl`, que se importan en `APP.App` como MT y M respectivamente, proveen una implementación del repositorio de sesiones como un environment en memoria.

```
-- *** El estado interno del repositorio in-memory de SESIONES
data State = State {
    stateSessions :: Map S.SessionId U.UserId
} deriving (Show, Eq)
initialState :: State
initialState = State {
    stateSessions = mempty
}

-- *** El repositorio in-memory de sesiones
-- r es un environment compartido: el estado.
-- MonadReader r representa funciones que leen ese environment
-- representado como TVar State

type InMemory r m = (Has (TVar State) r,MonadReader r m,MonadIO m)

-- *** newSession
-- Cambio de estado.  Genero la session Id como un string ramdom
-- con uid como prefijo
newSession :: (InMemory r m) => UserId -> m SessionId
newSession uid = do
      tvar <- asks getter
      sid <- liftIO $ ((tshow uid) <>)
             <$> stringRandomIO "[a-zA-Z0-9]{16}"
```

```
        atomically $ do
              state <- readTVar tvar
              let sessions = stateSessions state
                  newSessions = insertMap sid uid sessions
                  newState = state { stateSessions = newSessions }
              writeTVar tvar newState
              return sid
```

Cambiando la implementación

En el módulo `APP.App` también se construye la aplicación seleccionando los componentes disponibles para los distintos usos. Funciona instanciando las clases con una u otra implementación.

Por ejemplo, para construir la aplicación donde los repositorios de usuarios y de sesiones se implementan como environments en memoria:

```
-- Puertos del core
import Core.Auth.RepoUsers
import Core.Auth.RepoSessions

-- Interfaces entre core y la implementacion del repositorio
-- de usuarios como environment en memoria
import qualified Adapter.InMemory.AuthTypes as MT
import qualified Adapter.InMemory.AuthImpl as M

-- Interfaces entre core y la implementacion del repositorio
-- de sesiones como environment en memoria
import qualified Adapter.InMemory.SessionTypes as ST
import qualified Adapter.InMemory.SessionImpl as S

instance UserRepo App where
    addUser = M.addUser
    updateUserMetadata = M.updateUserMetadata
    setEmailAsVerified = M.setEmailAsVerified
    findUserByAuth = M.findUserByAuth
    findEmailByUserId = M.findEmailByUserId
    findNameByUserId = M.findNameByUserId
    findSurnameByUserId = M.findSurnameByUserId
    registeredUsers = M.registeredUsers
```

```
instance SessionRepo App where
    newSession = S.newSession
    killSession = S.killSession
    findUserBySessionId = S.findUserBySessionId
    findSessionIdByUser= S.findSessionIdByUser
    loggedUsers = S.loggedUsers
```

Pero, si tuviera implementado el componente de interfase con postgreSQL, de forma muy sencilla podría reconstruir la aplicación para, por ejemplo, elegir dar persistencia al repositorio de usuarios, implementándolo en base de datos.

Para ello, en `APP.App` simplemente y sin tocar otra línea de código de las indicadas abajo, debería:

1. Cambiar los componentes dela implementación InMemory por:

```
-- Interfaces entre core y la implementacion del repositorio
-- de usuarios como environment DB postgreSQL
import qualified Adapter.PostgreSQL.PSQLTypes as DBT
import qualified Adapter.PostgreSQL.PSQLImpl as DB
```

2. Cambiar la instancia de UserRepo:

```
instance UserRepo App where
    addUser = DB.addUser
    updateUserMetadata = DB.updateUserMetadata
    setEmailAsVerified = DB.setEmailAsVerified
    findUserByAuth = DB.findUserByAuth
    findEmailByUserId = DB.findEmailByUserId
    findNameByUserId = DB.findNameByUserId
    findSurnameByUserId = M.findSurnameByUserId
    registeredUsers = DB.registeredUsers
```

3. Redefinir el estado de la aplicación:

```
type AppState = (TVar DBT.State, TVar ST.State)
appInitialState = (DBT.initialState, ST.initialState)
```

La interfase de usuario (UI)

Teniendo implementadas las dos opciones de UI (std IO y HTTP) también de forma muy sencilla es posible reconstruir la aplicación para que utilice una u otra.

La UI se establece cuando se define el `main` de la aplicación, en `src/Lib.hs` La única diferencia es importar el módulo correspondiente y usar el runner de esa interfase:

- Para UI por stdIO

```
import qualified Adapter.StdIO.Main as StdIO
main = do   . . . . .
            StdIO.main (runApp (astate, sstate))
```

- Para UI por web

```
import qualified Adapter.HTTP.Main as HTTP
main = do   . . . .
            HTTP.main CFG.port(runApp (astate, sstate))
```

En `Adapter.StdIO.Main`:

```
main :: (MonadIO m,
         UserRepo m,
         EmailVerificationNotif m,
         SessionRepo m) => (m () -> IO ()) -> IO ()
main runner = runner actionMain

-- Menu Principal
--
actionMain :: (MonadIO m,
               UserRepo m,
               SessionRepo m,
               EmailVerificationNotif m) => m ()
actionMain = do
    liftIO $ MM.displayMenuOptions
    usrOpt <- liftIO $ MM.readMenuOption
    case usrOpt of
        -- Quit
        MM.OptQuit ->
            return ()
        -- Registro usuario
        MM.OptRegister -> do
            doRegister
            actionMain
        -- Login
        MM.OptLogin -> do
            doLogin
            actionMain
        -- Logout
        MM.OptLogout -> do
            doLogout
            actionMain
```

```
            -- Aministracion
            MM.OptAdmin ->
          actionAdmin
```

En `Adapter.HTTP.Main`:

```
main :: (MonadIO m,
         UserRepo m,
         EmailVerificationNotif m,
         SessionRepo m) =>
         Int -> (m Response -> IO Response) -> IO ()
main port runner = do
    api <- API.main runner
    svr <- SVR.main runner
    web <- WEB.main runner
    run port $ vhost [(pathBeginsWith "api", api),
                      (pathBeginsWith "svr", svr)] web
    where
  pathBeginsWith path req = headMay (pathInfo req) == Just path
```

En `Adapter.HTTP.Backend.Main`:

```
main :: (MonadIO m,
         UserRepo m,
         EmailVerificationNotif m,
         SessionRepo m) =>
        (m Response -> IO Response) -> IO Application
main runner = do
    cacheContainer <- initCaching PublicStaticCaching
    scottyAppT runner $ routes cacheContainer

routes :: (MonadIO m,
           UserRepo m,
           EmailVerificationNotif m,
           SessionRepo m) => CacheContainer -> ScottyT LText m ()
```

```
routes cacheContainer = do
    middleware $
        staticPolicy' cacheContainer (addBase CFG.appBase)
    -- Acceso a Auth
    AuthR.routes
    -- Acceso a Auth
    AdminR.routes
    -- Other routes
    notFound $ do
        status status404
        json $ ("NotFound" :: Text)
--        html "NotFound"
    -- Default
    defaultHandler $ \e -> do
        status status500
  json ("InternalServerError" :: Text)
```

Otros componentes

hsAuth, para su versión web, también tiene implementados:

- Una API REST que permite acceder a las funciones del core desde un cliente HTTP por las siguientes rutas de la aplicación web.

  ```
  routes :: (MonadIO m,
             UserRepo m,
             EmailVerificationNotif m,
             SessionRepo m) => ScottyT LText m ()
  ```

  routes = do

  ```
      -- register
      post "/api/auth/register" doRegister
      -- login
      post "/api/auth/login" doLogin
      -- logout
      post "/api/auth/logout" doLogout
      -- User info
      post "/api/auth/user" doWhoAmI
  ```

  Está implementada en el módulo `Adapter.HTTP.API`

- Un servidor HTTP que provee una función que permite que las aplicaciones testigo utilicen *hsAuth* como servidor de autenticación externo, delegando el registro de usuarios y la validación de credenciales.

```
-- *** authenticate
-- Autentica un usuario en el auth server, dadas sus credenciales,
-- si esta registrado y su mail validado.
-- Devuelve la user metadata
authenticate :: (UserRepo m) => Auth -> m (Either AuthError User)
```

La función está disponible en la siguiente ruta de la aplicación web:

```
-- Authenticate
post "/svr/auth/authenticate" doAuthenticate
```

Está implementado en el módulo `Adapter.HTTP.Server`

3.3. hsColor

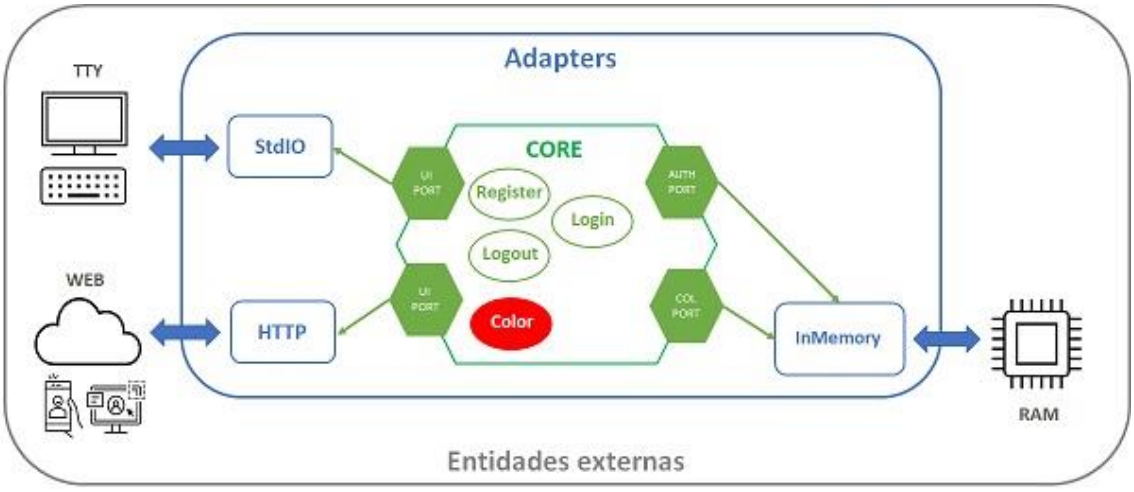
hsColor es una aplicación que hace registro y login de usuarios y que su única funcionalidad específica ("feature") es mostrar un "color secreto" al usuario loggeado. Las funciones del core son:

- register
- login
- logout
- myColor

Es una aplicación muy sencilla, concebida para mostrar de forma simple y efectiva las virtudes del diseño adoptado para agregar funcionalidades a una aplicación existente. Se construyó tomando *hsAuth* y agregando la funcionalidad específica. Dicho de otro modo, *hsColor* es *hsAuth* con una nueva funcionalidad.

Soporta UI por stdIO y por la web, lo cual es configurable en la definición de la aplicación. La implementación del estado interno es un environment en memoria.

Su arquitectura es la siguiente:



Agregando funcionalidades

La construcción de *hsColor* sirve para ilustrar la forma en que el diseño adoptado permite agregar nuevas funcionalidades a una aplicación existente, de forma sencilla, transparente y separada de lo ya implementado.

Sobre la estructura de módulos de *hsAuth*:

1. Se agregó el módulo `Core.Feature` donde se define Color, se especifica el repositorio de colores y se define la función `myColor`
2. Se agregó los módulos `Adapter.InMemory.FeatureImpl` y `Adapter.InMemory.FeatureTypes` donde se define el repositorio de colores como un environment en memoria

   Para simplificar, se implementó como el estado inicial hard-coded en Adapter.InMemory.FeatureTypes
3. Se agregó el módulo `Adapter.StdIO.Feature` donde se implementa el acceso por stdIO a la nueva funcionalidad
4. Se agregó el módulo `Adapter.StdIO.Menu.FeatureMenu` donde se implementa el menú de features de la aplicación. Se lo incluye en el menú principal implementado en `Adapter.StdIO.Menu.MainMenu`
5. Se agregó el módulo `Adapter.HTTP.Web.Feature` donde se implementa el acceso por HTTP a la nueva funcionalidad; se incluye ahí la nueva ruta
6. Se rediseñaron las páginas HTML de la aplicación en `Adapter.HTTP.Web.Pages`

Luego, sólo queda redefinir la aplicación en `APP.App` de la siguiente forma:

1. Extender el estado de hsAuth para agregar el repositorio de colores:

   ```
   type AppState = (TVar MT.State, TVar ST.State, TVar FT.State)
   appInitialState = ( MT.initialState
                     , ST.initialState
                     , FT.initialState )
   ```

   donde MT, ST y FT son las implementaciones del repositorio de usuarios, sesiones y colores respectivamente
2. Instanciar la clase ColorRepo con la implementación d el repositorio de colores.

   ```
   -- Interfaces entre core y la implementacion del repositorio de
   -- colores como environment en memoria
   import qualified Adapter.InMemory.FeatureTypes as FT
   import qualified Adapter.InMemory.FeatureImpl as F
   instance ColorRepo App where
       colorsInRepo = F.colorsInRepo
       genColor = F.genColor
   ```
3. Ajustar el `main` en `src/Lib.hs` para iniciar la aplicación con el estado extendido.

   ```
   main :: IO ()
   main = do
       astate <- newTVarIO $ appAuthState appInitialState
       sstate <- newTVarIO $ appSessionState appInitialState
       fstate <- newTVarIO $ appFeatureState appInitialState
       HTTP.main AppConfig.port (runApp (astate, sstate, fstate))
   ```

De esta forma, bajo `hsColor/src/` la estructura de módulos de *hsColor* quedo de la siguiente forma:

```
Core
      Auth        -- Modulo interno para manejo de usuarios
            Auth              -- Credenciales
            User              -- Usuario: Credenciales y Metadatos
            Session           -- Sesión del usuario loggeado
            RepoUsers         -- Repo de usuarios registrados (register)
            RepoSessions      -- Repo de sesiones (login y logout)
            Admin             -- Admin ususarios (listLoggedUsers y
                              -- listRegisteredUsers)
      Feature     -- Features específicas de hsColor (myColor)
            Color       -- Colores
            RepoColors  -- Repositorio de colores (myColor)
Adapter
      InMemory    -- Implementaciones como environment en memoria
            Auth        -- Repositorio de usuarios
            Session     -- Repositorio de sesiones
            Feature     -- Repositorio de colores
      PostrgreSQL -- Implementaciones como estado persistente en DB
                  -- (NO IMPLEMENTADO)

      StdIO       -- UI por std IO
            Auth        -- Acceso al auth por std IO
            Admin       -- Acceso al admin por std IO
            Feature     -- Acceso al feature stdIO
      HTTP        -- Interfaces para la web
                  API         -- REST API para funciones del auth en HTTP
                  Client      -- HTTP client para auth en hsAuth
                  ClientSSO   -- OIDC client para SSO con BROCH
                  Web         -- UI por HTTP
```

Cabe destacar que en este proceso, las modificaciones al código reutilizado de *hsAuth* fueron mínimas y claramente localizadas.

3.4. hsMsgs

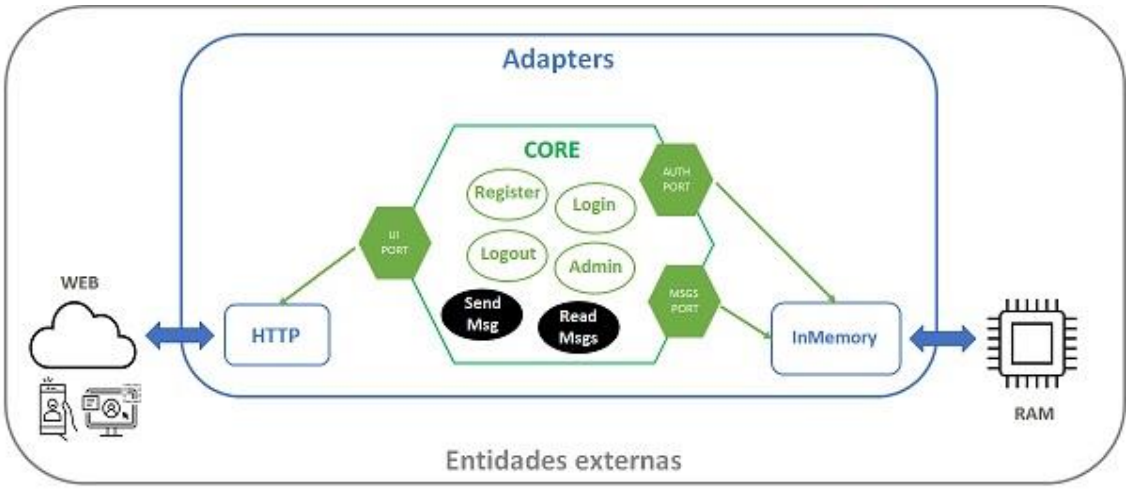
hsMsgs es una aplicación implementada a los efectos de tener una segunda aplicación en el ecosistema SSO para ilustrar mejor el problema.

Permite el intercambio de mensajes entre los usuarios. Cualquier usuario que ingrese al sistema puede ver los mensajes recibidos y puede enviar un mensaje a cualquier usuario registrado.

Las funciones del core de esta aplicación son:

- register
- login
- logout
- sendMsg
- readMsgs

La UI es sólo por la web y la implementación del estado interno es un environment en memoria. Su arquitectura es la siguiente:



El proceso para su implementación fue idéntico al utilizado para implementar *hsColor*: tomar *hsAuth* como base y agregarle las funcionalidades específicas (o remover alguna funcionalidad, como es el caso de la UI por std IO).

En el módulo `APP.App` se define y configura la aplicación. Alli puede verse que:

- el estado interno es una extensión del estado interno de *hsAuth*, al que se agrega el repositorio de mensajes

  ```
  type AppState = (TVar MT.State, TVar ST.State, TVar FT.State)
  ```

  donde MT, ST y FT son las implementaciones del repositorio de usuarios, sesiones y mensajes respectivamente.

- la aplicación no cambia:

  ```
  newtype App a = App {
      unApp :: ReaderT AppState IO a
  } deriving (MonadReader AppState, MonadIO)
  runApp :: AppState -> App a -> IO a
  runApp state = flip runReaderT state . unApp
  ```

El repositorio de mensajes se especificó en el módulo `Core.Feature.RepoMsgs` como la clase de una mónada con los métodos necesarios para implementar las funciones específicas del core.

```
class Monad m => MsgRepo m where
    -- Agrega un mensaje al repositorio
    newMsg :: UserId -> UserId -> Msg -> m ()
```

```
    -- Los mensajes depositados para un usuario
    getMsgs :: UserId -> m [(UserId, Msg)]

-- *** sendMsg
-- Envia un mensaje a un usuario registrado.
-- Da Nothing si el recipient no esta registrado
--
sendMsg :: (UserRepo m, MsgRepo m) =>
           UserId -> UserId -> Msg -> m (Maybe ())
sendMsg recipient sender msg = do
    result <- findEmailByUserId recipient
    case result of
        Nothing ->
            return Nothing
        Just email -> do
            newMsg recipient sender msg
            return (Just ())

-- *** readMsgs
-- Trae los mensajes de un usuario
readMsgs :: (UserRepo m, MsgRepo m) => UserId -> m [(UserId, Msg)]
readMsgs = getMsgs
```

Para la implementación InMemory del repositorio de mensajes se definió el siguiente tipo en el módulo `Adapter.InMemory.FeatureTypes`

```
 data State = State {
      -- (receipient, [(sender, msg)]
      stateMsgs :: [(U.UserId, [(U.UserId, M.Msg)])]
} deriving (Show, Eq)

initialState :: State
initialState = State {
      stateMsgs = []
}
-- *** El repositorio in-memory de mensajes
--
type InMemory r m = (Has (TVar State) r,MonadReader r m,MonadIO m)
```

Con esa definición, en el módulo `Adapter.InMemory.FeatureImpl` se implementaron los métodos de la clase.

En el módulo APP.App también se instancian las clases con las que se especificaron las interfaces al core (ports) como forma de elegir entre todas las posibles implementaciones (adapters). Alli puede verse que la aplicación se configuró para la implementación InMemory del respositorio de mensajes ya que F es el módulo `Adapter.InMemory.FeatureImpl` indicado antes.

```
instance MsgRepo App where
    newMsg = F.newMsg
    getMsgs = F.getMsgs
```

Finalmente, la aplicación también tiene un cliente HTTP para delegar la autenticación del usuario en *hsAuth* cuando se lo utiliza como servidor externo de autenticación y un cliente OIDC para su integración a la infraestructura SSO.

Ambas features se configuran en `APP.AppConfig`, sin ser necesario tocar una solo línea de código. Los parámetros que pueden configurarse son las constantes exportadas por el módulo:

```
module APP.AppConfig (
        appName
      , appBase
      , appFavicon
      , appLogo
      , port
      , AuthMode (..)
      , authMode
      , authServerURI
) where .........

data AuthMode = Local
              | AuthSvr
              | SSO
              deriving (Show, Eq)
```

4. Desarrollo de la infraestructura SSO

4.1. BROCH

BROCH ([16]), “A Haskell implementation of OpenID Connect”, fue la solución seleccionada como servidor de autenticación de la infraestructura SSO. Fue el único paquete 100% Haskell que se encontró, que implementa completamente el proceso de autenticación de usuarios del protocolo OIDC.

Está listado en OpenID (https://openid.net/developers/uncertified/) donde se lo presenta como “Currently more a research project than production ready”.

Fue desarrollado por Luke Taylor (tekul@pm.me), quien parece ser un profesional independiente y un “Haskell enthusiast”, que trabaja en Glasgow. El desarrollo inicial es de 2016; el último commit en GitHub fue el 17/11/2020. La solución completa está implementada en 4700 líneas de código Haskell.

En el proyecto, BROCH se configuró para correr en el puerto 8085 del servidor 18.237.171.141, con una base de datos postgeSQL.

Responde a las rutas /home, /login y /logout más los “endpoints” requeridos por OIDC.

El entendimiento de cómo BROCH resuelve los diferentes problemas, se obtuvo a partir del entendimiento del protocolo, de la escasa documentación en [16], de analizar algunas partes del código y de realizar numerosas pruebas. Su modularización y el estilo de codificación fueron de gran ayuda para entender la aplicación.

Registro del cliente

Tiene una API REST para el registro del cliente. Para acceder a la API debe utilizarse cURL, alguna herramienta más amigable o implementar una aplicación cliente a los efectos. En el proyecto se utilizó la herramienta Insomnia.

El Client Registration Endpoint es /connect/register.

Los datos requeridos por OIDC se envían en el body de la REQUEST en formato JSON; el resultado también viene como JSON en el body de la RESPONSE. Ejemplo:

REQ BODY:

```
{
   "application_type": "web",
   "client_name": "hsMsgs",
   "redirect_uris":
     ["http://18.237.171.141:8083/sso/auth/login"],
   "token_endpoint_auth_method": "client_secret_basic",
   "keys": [],
   "contacts": ["federico.yemurenko@internet.com.uy"]
}
```

RESP BODY:

```
{
  "application_type": "web",
  "redirect_uris": [
    "http://18.237.171.141:8083/sso/auth/login"
  ],
  "client_secret": "de98319bdcfcc57b",
  "client_id": "cfcead67c9c57e7a",
  "client_name": "hsMsgs",
  "keys": [],
  "token_endpoint_auth_method": "client_secret_basic",
  "contacts": [
    "federico.yemurenko@internet.com.uy"
  ]
}
```

El `client_id` y el `client_secret` deben utilizarse para configurar la aplicación cliente en su integración al SSO.

OBS: Por lo que parece ser un bug, luego del registro de un cliente hay que acceder a la tabla `oauth2_client` y cambiar el valor del campo `keys` del nuevo registro, reemplazando `[null]` por `[]`.

Autenticación de usuarios

Soporta el proceso completo del OIDC para la autenticación de usuarios, en la variante Authorization Code Flow. Los endpoints del proceso son:

- Authorization Endpoint: /oauth/authorize
- Token Endpoint: /oauth/token
- UserInfo Endpoint: /connect/userinfo

Las credenciales de usuario son username y password.

Tiene una interfase HTML muy sencilla para el ingreso de las credenciales del usuario.

Registro de usuarios

El repositorio de usuarios está implementado en dos tablas de la base de datos:

- `op_user` para las credenciales username y password
- `user_info` para los metadatos de usuario, con los campos y nombres requeridos por OIDC.

No tiene implementada una interfase para el registro de nuevos usuarios. Debe hacerse directamente en la base de datos o implementar una aplicación a los efectos.

En el proyecto, el registro de usuarios se hizo directamente en la base de datos, utilizando la herramienta pgAdmin.

Por default, BROCH espera encontrar la password encriptada con *bcrypt*. A los efectos de facilitar las pruebas, los usuarios se registraron en la base de datos con la password en claro y se modificó el código de BROCH, reemplazando `validatePassword` por `(==)` en la función `authenticate` del main de la aplicación `($BROCH_HOME/broch-server/broch.hs`.

`validatePassword` es una función de la biblioteca `Crypto.KDF.BCrypt`. Dados dos strings, los compara por igualdad luego de aplicar *bcrypt* al primero.

```
validatePassword :: (ByteArray password, ByteArray hash) =>
                    password -> hash -> Bool
```

Sesiones

El manejo de las sesiones globales está resuelto con “cookies de sesión”.

Al autenticar por primera vez a un usuario en un cliente, con la respuesta al Authorization Request, BROCH también envía una cookie llamada *sid*, que contiene, entre otros, el Authorization code y la fecha de vencimiento.

Mientras la cookie esté vigente, el browser la envía en todas las interacciones con el servidor. El servidor verifica si el Authorization code es válido, para determinar si se trata de un usuario que ya tiene una sesión iniciada en el SSO; esto es, si se trata de un usuario autenticado. En ese caso, no se le solicita que se autentique nuevamente presentando credenciales. Si el browser no envía la cookie, porque no la tiene (tal el caso del usuario que pretende ingresar por primera

vez) o porque está vencida, se entiende que se trata de un usuario no autenticado. En ese caso, se le pide credenciales, presentándole la interfase de login.

El cierre de sesión (el logout en el SSO) se resuelve invalidando el Authorization code que el browser envía en la cookie se sesión.

El manejo de cookies es un mecanismo dependiente del browser y es configurable.

4.2. Integración de hsColor y hsMsgs al SSO

Para la integración de cualquiera de las aplicaciones testigo al SSO fue necesario desarrollar un módulo cliente que implementó la interacción con el servidor siguiendo lo especificado en el protocolo OIDC para el Authorization Code Flow.

Se volvió a comprobar que el diseño de la aplicación facilitó notablemente este proceso.

Cliente OIDC

Se implementó el cliente OIDC en el módulo `Adapter.HTTP.ClientSSO.Auth` que se agregó a la estructura de módulos de la aplicación. Introduce como nueva funcionalidad la posibilidad, configurable, de autenticar usuarios en un servidor OIDC externo. El cliente reemplazó las funcionalidades nativas de *login* y *register* implementadas en el módulo interno de manejo de usuarios.

Primero, para implementar el paso 1 del protocolo, se cambió la acción que se ejecuta al hacer click en el botón LOGIN de la interface. La modificación se hizo en el módulo que implementa la UI HTML del sistema: `Adapter.HTTP.Web.Pages.Html`

La siguiente es la función que implementa la barra superior de navegación, que contiene el botón LOGIN. Como se verá, depende de la configuración del sistema.

```
-- *** navbarHome
-- La nav bar de la home page de la app
navbarHome :: H.Html
navbarHome = do
    H.div ! A.class_ "nav navbar-nav navbar-right" $ do
      H.ul ! A.class_ "nav navbar-nav navbar-right" $ do
        H.li $
          H.a ! A.class_ "btn btn-auth"
              ! A.href loginRoute $ "Login"
    where
        loginRoute = case CFG.authMode of
                       CFG.Local ->
                         "/auth/login"
                       CFG.AuthSvr ->
                         "/auth/login"
                       CFG.SSO ->
                         -- OIDC Step 1
                         -- Mando al usuario a autenticarse en
                         -- el auth point del BROCH
```

```
                                    H.toValue SSO.sso_LoginPageURL
```

Si la aplicación se configuró para SSO, el botón LOGIN redirecciona al usuario al SSO.sso_LoginPageURL, lo cual dispara el proceso de autenticación en BROCH.

Debe ser una URI bien formada según lo especificado por el OIDC para el paso 1. Incluye, entre otros parámetros, el Authorization Endpoint de BROCH en el path y en la query, el client_id obtenido en el proceso de registro de la aplicación en BROCH y la redirect_uri, que es adonde BROCH debe enviar la REQUEST GET que inicia el paso 2 luego que el usuario es autenticado.

Luego, en el nuevo módulo `Adapter.HTTP.ClientSSO.Auth`:

1. Se definió la ruta /sso/auth/login de la redirect_uri, que atiende la REQUEST GET del servidor, que se genera como resultado la solicitud de autenticación disparada por la acción que atiende el botón LOGIN (el paso 1 del proceso)

```
routes :: (ScottyError e,
           MonadIO m,
           UserRepo m,
           EmailVerificationNotif m,
           SessionRepo m) => ScottyT e m ()


routes = do
    -- authenticate en BROCH server
    get "/sso/auth/login" doAuth
```

2. Se implementó `doAuth` la acción que se ejecuta cuando se recibe un GET a esa ruta. Se puede verificar en el código adjunto, que su implementación resultó muy limpia y clara, siguiendo estrictamente los tres pasos del proceso presentados antes.

```
doAuth :: (ScottyError e,MonadIO m,UserRepo m,SessionRepo m,
           EmailVerificationNotif m) =>
           ActionT e m ()
```

Utiliza dos funciones auxiliares que implementan los pasos 2 y 3 del proceso:

```
step2Req :: (ScottyError e,
             MonadIO m) =>
             SSOT.OIdAuthCode ->
             ActionT e m (Maybe SSOT.OIdTokenData)


step3Req :: (ScottyError e,
             MonadIO m) =>
             SSOT.OIdTokenData ->
             ActionT e m (Maybe SSOT.BrochUser)
```

Autenticado el usuario y obtenidos sus metadatos, se cierra el proceso dando acceso a la aplicación e iniciando una sesión local, lo que está implementado en:

```
loginInApp :: (ScottyError e,
               MonadIO m,
               UserRepo m,
               SessionRepo m,
               EmailVerificationNotif m) =>
               SSOT.BrochUser -> ActionT e m ()
```

Configuración del cliente

LoginPageURL, client_id, client_secret, los Endpoints y el resto de los parámetros de configuración de BROCH como servidor de autenticación SSO, está definida en el módulo APP.SSOConfig.hs

El Logout

No se implementó el logout en el SSO. El botón LOGOUT de la interfase, solo cierra la sesión local en la aplicación. Por un lado, representaba un esfuerzo adicional que se resolvió no realizar en esta instancia. Además, ambos comportamientos pueden ser aceptables, dependiendo del contexto del ecosistema (el tipo de aplicaciones, cómo se usan, para qué)

1. que el logout en la aplicación A sólo cierre la sesión local en A pero mantiene al usuario autenticado en el SSO, en el entorno del browser. Por lo tanto, en ese entorno, el login a otra aplicación B integrada al SSO no le pedirá credenciales al usuario; ya lo da por autenticado y le da acceso. Llo mismo, si se quiere volver a entrar a A. Además, si estaba loggeado y trabajando en C, no pasa nada; sigue trabajando. El issue es cuando se quiere entrar como otro usuario: se tiene que ir a otro browser.
2. el comportamiento estándar en un SSO: que el logout en A, hace también un logout en el SSO ("desautentica" al usuario en ese entorno), lo cual debe desencadenar logouts locales en todas las aplicaciones integradas al sistema en las que estuviera loggeado, en ese entorno. Como consecuencia, si también estaba loggeado en B pero sólo quería salir de A pero seguir trabajando en B, tendría que volver a presentar credenciales desde B.

No obstante ello, se comprobó que la función en /logout de BROCH funciona de acuerdo a lo esperado por el comportamiento estándar en un SSO.

Notas sobre la implementación

No se implementaron todos los detalles requeridos por el protocolo; por ejemplo, no se validan client sercret ni id token. Queda para el trabajo futuro.

El único impacto en el resto del código implementado hasta el momento, es que obviamente la nueva webapp definida para atender a la nueva ruta /sso/auth/login (definida en `Adapter.HTTP.ClientSSO.Main.hs`) debe agregarse al vhost de la webapp principal (`Adapter.HTTP.Main.hs`) que el main de la aplicación sirve al web server.

```
main port runner = do
    api <- API.main runner
    web <- WEB.main runner
    cli <- CLI.main runner
    sso <- SSO.main runner
```

```
run port $ vhost [(pathBeginsWith "api", api)
                 ,(pathBeginsWith "cli", cli)
                 ,(pathBeginsWith "sso", sso)] web
where
 pathBeginsWith path req = headMay (pathInfo req) == Just path
```

Soportado por el diseño de la aplicación, su integración al SSO resultó un tarea sencilla: se resolvió *agregando* un nuevo módulo y *reconfigurando* el comportamiento del botón LOGIN.

La solución implementada no es dependiente de BROCH ni de cualquier otra aplicación que se use como auth server sino sólo del protocolo. Cualquier otro servidor que soporte el protocolo OpenID Connect podría reemplazar a BROCH con auth server, solamente reconfigurando lo que corresponda en `APP.SSOConfig.hs` y sin tocar ni una sola línea de código en el cliente SSO presentado antes ni en cualquier otro módulo de la aplicación.

5. Haskell

Está fuera de alcance analizar y presentar el lenguaje de programación funcional Haskell. Sólo mencionaré algunos puntos destacados que surgieron directamente de la experiencia.

5.1. Features destacadas

Las features de Haskell que fueron de gran ayuda en el diseño y la implementación son:

- El sistema de tipos para corregir errores, para guiar la implementación y para entender una solución implementada por otro. En general, la experiencia mostró que “si compila, si pasa el type-checker, entonces funciona bien”.
- Las funciones de alto orden (ejemplo: validate password) para escribir código general y reutilizable
- Las type classes para el diseño ports and adapters, permitiendo independizar la definición de los repositorios de usuarios y sesiones de varias posibles implementaciones
- Los data types para la definición y manejo de los objetos: usuarios, credenciales, estado, etc.
- Las mónadas para los efectos laterales por IO y por el diseño en base a un estado interno en memoria. El 80% de la programación fue con mónadas: sin la “do notation” hubiera sido enormemente más complejo.

5.2. Proyectos y modularización

stack y cabal funcionan muy bien. Son herramientas efectivas, amigables y fáciles de utilizar para la organización y compilación de proyectos complejos. La documentación es suficiente.

stack resultó una herramienta más efectiva y amigable cuando se complicó la compilación, por cantidad y variedad de dependencias. Luego de aprender a usar sus features básicas, permite trabajar en un ambiente altamente productivo.

5.3. Soporte

Hay una comunidad involucrada en el desarrollo del lenguaje y las tecnologías asociadas. Es un ecosistema abierto y open-source activo y en crecimiento No es exclusivamente académico, sino

que está integrado tanto por muchos de los “padres fundadores” como por entusiastas y por un número amplio y creciente de profesionales.

Hackage es un repositorio muy completo, bien estructurado y con muy buena documentación. Junto con haskell.org y wiki.haskell.org contienen casi toda la información, soporte, recursos, referencias a otras fuentes, etc. que se necesitan para el aprendizaje, la actualización y el desarrollo de un proyecto complejo.

5.4. Ambientes productivos

En este trabajo no se analizó si las aplicaciones Haskell se comportan adecuadamente en ambientes productivos, en cuanto performance, seguridad, requerimientos de recursos, etc. Sin embargo, hay numerosos testimonios de académicos y de profesionales de la industria que aseguran que las aplicaciones Haskell son rápidas, eficientes, estables y seguras.

5.5. Compatibilidad

Otra ventaja es que es un sólo estándar y una única distribución (o como dice Peyton-Jones en [8]: no hay “distribuciones”); no existe el Haskell de tal o el de cual; sólo hay Haskell 2010. Por tanto, todas las aplicaciones Haskell son compatibles.

6. Conclusiones

De la bibliografía y otras fuentes consultadas y de la experiencia realizada surge que Haskell estaría listo “for the real world”. Sería una herramienta más entre todas las que puede utilizar el programador profesional para el desarrollo de aplicaciones en diversas áreas.

El lenguaje y el ecosistema están maduros: no carecen de las principales features que se le requieren a las plataformas “mainsteam” de propósito general. Tendría lo necesario para ser una tecnología capaz de ser utilizada para el desarrollo de sistemas en una variedad amplia de áreas de aplicación. Seguramente para algunas esté más maduro que para otras; o haya más experiencia y recursos; y seguramente también en ciertas áreas de aplicación sea una opción para ambientes productivos exigentes y para otras no lo sea. Con esto se abre una pregunta interesante para el futuro: ¿en dónde (o para qué) es realmente bueno Haskell?

Desde el punto de vista del proceso de desarrollo de software, permite resolver completamente problemas reales y llegar a productos mantenibles, reutilizables, extensibles, efectivos, etc. Es una herramienta potente para el desarrollo de prototipos completos y para el ensayo de ideas. Quizás también sea adecuada para desarrollar sistemas que corran en ambientes productivos, aunque no puede concluirse nada en este aspecto sin más estudio y experiencia en su desempeño, particularmente en cuanto a performance, capacidad de procesamiento, estabilidad y seguridad. Este estudio debería hacerse en relación con las diferentes áreas de aplicación.

Otra conclusión de la experiencia es la rapidez con la que se pueden desarrollar sistemas que no sólo “funcionan” sino que también poseen los atributos requeridos por la ingeniería de software, lo cual está soportado por el lenguaje de forma nativa y natural. En otras palabras, con Haskell sería posible acelerar el proceso de desarrollo de software sin renunciar a las buenas prácticas de diseño y codificación.

Luego, para que un profesional llegue a un “estado productivo” donde aplique la conclusión anterior, la curva de aprendizaje sería más lenta, comparando con un lenguaje procedural tradicional. Para adquirir un manejo adecuado y completo del lenguaje y la tecnología asociada, parece ser necesario contar con una formación de base sólida en lógica, matemática,

fundamentos de programación, resolución de problemas e ingeniería de software. Además, se requiere una buena dosis de "madurez matemática" y capacidad de abstracción, que requieren entrenamiento y tiempo. De otro modo creo que sería difícil entender Haskell y utilizarlo adecuadamente explotando todo su potencial. No obstante ello, de aquí surgen otras preguntas: ¿cómo resultaría si la formación en programación se inicia enseñando a definir funciones en lugar de enseñar a escribir algoritmos?[1] y ¿con qué debe acompañarse?

Finalmente, el lenguaje tiene sus propias ventajas y features destacadas, muchas de las cuales han sido integradas en las nuevas versiones de lenguajes de programación más utilizados. Entre ellas, hay que destacar al sistema de tipos de Haskell y su implementación en el compilador *ghc*, que resultó de gran ayuda para evitar y corregir errores y para guiar el diseño y la programación, i.e. la especificación y la resolución de los problemas. También, las funciones de alto orden y patrones de recursión sobre listas.

7. Trabajo futuro

Sobre la base del trabajo desarrollado en el proyecto y los productos entregados, se abren algunas posibilidades de trabajo; desde ejercicios de programación hasta proyectos medianos.

- En Haskell y programación funcional
    - Generalizar el generador de menús para Std IO
    - Agregar funcionalidades a las aplicaciones.
    - Implementar las UI que le faltan a BROCH
    - Implementar otras aplicaciones que requieran autenticación de usuarios, reutilizando el código de hsAuth y/o el del cliente OIDC implementado.
    - Implementar la verificación del mail, integrando un servidor smtp
    - Integrar una base datos a las aplicaciones
    - Mejorar el manejo de errores y excepciones. Integrar un sistema de log
    - Mejorar la modularización de la implementación de la interfase HTML
    - Integrar páginas dinámicas y javascript
    - Analizar el comportamiento de las aplicaciones en un ambiente de acceso concurrente y mejorar los mecanismos implementados
    - Hacer que las aplicaciones sean configurables sin necesidad de recompilar por tener parámetros hard-coded
    - Integrar una aplicación Haskell existente al SSO con BROCH
    - Cambiar BROCH por otro IdP (Haskell o no)
- En ingeniería de software
    - Mejoras al diseño y modularización
    - Analizar Haskell y las aplicaciones desarrolladas con esta tecnología para un ambiente de producción real. Performance; volumen; carga; concurrencia; eficiencia; estabilidad; seguridad; etc.

[1] Seguramente habría un conflicto grande con la industria …….

- En seguridad
    - Profundizar en los aspectos de seguridad del SSO y del protocolo OIDC
    - Implementar mejoras de seguridad en las aplicaciones; mejoras al cliente OIDC básico.
    - Implementar las aplicaciones sobre HTTPS
- En testing y verificación
    - Hacer el testing de las aplicaciones utilizando alguna herramienta específica, como QuickCheck
    - Especificar algún aspecto del problema y hacer una verificación formal en algún asistente de pruebas.

8. Bibliografía y recursos de información

[1] haskell.org
Información y documentación general sobre Haskell
https://www.haskell.org/documentation/

[2] Wiki. The Haskell Programming Language
https://wiki.haskell.org/Haskell

[3] "Haskell 2010. Language Report". Marlow, Simon (editor) [July 2010]

[4] Hackage: The Haskell Package Repository
http://hackage.haskell.org/

[5] Haskell web-frameworks
http://wiki.haskell.org/Web/Frameworks

[6] Spock. A complete Haskell web framework for rapid development.
https://hackage.haskell.org/package/Spock/
http://www.spock.li

[7] Scotty: Haskell web framework.
https://hackage.haskell.org/package/scotty/
https://hackage.haskell.org/package/scotty-0.12/docs/Web-Scotty.html/
https://github.com/scotty-web/scotty/wiki/Scotty-Tutorials-&-Examples/

[8] "Escape from the ivory tower: the Haskell journey". Peyton-Jones, Simon [2017]. Conferencia dictada en el Churchill College, Cambridge, UK.
https://www.youtube.com/watch?v=re96UgMk6GQ

[9] "A History of Haskell: Being Lazy with Class". Hudak, Paul. Hughes, John. Peyton-Jones, Simon. Wadler, Philip. [2007]. Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, CA

[10] "Programming in Haskell". Hutton, Graham. [2016]. Cambridge University Press

[11] "Practical Web Development with Haskell". Putrady, Ecky. [2018]. Apress

[12] "Practical Haskell: A real world guide to programming". 2nd Ed. Serrano Mena, Alejandro. [2019]. Apress

[13] "Developing Web Applications with Haskell and Yesod". Snoyman, Michael. [2013]. O'Reilly

[14] "The Snap Framework. A Web Toolkit for Haskell". Collins, Gregory. Beardsley, Doug. [2011]. IEEE.

[15] "Introduction to Haskell (Spring 2013)". Curso on-Line. Yorgey, Brent [2013]; UPenn
https://www.cis.upenn.edu/~cis194/spring13/lectures.html

[16] BROCH. OAuth2 and OpenID Connect in Haskell [2016]
https://broch.io/posts/oauth2-openid-connect-haskell/
https://github.com/tekul/broch/

[17] OpenID Connect. Información general
https://openid.net/

[18] OpenID Connect. Especificaciones y guías de implementación
https://openid.net/wg/connect/status/

[19] OpenID Connect. Basic Client Implementer's Guide 1.0
https://openid.net/specs/openid-connect-basic-1_0.html

[20] "SoK: Single Sign-On Security – An Evaluation of OpenID Connect". Mainka, C. Mladenov, V. Schwenk, J. Wich, T. [2017]. 2017 IEEE European Symposium on Security and Privacy, Paris

[21] "RFC 2616. Hypertext Transfer Protocol -- HTTP/1.1". IETF. [1999]

[22] "RFC 6749. The OAuth 2.0 Authorization Framework". IETF. [2012]

[23] "Essential system administration". 2nd Ed. Frisch, Aeleen. [1995]. O'Reilly

[24] "HTML. La guía completa". 2nd Ed. Musciano, Chuck. Kennedy, Bill. [1997]. O'Reilly

[25] "HTML & CSS. Design and build websites". Duckett, Jon. [2011]. John Wiley & Sons.

[26] "PHP and MySQL. Web development". 4th Ed. Thomson, Laura. Welling, Luke. [2009]. Addison Wesley

[27] "Fundamentals of software engineering". Ghezzi, Carlo. Jazayeri, Medhi. Mandioli, Dino. [1991]. Prentice – Hall

[28] QuickCheck: Automatic testing of Haskell programs.
https://hackage.haskell.org/package/QuickCheck/
http://www.cse.chalmers.se/~rjmh/QuickCheck/manual.html/

[29] Verify Haskell Programs with *hs-to-coq*. [2020] Tutorial. The programming languages group. University of Pennsylvania
https://www.cis.upenn.edu/~plclub/blog/2020-10-09-hs-to-coq/

Anexo 1. Ambiente de trabajo

El ambiente de trabajo se montó en una máquina virtual Linux, en la nube de Amazon Web Services (AWS).

Plataforma

- AWS EC2 Instance type t3.medium
- 4 GB / 2-Core CPU
    - Public IP: 18.237.171.141
    - Open ports: 8081 - 8085
    - Root Volume: 8GB montado en /
    - Data Volume: 30GB montado en /opt
- OS: Ubuntu 18.04.4 LTS (bionic)

Software de base

- Haskell.
    - The Glorious Glasgow Haskell Compilation System, version 9.0.1 que incluye:
        - Compilador ghc
        - Intérprete interactivo ghci
    - Cabal version 3.4.0.0
    - Stack version 2.7.3
- Bibliotecas Haskell y paquetes destacados
    - classy-prelude-1.5.0

      Un prelude mejorado que reemplaza al standard prelude. Muy utilizado por desarrolladores de aplicaciones.
    - warp-3.3.17

      El web server utilizado en la mayoría de los sistemas Haskell en producción.
    - wai-3.2.3 (Web Application Interface)

      API o capa de abstracción entre el web server y las aplicaciones.
    - scotty-0.12.6

      Biblioteca de alto nivel para la implementación de aplicaciones web. Simple, fácil de usar y efectiva. Provee funciones para el ruteo y para el manejo de HTTP requests y responses. Corre sobre wai.
    - blaze-html-0.9.1.2

      Biblioteca de combinadores para escribir HTML
    - digestive-functors-0.8.4.2

      Biblioteca para manejo de web forms. Compatible con Scotty.

    - aeson-1.4.7.1

      Biblioteca para manejo de JSON

- postgreSQL 9.4+

Herramientas de desarrollo y administración

- Cliente SSH: puTTY 0.7+
- Cliente FTP: winSCP 5.13+
- GUI LXDE (Lightweight X11 Desktop Environment)
- XRDP for Remote Desktop Protocol (RDP)
- Editor de texto: Visual Studio Code ver 1.54.1 w./ ghc-mod and haskero extensions
- Cliente postgreSQL: pgAdmin 4.3.5+
- Insomnia Client, herramienta para el testing de clientes web

Anexo 2. Instalación y uso de las aplicaciones

Instalación

Montar una plataforma como la indicada en el Anexo 1.

Instalar y configurar el software de base y las herramientas allí indicadas.

Referencias, información y recursos indicados en la Bibliografía.

Ejecución

Las instrucciones para instalar y correr las aplicaciones se incluyen en los Readme en la distribución de cada una de las aplicaciones.

Anexo 3. Cliente SSO

Incluimos el código completo del cliente SSO implementado.

```
-- Cliente HTTP para auth en BROCH SSO

module Adapter.HTTP.ClientSSO.Auth (
      routes
) where

import ClassyPrelude
import Data.Has
import Data.Aeson hiding (json, (.:))
import Web.Scotty.Trans
import Network.HTTP.Client
import Network.HTTP.Types
import Network.HTTP.Types.Status
import Core.Auth.User
import Core.Auth.Auth
import Core.Auth.RepoUsers
import Core.Auth.RepoSessions
import Adapter.HTTP.Web.Pages.Html
import Adapter.HTTP.Common
import qualified APP.AppConfig as CFG
import qualified APP.SSOConfig as SSO
import qualified Adapter.HTTP.ClientSSO.Types as SSOT

routes :: (ScottyError e,
           MonadIO m,
           UserRepo m,
           EmailVerificationNotif m,
           SessionRepo m) => ScottyT e m ()
routes = do
    -- authenticate en BROCH server
    get "/sso/auth/login" doAuth
```

```
-- OId Auth-Workflow. Step 2 y Step 3

doAuth :: (ScottyError e,
           MonadIO m,
           UserRepo m,
           SessionRepo m,
           EmailVerificationNotif m) => ActionT e m ()
doAuth = do
    -- Recibo REQ del auth point del SSO luego de haber autenticado
    -- al usuario. Tomo la data para el Step 2. Viene en la query

    authCode <- param (fromStrict SSO.sso_AUTH_CODE_NAME)

    -- OId Step 2
    step2Res <- step2Req authCode
    case step2Res of
      Nothing ->
          renderHtml $
          generalErrorPage "BROCH error. RESP to Step 2 REQ"
      Just tknData -> do
          -- OId Step 3
          step3Res <- step3Req tknData
          case step3Res of
            Nothing ->
                renderHtml $
                generalErrorPage "BROCH error. RESP to Step 3 REQ"
            Just userData -> do
                -- Fin del proceso. Doy acceso en la app al usuario
                -- autenticado
                loginInApp userData

-- *** OId Step 2
-- Armo la REQ para el token point del SSO con auth code.
-- La data va por body en un cliente
-- Devuelvo la respuesta en un OIdTokenData
--
```

```
step2Req :: (ScottyError e,
             MonadIO m) =>
             SSOT.OIdAuthCode ->
             ActionT e m (Maybe SSOT.OIdTokenData)

step2Req authCode = do
    -- Armo la REQ para el token endpoint.
    -- La data va por body.
    let code = SSOT.toTxt authCode
        accessTokenData =   "grant_type=" ++ "authorization_code"
                          ++ "&"
                          ++ "client_id=" ++ SSO.sso_CLIENT_ID
                          ++ "&"
                          ++ "client_secret="
                          ++ SSO.sso_CLIENT_SECRET
                          ++ "&"
                          ++ "code=" ++ code ++ "&"
                          ++ "redirect_uri="
                          ++ SSO.app_step2redirectURI
    initRequest <- liftIO $
                   parseRequest (unpack SSO.sso_accessTokenURL)
    let request = {
         method = "POST"
       , requestHeaders = [
        ("Content-Type", "application/x-www-form-urlencoded")
        ]
      , requestBody = RequestBodyLBS $
                      (fromStrict . encodeUtf8) accessTokenData
       }
    manager <- liftIO $ newManager defaultManagerSettings
    responseStep2 <- liftIO $ httpLbs request manager
    return $ decode (responseBody responseStep2)

-- *** OId Step 3
-- Armo la REQ para el user point del SSO con los token.
-- La data va por headers en un cliente
-- Devuelvo la respuesta en un BrochUser
--
```

```
step3Req :: (ScottyError e,
             MonadIO m) =>
             SSOT.OIdTokenData ->
             ActionT e m (Maybe SSOT.BrochUser)
step3Req s2data = do
    -- Armo la REQ para el user endpoint.
    -- El access_token va por header.
    let accessToken = SSOT.access_token s2data
        idToken = SSOT.id_token s2data
        tokenType = SSOT.token_type s2data
        authHdr = "Bearer " ++ accessToken

    initRequest <- liftIO $
                    parseRequest (unpack SSO.sso_userMdataURL)
    let request = initRequest {
                  method = "POST"
                  , requestHeaders = [
                    ("Accept", "application/json")
                    , ("cache-control", "no-cache")
                    , ("Authorization", (encodeUtf8 authHdr))
                    ]
                  }
    manager <- liftIO $ newManager defaultManagerSettings
    responseStep3 <- liftIO $ httpLbs request manager
    return $ decode (responseBody responseStep3)

-- *** loginInApp
-- Con los metadatos del usuario autenticado que obtuve del SSO,
-- procedo al login en la app
-- Doy acceso a sus resources y actualizo los metadatso locales
-- por si cambiarion en el SSO
--
loginInApp :: (ScottyError e,
               MonadIO m,
               UserRepo m,
               SessionRepo m,
               EmailVerificationNotif m) =>
               SSOT.BrochUser -> ActionT e m ()
```

```
loginInApp buser = do
    let localauth = Auth (Email (SSOT.email buser))
                         (Password "NO_LOCAL_PWD")
        localmdata = UsrMetadata (SSOT.given_name buser)
                                 (SSOT.family_name buser)
        localuser = User localauth localmdata
    findResult <- lift $ findUserByAuth localauth
    case findResult of
      Nothing -> do
        regResult <- lift $ register localuser
        case regResult of
          Left err -> do
            renderHtml $ generalErrorPage (pack $ show err)
          Right () -> do
            doLocalEmailVerif (authEmail localauth)
            doLocalLogin localauth
      Just (uid,_) -> do
        updResult <- lift $ updateUser uid (usrMetadata localuser)
        case updResult of
          Nothing -> do
            renderHtml $ generalErrorPage "Auth OK pero no
                         se pudo actualizar datos del usuario"
          Just () ->
            doLocalLogin localauth
    where doLocalLogin auth = do
              loginResult <- lift $ login auth
              case loginResult of
                Left err ->
                  renderHtml $ generalErrorPage (pack $ show err)
                Right sId -> do
                  setSessionIdInCookie sId
                  redirect "/user"
          doLocalEmailVerif mail = do
              verifres <- lift $ forceVerifyEmail mail
              case verifres of
                Left err -> do
                  renderHtml $ generalErrorPage (pack $ show err)
                Right () -> return ()
```