Memòria pràctica 4

DAT QT2019

Alba Mendez

28 de desembre de 2019

1 Introducció

L'objectiu d'aquesta última pràctica és programar el mateix sistema de fòrum de la pràctica anterior, però amb una SPA de client, que ataca contra una API REST on està el backend.

Igual que en la pràctica anterior, el backend es dona amb l'estructura bàsica, el model i les rutes (que ara són diferents) fetes. L'estudiant ha d'omplir la lògica de cada mètode / ruta.

Per fer el frontend es recomana fer servir AngularJS, i així ho hem fet.

Durant el transcurs de la pràctica s'han corregit alguns bugs o problemes en el codi donat, que es detallen en la última secció de la memòria.

2 Backend

Abans de començar amb el backend, importarem les funcions d'utilitat que vam definir en la última secció de la memòria anterior, `requireAdmin` i `requireLeader`. La funció `isAdmin` ja estava definida pel professor en aquesta pràctica. Ara podem canviar totes les comprovacions existents d'administrador, per una crida a `requireAdmin`.

També copiem la funció `deleteFullQuestion` de la pràctica anterior, al nou projecte:

```
deleteFullQuestion :: QuestionId -> ForumDb -> IO ()
deleteFullQuestion qid conn = do
    answers <- getAnswerList qid conn
```

```
    forM_ answers $ \ (aid, _) -> deleteAnswer aid conn
    deleteQuestion qid conn
```

Ara podem omplir la funció `deleteFullTheme`, amb un codi molt similar:

```
deleteFullTheme :: ThemeId -> ForumDb -> IO ()
deleteFullTheme tid conn = do
    questions <- getQuestionList tid conn
    forM_ questions $ \ (qid, _) -> deleteFullQuestion qid conn
    deleteTheme tid conn
```

Només queda doncs, emplenar els handlers que no estan implementats. El codi és el següent:

```
-- Questions list

getThemeQuestionsR :: ThemeId -> HandlerFor Forum Value
getThemeQuestionsR tid = do
    questions <- runDbAction $ getQuestionList tid
    jqs <- forM questions questionToJSON
    pure $ object [("items", toJSON jqs)]

postThemeQuestionsR :: ThemeId -> HandlerFor Forum Value
postThemeQuestionsR tid = do
    user <- requireAuthId
    (pTitle, pText) <- getRequestJSON $ \ obj -> do
        title <- obj .: "title"
        text <- obj .: "text"
        pure (title, text)
    time <- liftIO $ getCurrentTime
    let newquestion = Question tid user time pTitle pText
    qid <- runDbAction $ addQuestion newquestion
    getQuestionR qid

-- Question

getQuestionR :: QuestionId -> HandlerFor Forum Value
getQuestionR qid = do
    question <- runDbAction (getQuestion qid) >>= maybe notFound pure
    questionToJSON (qid, question)

deleteQuestionR :: QuestionId -> HandlerFor Forum Value
deleteQuestionR qid = do
    user <- requireAuthId
```

```
    question <- runDbAction (getQuestion qid) >>= maybe notFound pure
    theme <- runDbAction (getTheme $ qTheme question) >>= maybe
      ↪ notFound pure
    requireLeader theme user
    runDbAction $ deleteFullQuestion qid
    pure $ object []

-- Answers list

getQuestionAnswersR :: QuestionId -> HandlerFor Forum Value
getQuestionAnswersR qid = do
    answers <- runDbAction $ getAnswerList qid
    janswers <- forM answers answerToJSON
    pure $ object [("items", toJSON janswers)]

postQuestionAnswersR :: QuestionId -> HandlerFor Forum Value
postQuestionAnswersR qid = do
    user <- requireAuthId
    pText <- getRequestJSON $ \ obj -> do
        obj .: "text"
    time <- liftIO $ getCurrentTime
    let newanswer = Answer qid user time pText
    aid <- runDbAction $ addAnswer newanswer
    getAnswerR aid

-- ---------------------------------------------------------------
-- Answer

getAnswerR :: AnswerId -> HandlerFor Forum Value
getAnswerR aid = do
    answer <- runDbAction (getAnswer aid) >>= maybe notFound pure
    answerToJSON (aid, answer)

deleteAnswerR :: AnswerId -> HandlerFor Forum Value
deleteAnswerR aid = do
    user <- requireAuthId
    answer <- runDbAction (getAnswer aid) >>= maybe notFound pure
    question <- runDbAction (getQuestion $ aQuestion answer) >>= maybe
      ↪ notFound pure
    theme <- runDbAction (getTheme $ qTheme question) >>= maybe
      ↪ notFound pure
    requireLeader theme user
    runDbAction $ deleteAnswer aid
    pure $ object []
```

3 Millores al backend

Per aconseguir tota la funcionalitat que teniem en la pràctica anterior, cal afegir algunes característiques al backend.

Validació del leader

Una cosa que ja no ve implementada en el codi donat és la validació del leader. Per tant modifiquem el handler POST de la ruta /themes per retornar `invalidArgs` si l'usuari no existeix:

```
postThemesR :: HandlerFor Forum Value
postThemesR = do
    user <- requireAuthId
    users <- getsSite forumUsers
    (pLeader, pTitle, pDescription) <- getRequestJSON $ \ obj -> do
        -- ...
    requireAdmin user
    unless (isJust $ lookup pLeader users) (invalidArgs ["Leader must
      ↪  be an existing user"])
    let newtheme = Theme pLeader "" pTitle pDescription
    tid <- runDbAction $ addTheme newtheme
    getThemeR tid
```

Especificar la categoria

Igual que en l'altra pràctica, el codi fixa la categoria a una cadena buida. Per fer que es pugui especificar la categoria quan es crea un tema, només cal modificar el handler:

```
postThemesR = do
    user <- requireAuthId
    (pLeader, pCategory, pTitle, pDescription) <- getRequestJSON $ \
      ↪  obj -> do
        leader <- obj .: "leader"
        category <- obj .: "category"
        title <- obj .: "title"
        description <- obj .: "description"
        pure (leader, category, title, description)
    requireAdmin user
    let newtheme = Theme pLeader pCategory pTitle pDescription
    tid <- runDbAction $ addTheme newtheme
```

```
    getThemeR tid
```

Modificar el tema

A continuació, volem permetre que el leader del tema pugui modificar-lo, per tant en el mòdul `App` registrarem un handler PUT de la URL d'un tema:

```
dispatch = routing $
    -- RESTful API:
    --  URI: /themes
    route ( onStatic ["themes"] ) ThemesR
        [ onMethod "GET" getThemesR             -- get the theme list
        , onMethod "POST" postThemesR           -- create a new theme
        ] <||>
    --  URI: /themes/TID
    route ( onStatic ["themes"] <&&> onDynamic ) ThemeR
        [ onMethod1 "GET" getThemeR             -- get a theme
        , onMethod1 "PUT" putThemeR             -- modify a theme
        , onMethod1 "DELETE" deleteThemeR       -- delete a theme
        ] <||>
    --  URI: /themes/TID/questions
    route ( onStatic ["themes"] <&&> onDynamic <&&> onStatic
      ↪  ["questions"] ) ThemeQuestionsR
```

Aquest mètode rebrà un objecte amb propietats `category`, `title` i `description`, actualitzarà el tema, i retornarà el nou objecte THEME. El mètode estarà restringit al leader del tema.

Ara només cal implementar el handler `putThemeR`:

```
putThemeR :: ThemeId -> HandlerFor Forum Value
putThemeR tid = do
    user <- requireAuthId
    theme <- runDbAction (getTheme tid) >>= maybe notFound pure
    (pCategory, pTitle, pDescription) <- getRequestJSON $ \ obj -> do
        category <- obj .: "category"
        title <- obj .: "title"
        description <- obj .: "description"
        pure (category, title, description)
    requireLeader theme user
    let newtheme = Theme (tLeader theme) pCategory pTitle pDescription
    runDbAction (updateTheme tid newtheme)
    getThemeR tid
```

4 Frontend

A continuació implementem el frontend. La intenció és reusar les plantilles que hem fet per a la pràctica anterior, adaptant-les al sistema d'AngularJS. No comentarem el resultat en extensió, però sí explicarem l'estructura general. Els fitxers són:

```
frontend/
  index.html
  utils.js
  app.config.js
  app.module.js
  home-view/
    home-view.component.js
    home-view.module.js
    home-view.template.html
  question-view/
    question-view.component.js
    question-view.module.js
    question-view.template.html
  theme-view/
    theme-view.component.js
    theme-view.module.js
    theme-view.template.html
```

`index.html` és el fitxer principal, i està basat en la plantilla general (`default-layout.html`) de la pràctica anterior. Carrega els estils, AngularJS, el codi de l'aplicació, i el router:

```
<!DOCTYPE html>
<html ng-app="forumApp">
<head>
  <meta charset="utf-8">
  <link rel="stylesheet" type="text/css"
    ↪ href="https://stackpath.bootstrapcdn.com/.../bootstrap.min.css">
  <link rel="stylesheet" type="text/css"
    ↪ href="https://stackpath.bootstrapcdn.com/.../font-awesome.min.css">
  <title>DatForum</title>
  <script src="https://ajax.../.../angular.min.js"></script>
  <script src="https://ajax.../.../angular-route.min.js"></script>
  <script src="utils.js"></script>
  <script src="app.module.js"></script>
  <script src="app.config.js"></script>
  <script src="home-view/home-view.module.js"></script>
  <script src="home-view/home-view.component.js"></script>
```

```
  <script src="home-view/theme-view.module.js"></script>
  <!-- ... -->
</head>
<body ng-controller="forumApp">
  <nav class="navbar navbar-dark bg-primary">
    <a class="navbar-brand mb-0 h1 mr-auto" href="#!/">DatForum</a>

    <span ng-if="user" class="navbar-text mx-3"><i class="fa
      ↪ fa-user"></i> Usuari: <strong>{{ user.name
      ↪ }}</strong></span>
    <a ng-if="user" class="btn btn-primary my-1" href="{{ API_BASE
      ↪ }}/auth/logout"><i class="fa fa-sign-out"></i> Tanca
      ↪ sessió</a>
    <a ng-if="user == null" class="btn btn-primary my-1" href="{{
      ↪ API_BASE }}/auth/login"><i class="fa fa-sign-in"></i>
      ↪ Login</a>
  </nav>

  <div ng-view></div>
</body>
</html>
```

Després, a `utils.js` es defineix la base de la API REST i una funció d'utilitat per a formularis:

```
const API_BASE = 'forum-backend.cgi'

function makeFormHandler(action, success, reset=true) {
    const form = {
        requesting: false,
        // ...
    }
    return form
}
```

Després tenim el mòdul principal `forumApp` (que es defineix als fitxers `app.module.js` i `app.config.js`). Aquest mòdul és qui configura el router perquè renderitzi un dels tres components que es defineixen a continuació, segons la URL. També hi ha el controlador de la plantilla general (la navbar, bàsicament).

Aquests tres components s'han definit en carpetes i mòduls separats, seguint les bones pràctiques d'AngularJS. Dins de cada carpeta trobem:

- `nom.module.js`: la definició de mòdul.

- `nom.component.js`: la definició del component, que conté el codi del seu controlador.
- `nom.template.html`: la plantilla HTML per renderitzar el seu contingut.

L'adaptació de les plantilles és bastant sistemàtica i no la comentarem. Sí mostrarem el codi d'un dels components, per exemple el de la vista home (`home-view.component.js`):

```
angular.module('homeView').component('homeView', {
    templateUrl: 'home-view/home-view.template.html',
    controller: function HomeViewController($http, $location,
      ↪  $routeParams) {
        $http.get(`${API_BASE}/user`).then(response => {
            this.user = response.data.name ? response.data : null
        })
        const THEMES_URL = `${API_BASE}/themes`
        $http.get(THEMES_URL).then(response => {
            this.themes = response.data.items
        })

        this.createTheme = makeFormHandler(
            newtheme => $http.post(THEMES_URL, newtheme),
            response => $location.url(`/themes/${response.data.id}`),
        )
    }
})
```

Veiem com, en quant es carrega el component, demanem la informació de l'usuari i la llista de temes. També definim el codi del formulari d'afegir un nou tema. La resta es gestiona en la plantilla.

En conclusió; tot i que el frontend té el 100% de la funcionalitat de l'anterior, definitivament hi ha coses que és podrien fer per millorar la qualitat del codi. Per exemple, fer components més petits i reusables, o definir un *servei* per fer les crides a l'API, o cachejar la informació d'aquestes crides.

5 Bugs / millores a DatFw

El primer problema que ens hem trobat ha estat que quan des del frontend s'eliminen diverses preguntes, només s'acaba eliminant la primera. El codi que es comunica amb la API REST és aquest:

```
$q.all(Object.keys(qids).filter(k => qids[k]).map(
    qid => $http.delete(`${API_BASE}/questions/${qid}`).catch(x => x)
))
```

S'inicien les N peticions simultàniament i s'espera a que totes acabin. La primera es fa correctament, però les altres sovint fallen perquè aparentment els DELETE bloquegen la base de dades i SQLite genera un error:

```
Internal Server Error: SQLite3 returned ErrorBusy while attempting to
perform prepare "SELECT theme,user,posted,title,body FROM questions
WHERE id = ?": database is locked
```

Això és només el comportament per defecte, per canviar-lo es pot fer servir el mètode `sqlite3_busy_timeout()`, però com que el binding de Haskell no l'exposa, podem aconseguir el mateix efecte amb un PRAGMA. Per tant, només hem de modificar el mòdul `Model` definint una funció:

```
customOpen :: String -> IO ForumDb
customOpen path = do
    conn <- open path
    execute_ conn "PRAGMA busy_timeout = 400;"
    pure conn
```

I fent-la servir com a substitut de `open`. Ara les peticions concurrents haurien de funcionar millor.

Un altre problema és que, per exemple, el codi `permissionDenied "Això és una prova"` genera el missatge:

```
Permission denied: Això és una pro
```

El problema el trobem al mòdul `Develop.DatFw.Content`:

```
instance ToContent [Char] where
    toContent text =  ContentBuilder (stringUtf8 text) (Just (length
      ↪  text))

instance ToContent Text where
    toContent text =  ContentBuilder (encodeUtf8Builder text) (Just
      ↪  (T.length text))
```

S'està fent servir la quantitat de caràcters del text com a valor de `Content-Length`. En UTF-8, això només coincideix amb la quantitat de bytes quan tots els caràcters són ASCII.

Una solució és canviar `Just (length text)` per `Nothing` (és a dir, no especificar cap `Content-Length` com es fa amb la resta de contingut). Sino, també es pot executar el builder que es crea i extreure el tamany del `ByteString` produit.