UNIVERSITY OF SZEGED
Department of Software Engineering
UNIVERSITAS SCIENTIARUM SZEGEDIENSIS

Practice 6
REST + AUTH

Topics

- Authentication – protecting the REST endpoints
- Keywords: password saving, outsourcing authentication, logging
- Example, using the project we started last week

Authentication

- We need to handle authorities – difference between guests, users and admins
- Middleware chain: with a series of function calls we get from the raw request to sending the appropriate response
- Authentication check will be the first in the chain

Various techniques

- We can create our own solution – cookie, authentication data sent in every request, BUT ExpressJS has a better solution
- PassportJS + Express-Session – we parse the data sent with the POST request, if they are correct, we create a secure session for the client!

PassportJS usage

First steps:

```
app.use(bodyParser.urlencoded({'extended': 'true'}));
app.use(cookieParser());
```

With these, we will be able to use the data sent with the request, they will be parsed automatically by our app.

```
app.use(expressSession({secret: 'abcd'}));
```

This will protect our session (of course, we'll have to use a better secret)

```
app.use(passport.initialize());
app.use(passport.session());
```

Activates passport, and prepares it for handling persistent login session

Strategy

Passport supports the usage of different strategies for protecting REST endpoints – we can use local strategy (we solve the storing and querying of the authentication data) but OAuth, Auth0, OpenID, Reddit, Tumblr, Spotify, web tokens are also supported among over 300 strategies!

OAuth: one of the most popular current solutions, for authentication, we send the client to a trustworthy third party (the OAuth provider – PayPal, MySpace, Facebook for example) the authentication procedure is done there, we only have to work with the data provided by them.

```
passport.use('login',
new LocalStrategy.Strategy(function(username,password,
done) {
  if(username === 'larry' && password === '12345') {
    return done(null, username);
  } else {
    return done('ERROR', username); }
}));
```

Strategy 2

- The previous function will receive the request, and it's username, password attributes as arguments
- If the data is correct, the callback will fire null as the error value, if we have a problem, we signal it to the caller function

```
router.post('/login', function(req, res, next) {
  passport.authenticate('login', function(error, user) {
    if(error) {
      res.status(500).send('ERROR');
      …
  }(req, res, next); })
```

- This is how we call the strategy when a POST request arrives to the login route

Session

- If the login was successful, we create a session for the user

```
req.logIn(user, function(error) {
  if(error) {
    res.status(500).send('Request login failed');
  } else {
  return res.status(200).send('You are free to pass'); }
 })
```

- We serialize the user, the data (which we gave to the done callback) will be part of every request we receive from the client

```
passport.serializeUser(function(user, done) {
  done(null, user);
});
```

Session Out

```
req.isAuthenticated()
```

Checks if the received request is authenticated (every route after login can be protected like this)

```
req.logout();
```

This will call the deserializeUser function – after this isAuthenticated will return with false

```
passport.deserializeUser(function(user, done) {
  done(null, user);
});
```

For sessions to work, we have to implement the serializeUser and deseriallizeUser functions!

Other important rules

- **Authorization:** after we serialized the user, we can store in it's object the concrete rights (like admin rights)
- **Logging:** If a user logged in, logged out, did security critical operations, we have to log them at least to the console, or better to a file (the fs module from NodeJS is a good solution for this)
- **Password protection:** user password has to be stored in a hashed format (we'll do an example of this next week, when we start working with databases)