This work is licensed under CC BY 4.0

Access Control

User Authentication and Authorization in Express



Why is Access Control so Important?

- ✓ You use access control when there is a need to restrict users' access to a web application's resources.
- ✓ When performing access control, web applications authenticate the user and authorize the request to grant or prevent access to the requested resource.
  - **Authentication** is to verify a user's identity (usually with a username and password).
  - **Authorization** is the process of allowing an authenticated user to access a requested resource or not.



Det här fotot licensieras enligt CC BY-SA

Different Ways to Authenticate

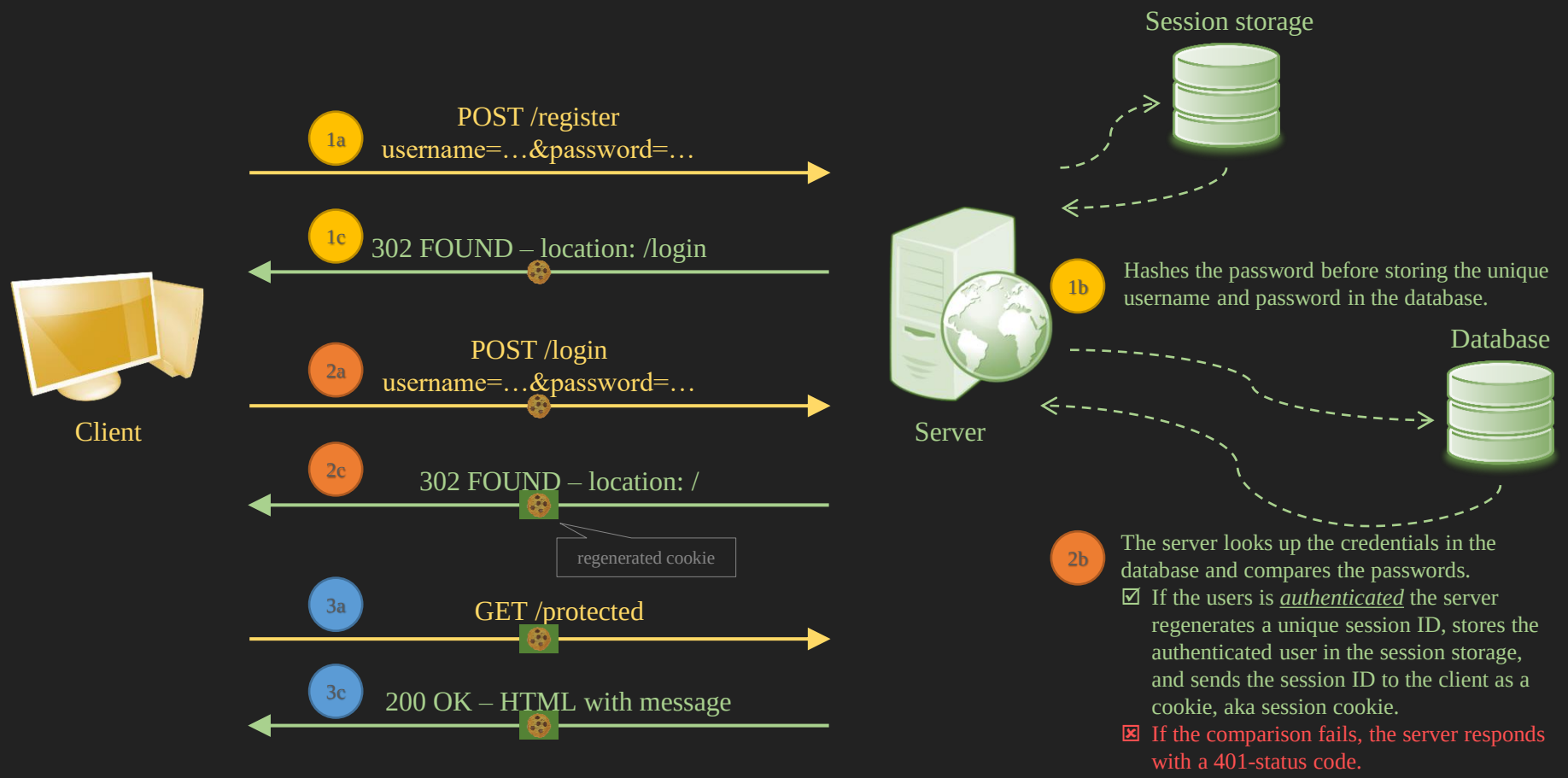
Authenticate === to verify the identity

- ✓ Cookie-based authentication
  - The server stores the session id.
- ✓ Token-based authentication
  - JSON Web Token (JWT)
  - The server does not store the token.
- ✓ Third-party access, OpenId Connect, SAML.

The authentication method you choose to use depends on the type of application. For a web application, cookie- or token-based authentication is often appropriate.



Cookie-Based Authentication



Password Security (for the User)



- **Passphrase** instead of a password?
- NEVER use the same password on multiple sites!
- Always turn on two-factor authentication (2FA) if it is an option.
- Do not try to remember your passwords. Use a password manager.
  - A password should be ten characters long, minimum. The longer, the better.
  - Using numbers and/or special characters does not make a password stronger.
- Never give an honest answer to a password recovery question.
  - Generate a random password and store it in your password manager.

Password Security (for the Server)

- ✓ **Managing passwords, as well as all other aspects related to user information, is a complex and risky task!**
- ✓ Send passwords only over HTTPS.
- ✓ Passwords should be stored in a hashed format using strong cryptographic algorithms.
  - A password should never be able to be read back!
  - Always hash a password using a secure hashing algorithm (Argon2, bcrypt, scrypt) before storing it.
    - ***Avoid*** MD5, SHA1, SHA2, SHA3, SHA256, etc., because they are not safe password hashing algorithms.
  - Allow looooong passwords.



Rainbow Tables

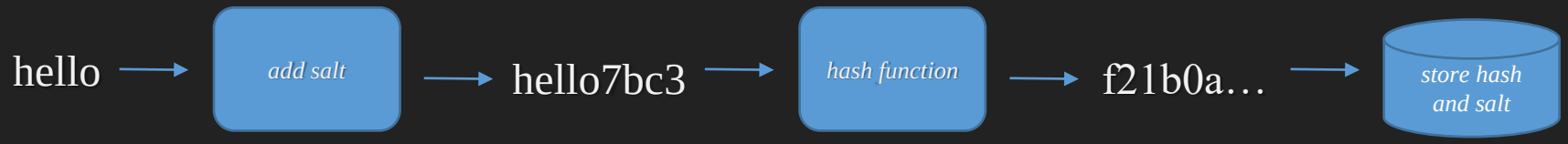
- A rainbow table is a precomputed table with hashed "known passwords" for cracking passwords.

Plain Text	Hash (MD5)
123456	e10adc3949ba59abbe56e057f20f883e
123456789	25f9e794323b453885f5181f1b624d0b
password	5f4dcc3b5aa765d61d8327deb882cf99
qwerty	d8578edf8458ce06fbc5bb76a58c5ca4
111111	96e79218965eb72c92a549dd5a330112



- A rainbow table attack is easily prevented by using salt techniques.

Salting



	username	salt	salted and hashed password
abc123	john	`wer074`	`f18d825b03e598d1293b3ddb3f7be945`
	mike	`jkl632`	`5df3853f391f80d6e6cfdfea8c9a4eaa`
xyz456	jane	`sdf864`	`bdbcefdfddb45cb39b49dbaa506754a4`

- ✓ Rainbow tables become useless by adding a small piece of random data, called salt, to the password before it is hashed.
- ✓ The same password has different hashes if the salt is unique to each user.

Authorize on the Server

- ✓ The server must perform the authorization because the request coming from a client cannot be trusted.
  - It is not sufficient to hide the delete button from a user who cannot delete something at the server.
- ✓ Your authorization process should always, in a perfect world, prevent access to requested resources by default.
  - It is safer to prevent access by default and override any requests that do not require permission.
- ✓ Resource authorization can be complicated because it validates whether or not a user can access a specific resource.
  - A user should be able to modify their profile and only their profile.
  - It is easy to allow access to a resource if the user needs to be logged in.

How to Manage Registration

- ✓ The user submits a registration form, and the credentials are stored in a MongoDB database using a Mongoose schema.
- ✓ Make sure the username is unique.
- ✓ Set the minimum length of passwords.
- ✓ Salt and hash password using bcrypt.
- ✓ Applying the principle of separation of concerns is a good idea by adding static methods to the Mongoose schema to use when registering users.

User Model

```
const schema = new Schema({
  username: {
    type: String,
    minlength: [3, 'The username must be of minimum length 3 characters.’],
    maxlength: [256, 'The username must be of maximum length 256 characters.'],
    required: true,
    unique: true,
    trim: true
  },
  password: {
    type: String,
    minlength: [10, 'The password must be of minimum length 10 characters.’],
    maxlength: [256, 'The password must be of maximum length 256 characters.'],
    required: true
  }
})
```

- ✓ The username must be unique.
- ✓ The length of the password is validated before salting and hashing.

Hashing the Password

- ✓ The save method triggers validation.
- ✓ Using a pre hook you can execute code after the validation and before the data is written to the database.
- ✓ You must declare `pre()` before creating the model.

```
// Salt and hash the password, if changed, before saving.
schema.pre('save', async function () {
  if (this.isModified('password')) {
    this.password = await bcrypt.hash(this.password, 10)
  }
})

// Create a model using the schema.
export const User = mongoose.model('User', schema)
```

- ✓ The salt and hash is auto-generated.

The Hashed Password



- A bcrypt hash has three fields, delimited by the "$" character.
  - `2a` identifies the bcrypt algorithm version that was used.
  - `10` is the cost; 2^{10} iterations of the key derivation function are used (some recommend a cost of 12 or more).
  - `/ZADTNI3LjfDSEYbdErZAOiw.M1NwS01ohmlM8JywWHLClz1ALLmK` is the salt and the ciphertext.

How to Manage Login

- ✓ The user submits a login form, and the credentials are compared with the data in the database.
  - Create a new session cookie, store user data in session store, and redirect.
  - If the authentication fails show a custom flash error message or send the status code 401 Unauthorized.
  - Applying the principle of separation of concerns is a good idea by adding static methods to the Mongoose schema to authenticate users.
- ✓ (Do not forget to give the user a possibility to log off!)

Authenticating a User (1 of 2)

```
accountController.loginPost = async (req, res, next) => {
  try {
    const userDoc = await User.authenticate(req.body.username, req.body.password)
    req.session.regenerate((err) => {
      if (err) {
        throw new Error('Failed to re-generate session.')
      }

      // Store the authenticated user in the session store.
      ...

      // Redirect.
      ...
    })
  } catch (error) {
      // Redirect to the login form and display an error message.
      ...
  }
}
```

- ✓ Regenerate a session cookie, store user data in the session store, and redirect.
- ✓ If the authentication fails redirect to the login page and show an error message or send the status code 401.

Authenticating a User (2 of 2)

```
schema.statics.authenticate = async function (username, password) {
  const userDoc = await this.findOne({ username })

  // If user found, compare the supplied password with the stored password
  // hash; otherwise, use a dummy hash to prevent timing attacks.
  const dummyPasswordHash = await bcrypt.hash(password, 10)
  const suppliedPasswordHash = userDoc ? userDoc.password : dummyPasswordHash
  const passwordMatch = await bcrypt.compare(password, suppliedPasswordHash)

  // If no user found or password is wrong, throw an error.
  if (!userDoc || !passwordMatch) {
    throw new Error('Invalid login attempt.')
  }

  // User found, and password correct, return the user.
  return userDoc
}
```

- ✓ Lookup the user document in the database and compare, using a “constant-time” algorithm, the provided password with the hashed one from the database.
  - If the authentication fails, throw an error, otherwise, return a document representing the user.

How to Authorize

- ✓ When defining a route, you can provide multiple callbacks. You can use this mechanism to authorize the requested resource.

```
router.post('/create',
  authorizeLoggedOn,
  (req, res, next) => controller.create(req, res, next)
)
```

```
const authorizeLoggedOn = (req, res, next) => {
  if (/* authorization logic here */) {
    const error = new Error('Forbidden')
    error.status = 403
    next(error)
    return
  }

  next()
}
```

next()

next(error)

