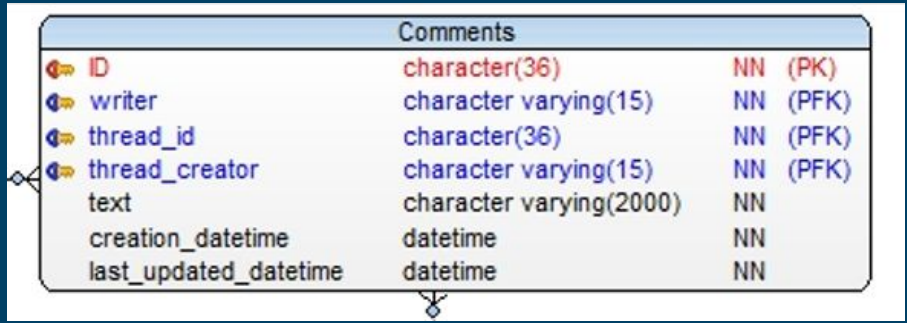
Group 22 Independent study: crypto module

- Documentation and source: https://nodejs.org/api/crypto.html
- Is a built-in Node.js module (i.e., don’t need to use npm install)
- Provides cryptography-related functions
- In our project, we used crypto for UUID generation, random salt generation, and password hashing server-side

Liam Turcotte, Tommy Turcotte, Samuel Bazinet, Guillaume Flores

UUID Generation

- Records in some of our database entities (Threads, Comments, and Notifications) cannot be identified uniquely using composite primary keys with multiple regular attributes
- Consequently, these tables store a 36-character string PK (ID) to distinguish rows
- These strings are generated using crypto's randomUUID() function
- UUID example: z3gf55f5-945e-41d0-8fe1-ebe3b8c54ee5



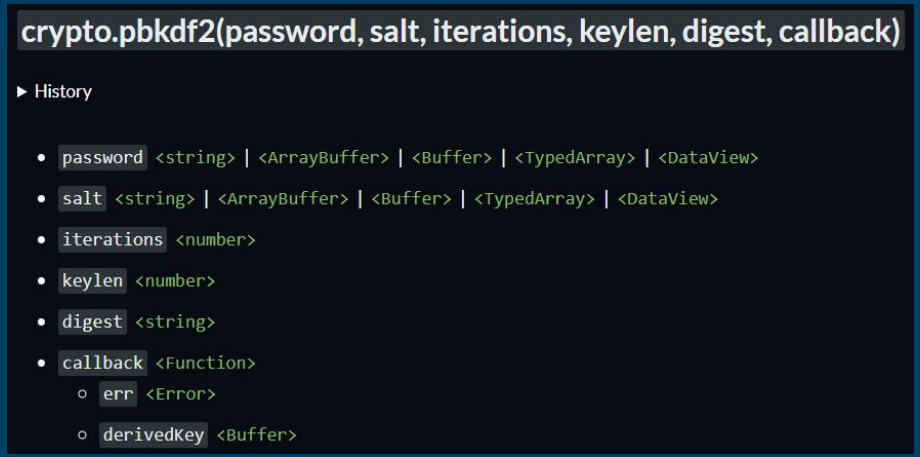
Comments entity in our database

```
let commentFields = {
    'id': crypto.randomUUID(),
    'username': userLoggedIn,
    't_id': fields.ID,
    't_creator': fields.creator,
    'text': fields.text,
    'c_dt': currentDateTime,
    'last_updated_dt': currentDateTime
};
```

backend/biz/forum.js lines 96-104

Password hashing

- Storing plaintext passwords in the database is a major security concern; as such, passwords are hashed and the hash values are stored instead
- We use crypto's asynchronous pbkdf2() function for hashing passwords in our backend
- Takes in password string, salt value, number of iterations, hash length, algorithm, and a callback function as arguments



crypto.pbkdf2(password, salt, iterations, keylen, digest, callback)

▸ History

- `password` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `salt` `<string>` | `<ArrayBuffer>` | `<Buffer>` | `<TypedArray>` | `<DataView>`
- `iterations` `<number>`
- `keylen` `<number>`
- `digest` `<string>`
- `callback` `<Function>`
  - `err` `<Error>`
  - `derivedKey` `<Buffer>`

Function definition from the documentation https://nodejs.org/api/crypto.html#cryptopbkdf2password-salt-iterations-keylen-digest-callback

Salts

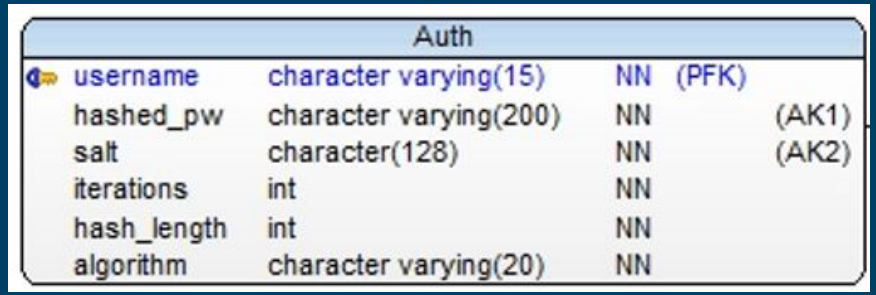
```
function generateSalt(saltLength) {
    // generate salt
    return crypto.randomBytes(saltLength).toString('base64');
}
```

We use crypto's randomBytes() function to generate random salts; this is used for registration. See backend/biz/auth.js

- A critical component of crypto's pbkdf2() hashing function which made it more favorable to us than other hashing algorithms for our server (e.g., MD5) is that is uses a salt value
- There are two main reasons why we wanted a function that uses a salt:
  - The function will produce different hashes for identical passwords given that the salt values are different
  - Much more resistant to brute force attacks
- Aside: we use MD5 client-side in order to not send plaintext passwords over the network because of its simplicity and to decrease the number of requests to the backend (by not having to fetch salt, iterations, hash length, and algorithm)
  - This hash is then re-hashed and compared to a database record by the backend

Integration into our system - Login

- Salt, iterations, hash length, and algorithm are all tied to each username/hashed password entry so that if any of these values get changed, it doesn't break old auth data
- Login takes in the (MD5-hashed) password from the frontend, computes its hash using crypto.pbkdf2() argument values for that user, and compares the result to user's hashed_pw field
- Based on the request and the outcome of the computation, a 200, 401, 403, or 500 response will be sent to the client



Auth			
username	character varying(15)	NN	(PFK)
hashed_pw	character varying(200)	NN	(AK1)
salt	character(128)	NN	(AK2)
iterations	int	NN	
hash_length	int	NN	
algorithm	character varying(20)	NN	

Auth entity in our database. Note: the DDL SQL code to initialize our database is located in backend/model/init.js

```
function computePwHash(pw, salt, iterations, hashLength, algo, send) {
    // computes hash for pw entered
    crypto.pbkdf2(pw, salt, iterations, hashLength, algo, (error, hashedPw) => {
        if (error) {
            console.error('Issue hashing password: ', error);
            send(null);
        } else {
            send(hashedPw.toString('hex'));
        }
    });
}
```

Wrapper for the crypto.pbkdf2() function; used by register and login system functions. Located at backend/biz/auth.js lines 13-23

Integration into our system - Register

- Register uses the most recent/currently set salt, iterations, hash length, and algorithm values to hash the (MD5-hashed) password from the frontend using our wrapper function for crypto.pbkdf2()
- Afterwards, if there were no problems with the computation, the following occurs:
  - A new record is added to the Users entity containing new user information
  - A new record is inserted into the Auth entity containing the values passed into the hashing algorithm and the hashed password itself
    - This is the succeeding step since username is a PFK in the Auth table and PK in Users