Explain basic security threads like: Cross Site Scripting (XSS), SQL Injection and whether something similar to SQL injection is possible with NoSQL databases like MongoDB, and DOS-attacks. Explain/demonstrate ways to cope with these problem

XSS enables attackers to inject client-side scripts into web pages viewed by other users.
A cross-site scripting vulnerability may be used by attackers to bypass access controls such as the same-origin policy.
Cross-site scripting carried out on websites accounted for roughly 84% of all security vulnerabilities documented by Symantec as of 2007.
Their effect may range from a petty nuisance to a significant security risk, depending on the sensitivity of the data handled by the vulnerable site and the nature of any security mitigation implemented by the site's owner.

A way to prevent or at least limit this is to:

1. Encode every data that is given by a user. If data is not given by a user but supplied via the GET parameter, encode these data too.
   Even a POST form can contain XSS vectors. So, every time you are going to use a variable value on the website, try cleaning for XSS.
   These are the main data that must be properly sanitized before being used on your website:
   The URL, HTTP referrer objects, GET parameters from a form, POST parameters from a form, Window.location, Document.referrer,
   document.location, document.URL, document.URLUnencoded, cookie data, headers data and database data, if not properly validated on user input.
2. Sanitize HTML markup with a Library designed for the job.
3. Prevent DOM-based XSS

SQL Injection is an attack method where hacker insert malicious SQL code into a Web form to gain access to resources, applications or databases. It has been on the rise with the advancement of automated exploit tools, and the attack method, which can enable data manipulation and the spread of malware, is becoming more advanced and popular among attackers.
An attacker uses SQL injection to manipulate a site's Web-based interfaces and force the database to execute undesirable SQL code, enabling data manipulation and spreading malware.

In order to prevent these types of attacks:

1. Enterprises must implement secure coding best practices and limit Web application coding privileges.
   Ensure employee security awareness: Make sure that employees who have a hand in website development (as well as dedicated Web developers) are aware of the SQL injection threat and know best practices to keep your servers safe.
2. Reduce debugging information.
   When a Web server experiences an error, make sure details of the error aren't displayed to the user, since this information could help a hacker commit malicious activity and gain the information he or she needs to successfully attack the server.
3. Test Web applications regularly.
   Test Web applications and check Web developers work by sending data through the Web server; if the result is an error message, the application might be susceptible to an SQL injection attack.

SQL Injection is it possible with NoSQL databases like MongoDB?

NoSQL definitely does not imply zero risk. In fact, NoSQL databases are vulnerable to injection attacks, cross-site request forgery (CSRF) and other vulnerabilities.
One of the common reasons for a SQL injection vulnerability is building the query from string literals which include user input without using proper encoding. The JSON query structure makes it harder to achieve in modern data stores like MongoDB. Nevertheless it is still possible.

Ex. Let us examine a login form which sends its username and password parameters via an HTTP POST to the backend which constructs the query by concatenating strings.

string query = "{ username: '" + post_username + "', password: '" + post_password + "' }"

But with malicious input this query can be turned to ignore the password and login into a user account without the password, here is an example for malicious input:

username=tolkien', $or: [{}, { 'a':'a&password=' }], $comment:'successful MongoDB injection'

This attack will succeed in any case the username is correct, an assumption which is valid since harvesting user names isn't hard to achieve.
The password becomes a redundant part of the query since an empty query {} is always true and the comment in the end does not affect the query.
Another reason is that Javascript execution exposes a dangerous attack surface if un-escaped or not sufficiently escaped user input finds its way to the query.
Furthermore the REST API can expose the NoSQL database to CSRF attacks allowing an attacker to bypass firewalls and other perimeter defenses.

In order to prevent these types of injection attacks:

1. Security scanning to prevent injections -
   It is recommended to use out of the box encoding tools when building queries. For JSON queries in MongoDB have good native encoding which will terminate the injection risk. It is also recommended to run Dynamic Application Security Testing (DAST) and static code analysis on the application in order to find any injection vulnerabilities if coding guidelines were not followed
2. REST API exposure
   To mitigate the risks of REST API exposure and CSRF attacks, there is a need to control the requests limiting their format.
   *It is here important to notice that databases like MongoDB have many third party REST API's which are encouraged by the main project, some of these are really lacking in the security measures we described here.
3. Access Control and Prevention of Privilege Escalation
   Utilizing proper authentication and role management mechanisms is important for two reasons. First, they allow enforcing the principle of least privilege thus preventing privilege escalation attacks by legitimate users. Second, similarly to SQL injection attacks, proper privilege isolation allows to minimize the damage in case of data store exposure via the above described injections.
   This can be done by making the data accessible via a web application authorized with a "user" role, while the sensitive entries require the "admin" role, which is never granted via the web interface. This allows scoping the damage in case of attack, ensuring that no administrators' data is leaked.

DOS-attacks
Denial of Service (DoS) are attacks against web sites occur when an attacker attempts to make the web server, or servers, unavailable to serve up the web sites they host to legitimate visitors. For some time, it was thought that these types of attacks were generally used against large corporations, government sites, and activist sites as a form of protest to disrupt their web presence.
Denial of Service attacks can result in significant loss of service, money and reputation for organizations. Typically, the loss of service is the inability of a particular network service, such as e-mail, to be available or the temporary loss of all network connectivity and services. An HTTP Denial of Service attack can also destroy programming and files in affected computer systems.
In some cases, HTTP DoS attacks have forced Web sites accessed by millions of people to temporarily cease operation.

In order to reduce the risk of this type of attack:

1. Purchase a lot of bandwidth.
   This is not only the easiest solution, but also the most expensive. If you simply have tons of bandwidth, it makes perpetrating a DoS attack much more difficult because it's more bandwidth that an attacker has to clog.
2. Use DoS attack detection technology.
   Intrusion prevention system and firewall manufacturers now offer DoS protection technologies that include signature detection and connection verification techniques to limit the success of DoS attacks.
3. Prepare for DoS response.
   The use of throttling and rate-limiting technologies can reduce the effects of a DoS attack. One such response mode stops all new inbound connections in the event of a DoS attack, allowing established connections and new outbound connections to continue.

DoS protection is more “art” than science, requiring a combination of techniques to limit the impact of such an attack on your organization.

Explain and demonstrate ways to protect user passwords on our backend, and why this is necessary.

In order to store a password in a safe way, and to make sure that only the user knows its password, we need to encrypt the password and never, ever, store it in plaintext, because if we do, we make personal information vulnerable to being stolen by e.g. simple SQL injections.

To encrypt the password we could use a One-way Hash Function. A one-way hash function is an algorithm that turns messages or text into a fixed string of digits, called a cryptographic hash value (or the Digest). So when a user logs in, the server takes the digest of the password-input by the user and compares it to the original digest stored in the database, which ensures that passwords never occur in plaintext.

Example: see PasswordHashing.js

Explain about password hashing, salts and the difference between bcrypt and older algorithms like sha1, md5 etc.

A one-way hash function is an algorithm that turns messages or text into a fixed string of digits, called the Cryptographic hash value or the Digest).
A one-way hash function takes variable-length input ex. a message of any length (even thousands or millions of bits) and produces a fixed-length output; say, 160-bits. This is great for protecting passwords, because we want to store passwords in a form that protects them even if the password file itself is compromised, but at the same time, we need to be able to verify that a user's password is correct.

Ex. hash("hello") = 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824

The hash function ensures that, if the information is changed in any way—even by just one bit—an entirely different output value is produced.
One way indicates that it's (almost) impossible to derive the original text given the digest.
One-way hashing is though not enough for passwords, because although the algorithm makes a unick kode, anyone can easily translate this hashed code just by using a link like: https://crackstation.net/ Therefore we have to use salts as well as hashing by employ salted password hashing.

Here you can see how to make password hashing in Node:
https://www.npmjs.com/package/password-hash

Salt
In cryptography, a salt is random data that is used as an additional input to a one-way function that "hashes" a password or passphrase.
The primary function of salts is to defend against dictionary attacks versus a list of password hashes and against pre-computed rainbow table attacks.
A new salt is randomly generated for each password. In a typical setting, the salt and the password are concatenated and processed with a cryptographic hash function, and the resulting output (but not the original password) is stored with the salt in a database. Hashing allows for later authentication while protecting the plaintext password in the event that the authentication data store is compromised.
In short terms: A salt is a random string that is appended or prepended to the password before it is hashed. This randomizes the same password completely each time, as the string is different. Se an example below:

hash ("hello" + "QxLUF1bgIAdeQX") =
9e209040c863f84a31e719795b2577523954739fe5ed3b58a75cff2127075ed1
hash ("hello" + "bv5PehSMfV11Cd") =
d1d3ec2e6f20fd420d50e2642992841d8338a314b8ea157c9e18477aaef226ab

The salt does not need to be secret. Just by randomizing the hashes, an attacker won't know in advance what the salt will be, so they can't pre-compute. If each user's password is hashed with a different salt, the reverse lookup table attack won't work either.

The salt should not be too short. Otherwise an attacker can build a lookup table for every possible salt. A good rule of thumb is to use a salt that is the same size as the output of the hash function.
Furthermore the username shouldn't be used as a salt. Usernames may be unique to a single service, but they are predictable and often reused for accounts on other services. An attacker can build lookup tables for common usernames and use them to crack username-salted hashes.

Cryptographic salts are broadly used in many modern computer systems, from Unix system credentials to Internet security.

Bcrypt is an algorithm that uses Blowfish internally. It is not an encryption algorithm itself. It is used to irreversibly obscure passwords, just as hash functions are used to do a "one-way hash". It does this by using Blowfish to encrypt a magic string, using a key "derived" from the password. Later, when a user enters a password, the key is derived again, and if the ciphertext produced by encrypting with that key matches the stored ciphertext, the user is authenticated. The ciphertext is stored in the "password" table, but the derived key is never stored.

The reason why this is so secure is that in order to break the cryptography here, an attacker would have to recover the key from the ciphertext. This is called a "known-plaintext" attack, since the attack knows the magic string that has been encrypted, but not the key used. Blowfish has been studied extensively, and no attacks are yet known that would allow an attacker to find the key with a single known plaintext.

MD5 and SHA1 are general purpose hash functions, designed to calculate a digest of huge amounts of data in as short a time as possible.
This means that they are fantastic for ensuring the integrity of data and utterly rubbish for storing passwords.

The difference between older versions like MD5 and SHA1 and bcrypt is that while bcrypt is far slower. This is also what makes MD5 and SHA1 weak, because they are so fast.
A modern server can calculate the MD5 hash of about 330MB every second. If your users have passwords which are lowercase, alphanumeric, and 6 characters long, you can try every single possible password of that size in around 40 seconds. With bcrypt instead of the server cracking a password every 40 seconds, I'd be cracked in 12 years or even not at all. This is illustrated in the example below.

Another difference is that MD5 and SHA1 are hashing algorithms, where as bcryp uses Blowfish (to make an encryption algorithm).

The most important difference is that bcrypt has not been haked where as both sha1 & md5 have been.

Ex. Of hashing the word: password.

MD5 76c8245727d68a781e98288f147184b5:5b0f4bf8fec9b53879a5bb929172c153

SHA1 fecbb8d3eb7c9af4fcf61a8722a52d84428ab519:7ac4cd7db7a9d9d7abb5c7ebb46aa9265c941f87

BCrypt $2a$10$ppsVmqeIhoD.XjE1Zdmj4.fjH2HZblKGycmpgU3SA2UB4cbo4nPLG

Hash rate - Using oclHashcat-Plus I ran a check against each to see the hash rate. Unfortunately due to a driver error, I couldn't run a check against a Bcrypt hash, so I used the weaker md5crypt instead.

Hashing Mechanism	Tries per second
MD5	5,357,000
SHA1	3,673,000
md5Crypt	12114

CPU Based Cracking - The numbers below are using the CPU to crack the exact same hashes as before.

Hashing Mechanism	Time to generate
MD5	0.008s
SHA1	0.012s
BCrypt (cost 10)	0.012s

Explain about JSON Web Tokens (jwt) and why they are very suited for a REST-based API

In authentication, when the user successfully logs in using his credentials, a JSON Web Token will be returned and must be saved locally (typically in local storage, but cookies can be also used).
Whenever the user wants to access a protected route or resource, the user agent should send the JWT, typically in the Authorization header using the Bearer schema. The content of the header could look like the following: `Authorization: <token>`
This is a stateless authentication mechanism as the user state is never saved in server memory. The server's protected routes will check for a valid JWT in the Authorization header, and if it's present, the user will be allowed to access protected resources. This allows you to fully rely on data APIs that are **stateless** and even make requests to downstream services.