1. Explain basic security terms like authentication, authorization, confidentiality, integrity, SSL/TLS and provide examples of how you have used them.

- **Authentication**
  Confirming the identity of a client (via some kind of login procedure).
- **Authorization**
  Determining whether an authenticated client is allowed to receive a service or perform an operation.
- **Confidentiality (privacy)**
  Protection from disclosure to unauthorized persons.
- **Integrity**
  Maintaining data consistency (data cannot be modified).
- **SSL/TSL**
  Secure **S**ockets **L**ayer and **T**ransport **L**ayer **S**ecurity are used to authenticate servers and clients across untrusted networks and encrypt data sent between authenticated parties.

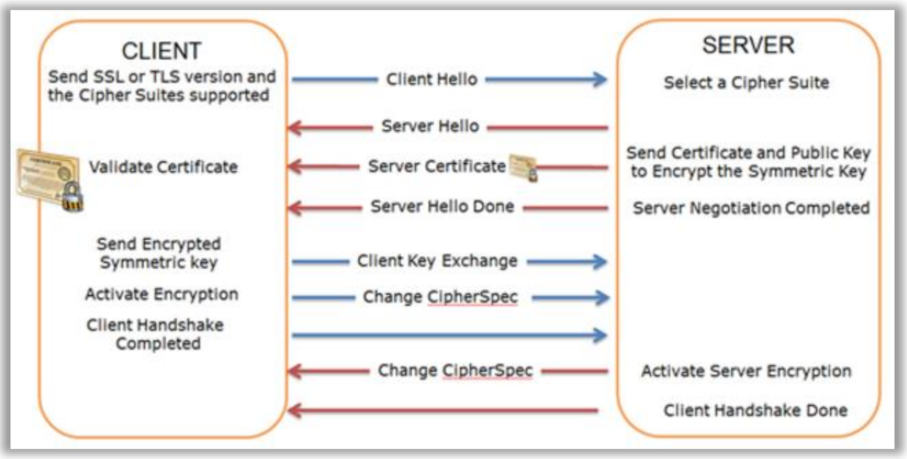
2. Explain basic security threads like: Cross Site Scripting (XSS), SQL Injection and whether something similar to SQL injection is possible with NoSQL databases like MongoDB, and DOS-attacks. Explain/demonstrate ways to cope with these problems.

- **Cross Site Scripting (XXS)**
  This is when an attacker injects malicious client-side code (scripts) into web pages viewed by other user.
- **SQL Injection**
  SQL injections is possible if an application hasn't got sufficient security around its database. It is used to get access to and/or manipulate data by injecting SQL queries.
- **Denial of Service**
  A DDOS attack is when a server is hit with so many request at the same time (normally through a botnet), that it becomes unavailable to the intended users or crashes completely.

SQL injections often ends with either stolen or lost data, but it is possible to secure yourself quite well by following a few guidelines while coding your application:
- Use Prepared Statements.
- Escaping all User Supplied Input.

And always consider:
- Minimizing privileges to accounts.
- Whitelist Input Validation.

3. Explain, at a fundamental level, the technologies involved, and the steps required initialize a SSL connection between a browser and a server and how to use SSL in a secure way.


CLIENT
Send SSL or TLS version and the Cipher Suites supported
Validate Certificate
Send Encrypted Symmetric key
Activate Encryption
Client Handshake Completed
Client Hello
Server Hello
Server Certificate
Server Hello Done
Client Key Exchange
Change CipherSpec
Change CipherSpec
SERVER
Select a Cipher Suite
Send Certificate and Public Key to Encrypt the Symmetric Key
Server Negotiation Completed
Activate Server Encryption
Client Handshake Done

1. Once the client has established a TCP session on port 443 with the server, the client sends a client hello message. This client hello includes information such as the cipher suites that it supports.
2. The server selects the cipher suite from the list presented and responds with a server hello indicating to the client the ciphers it deems suitable. The client and the server have now agreed on a cipher suite to use.
3. The server then issues the client a copy of its certificate (remember that this certificate also contains the public key). Optionally, the server may request a copy of the client's certificate if client-side authentication is required.
4. Next, the server sends a server hello done message to tell the client it has completed the first phase of the session setup. As there is no key yet, this process is carried out in clear text.
5. The client now generates a random number, encrypts it with its public key, and sends the server the client key. This process is known as the client key exchange. This is the symmetric key that will be used for the duration of the symmetric encryption session. Communication from here on is encrypted.
6. The client now sends a change cipher spec message to the server to say it will now begin using the negotiated cipher suite (determined in step 2) for the duration of the session.
7. Once this is done, the client sends a finished message to the server to say that it is ready.
8. The server, in turn, sends a change cipher spec message to the client using the agreed information. The server also sends out a finished message on completion.
9. A secure encrypted tunnel is now set up, and communication can begin using the symmetric encryption details negotiated.

Source: http://www.informit.com/articles/article.aspx?p=169578&seqNum=4

4. 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.

5. Explain about password hashing, salts and the difference between bcrypt and older (not recommended) algorithms like sha1, md5 etc.

Say your password is "baseball". I could simply store it in plaintext, but anyone who gets my database gets the password. So instead I do an SHA1 hash on it, and get this:
`a2c901c8c6dea98958c219f6f2d038c44dc5d362`

Theoretically it's impossible to reverse a SHA1 hash. But go do a google search on the exact string, and you will have no trouble recovering the original password.
Plus, if two users in the database have the same password, then they'll have the same SHA1 hash. And if one of them has a password hint that says `try "baseball"` -- well now I know what *both* users' passwords are.
So before we hash it, we prepend a unique string. Not a *secret*, just something unique.
Let's say `WquZ012C`. So now we're hashing the string `WquZ012Cbaseball`. That hashes to this:
`c5e635ec235a51e89f6ed7d4857afe58663d54f5`
Googling that string turns up nothing, so now we're on to something. And if person2 also uses "baseball" as his password, we use a different salt and get a different hash.
Of course, in order to test out your password, you have to know what the salt is. So we have to store that somewhere. Most implementations just tack it right on there with the hash, usually with some delimiter.

Instead of hashing the password yourself using md5, sha1 or whatever, you'll want to run an *iterative* process which runs the hash function thousands of times.
This can be done by using:

- **crypt**
  The original password hashing mechanism (DES-based) is horribly insecure; don't even consider it. The MD5-based is better, but still shouldn't be used today. Later variations, including the SHA-256 and SHA-512 variations should be reasonable. All recent variants implement multiple rounds of hashes.
- **bcrypt**
  The blowfish version of the `crypt` functional call mentioned above. Capitalizes on the fact that blowfish has a very expensive key setup process, and takes a "cost" parameter which increases the key setup time accordingly.

Source: http://security.stackexchange.com/a/51983

6. 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.

Source: https://jwt.io/introduction/

7. Explain and demonstrate a system using jwt's, focusing on both client and server side.

Server

For all request to endpoint /api, the server will run "passport.authenticate" to authenticate the user.

```
app.use('/api', function (req, res, next) {
    passport.authenticate('jwt', {session: false}, function (err, user, info) {
        if (err) {
            res.status(403).json({mesage: "Token could not be authenticated",
fullError: err})
        }
        if (user) {
            return next();
        }
        return res.status(403).json({mesage: "Token could not be authenticated",
fullError: info});
    })(req, res, next);
});
```

Checks incoming jwt from client against jwtConfig. Also checks that user exists in db.

```
module.exports = function (passport) {

    var opts = {};
    opts.secretOrKey = jwtConfig.secret;
    opts.issuer = jwtConfig.issuer;
    opts.audience = jwtConfig.audience;
    opts.jwtFromRequest = ExtractJwt.fromAuthHeader();
    // opts.jwtFromRequest = ExtractJwt.fromAuthHeaderWithScheme("Bearer");
    passport.use(new JwtStrategy(opts, function (jwt_payload, done) {
        console.log("PAYLOAD: " + jwt_payload);
        User.findOne({username: jwt_payload.sub}, function (err, user) {
            if (err) {
                return done(err, false);
            }
            if (user) {
                done(null, user); //You could choose to return the payLoad instead
            }
            else {
                done(null, false, "User found in token not found");
            }
        })
    }))
};
```

Client

When submit() is invoked, save token from server in a session:

(`$window.sessionStorage.token = data.token;`).

The factory "authInterceptor" will intercept every outgoing http request and add an authorization header with the saved token:

(`config.headers.Authorization = $window.sessionStorage.token;`).

8. Explain and demonstrate use of the npm passportjs module

Not answered.

9. Explain, at a very basic level, OAuth 2 + OpenID Connect and the problems it solves.

Not answered.

10. Demonstrate, with focus on security, a proposal for an Express/Mongo+Angular-seed with built in support for most of the basic security problems, SSL and ready to deploy on your favourite Cloud Hosting Service.

Not answered.