ELABORATO PER LA PROVA DI ESAME

INDIRIZZO: ITIA - INFORMATICA E TELECOMUNICAZIONI
ARTICOLAZIONE "INFORMATICA"
ANNO SCOLASTICO 2019-2020
Discipline: INFORMATICA E SISTEMI E RETI

1 TRACCIA

Progettare un sistema per in grado di supportare le persone, in particolare anziane, nella gestione quotidiana delle cure mediche.
In particolare, fatte le opportune ipotesi, si chiede di descrivere:

- Il progetto dell'infrastruttura tecnologica ed informatica necessaria a gestire il sistema, dettagliando:
  - L'architettura di rete e le caratteristiche dei sistemi server;
  - Le modalità di comunicazione;
  - Gli elementi per garantire la sicurezza del sistema;
- L'architettura dei dati e dei software che si intende implementare, dettagliando il modello dei dati e le modalità di interazione;
- Una parte significativa dell'applicazione.

2 SOMMARIO

3 SOLUZIONE

3.1 ANALISI

Dal testo si evince che la soluzione deve essere indirizzata in particolar modo verso persone anziane, quindi in generale con meno dimestichezza nell'uso delle nuove tecnologie. Propongo quindi la creazione di un dispositivo "intelligente" partendo da un semplice porta-medicine a slot, al quale vengono aggiunti dei led, uno per slot, che si illumineranno di verde quando il medicinale contenuto deve essere assunto, e un buzzer, che avvertirà con un segnale acustico quando si è in ritardo con un'assunzione.

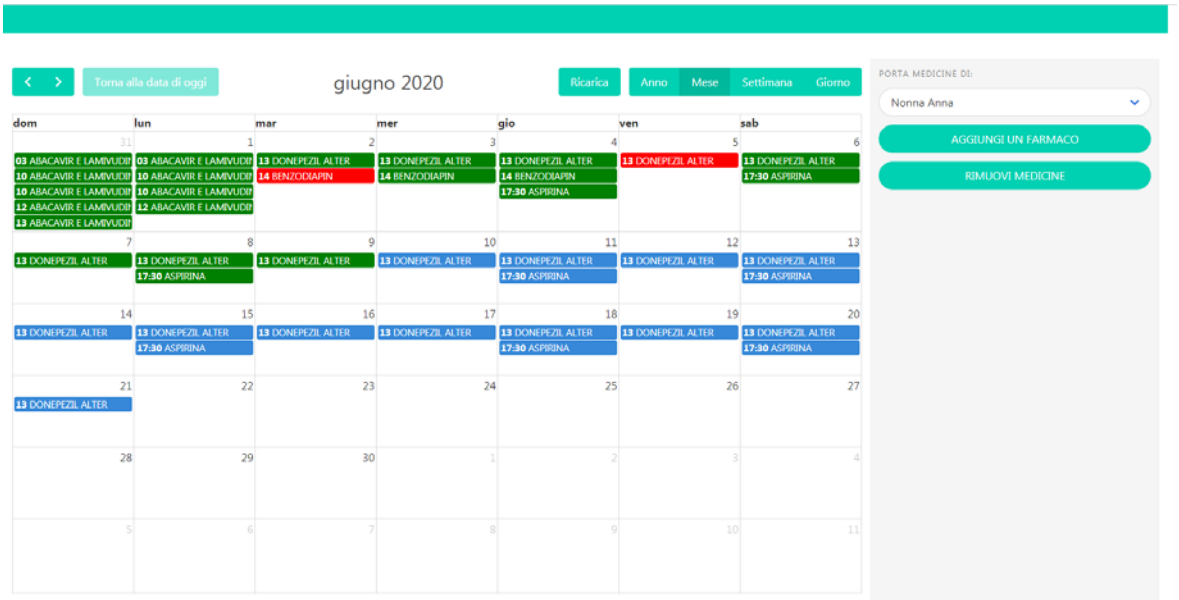
Il dispositivo dovrà quindi essere in grado di mantenere internamente una lista di assunzioni di medicinali, con data, ora e slot. Si necessita quindi di consentire la modifica di tale lista all'utilizzatore. Questa operazione potrà essere eseguita mediante un apposito portale che consentirà all'utente o a chi per esso (ad esempio il figlio o chi se ne prende cura) non solo di modificare le assunzioni da effettuare, ma anche di avere un riepilogo delle assunzioni avvenute o non avvenute, con che ritardo, eccetera.

Gli utenti dovranno registrarsi sul portale e associare al proprio account uno o più porta-medicine che andranno a gestire. Tale associazione verrà effettuata mediante un QR code sul dispositivo, contenente un codice univoco in base64 che, una volta scannerizzato, effettuerà il pairing.

L'interfaccia si comporrà quindi di un calendario, in cui sarà possibile aggiungere o rimuovere le varie assunzioni, specificando data e ora, il farmaco da assumere e lo slot in cui sarà posizionato. Per semplificare la gestione dovrà anche permettere di specificare una cadenza di assunzione, ad esempio:

- Ogni 3 giorni
- Ogni settimana di lunedì, martedì e mercoledì

Di conseguenza si dovrà anche essere in grado di specificare fino a quando tali assunzioni dovranno avvenire, specificando una data o un numero di occorrenze.



Il portale sarà un sito web realizzato mediante l'utilizzo di framework **responsive** (es. Bootstrap, Bulma), in modo da essere facilmente accessibile da qualsiasi piattaforma, sia desktop che mobile.

3.2 INFRASTRUTTURA TECNOLOGICA ED INFORMATICA

3.2.1 Modalità di comunicazione

Sarà quindi necessario fare comunicare tra loro i porta-medicine e le applicazioni utente per scambiare i dati che ciascuno produce, ovvero:

- I dati prodotti dai vari porta-medicine, relativi a quando le assunzioni sono avvenute
- I dati relativi all'applicazione utente, che permette di configurare quali medicine assumere e con che cadenza

Si potrebbero adottare due soluzioni diverse:

- Comunicare mediante tecnologie wireless come WiFi (rete locale) o Bluetooth, salvando tutti i dati sul porta-medicine stesso
- Utilizzare un'architettura Client-Server, salvando tutti i dati sul server centralizzato

La prima soluzione ha un vantaggio in termini di costi, in quanto oltre ai costi di produzione del prodotto non se ne hanno altri relativi al mantenimento dei server. Tuttavia, si potrebbe accedere allo storico del porta-medicine solo quando si è in prossimità di esso. Preferisco quindi adottare la seconda soluzione, permettendo ai tutori di poter consultare da ovunque lo storico dei porta-medicine ed avere informazioni in tempo reale.

Ogni porta-medicine dovrà quindi essere dotato di una connessione a Internet. Si potrebbe pensare di connetterlo tramite una connessione WiFi ma, essendo il target del prodotto le persone anziane, conviene utilizzare servizi di rete mobile tramite una scheda SIM.

Si dovrà quindi predisporre un server che si occuperà dello scambio di messaggi tra i porta-medicine e le applicazioni utente. Si comporrà di un'architettura **three-tier**: verrà utilizzato un DBMS per la persistenza delle informazioni a cui solo il server avrà accesso, mentre i client (ovvero porta-medicine e applicazioni utente) accederanno ad esse passando per l'interfaccia esposta dal server.

Presento quindi la seguente architettura:

Viene quindi esposto un web-service con un API Rest per potere fornire le informazioni alle applicazioni utente. Utilizzando un web-service si avrà in futuro anche l'opzione di sviluppare delle applicazioni mobile e desktop native per una migliore user experience.

La comunicazione con i porta-medicine invece deve essere bidirezionale. Con questo si intende che:

- I porta-medicine devono spedire al server tutte le assunzioni
- Il server deve comunicare a i porta-medicine eventuali cambiamenti nelle configurazioni delle assunzioni

Si potrebbero adottare quindi due vie:

- Utilizzare sempre un API Rest, facendo operazioni di polling a intervallo costante dai porta-medicine. Questo significa che ogni porta-medicine dovrebbe effettuare a intervallo costante, ad esempio 1 volta ogni 5 minuti, una serie di richieste all'API per ottenere i dati necessari.
- Utilizzare un socket, aperto dal porta-medicine al suo avvio e mantenuto aperto fino al suo spegnimento. Il server potrebbe quindi comunicare informazioni al porta-medicine nell'esatto istante in cui si ha un cambiamento.

Ho scelto di adottare la seconda via. La prima infatti risulterebbe in un costante scambio di grandi quantità di informazioni. L'API Rest, essendo stateless, non manterrebbe alcun dato relativo allo stato di ogni porta-medicine e quindi sarebbe costretto a inviare ogni volta l'intera configurazione.

Nel secondo metodo invece la configurazione completa sarebbe mandata solo all'apertura della connessione. Dopodiché il server manderebbe solo gli effettivi cambiamenti. Questo perché andremo a utilizzare il **TCP**, un protocollo **connesso e affidabile**, e quindi il server ha la sicurezza che il client abbia ricevuto tutti i messaggi precedenti e quindi le due configurazioni sono sincronizzate.

3.2.2 Sistemi server

Il sistema non necessita quindi di una particolare struttura di rete, ma solo di un server su cui eseguire il software necessario, composto da:

- Un DBMS, ovvero MariaDB
- Un web server, ovvero il servlet container Tomcat
- Il software dell'applicazione, che implementerà sia la server-socket che il web service
- Una runtime Java (JRE), per eseguire Tomcat e il nostro software

L'opzione migliore è quindi quella di utilizzare un servizio di **cloud hosting**, ovvero che permetta di acquistare delle risorse computazionali come potenza di calcolo, spazio di archiviazione e velocità della rete sulla quale si può eseguire il software desiderato.

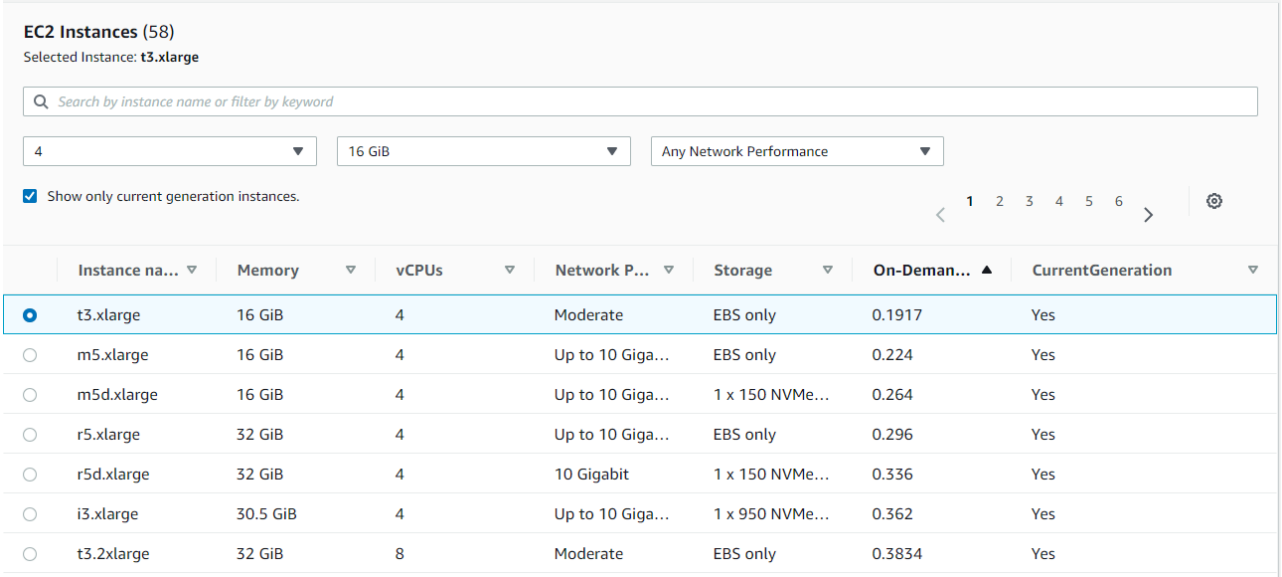
Questi servizi permettono di ridurre i costi di mantenimento dell'infrastruttura e di gestire meglio le risorse necessarie per applicazioni con un traffico instabile e con dei periodi di picco. Infatti, non ci si dovrà preoccupare dell'acquisto e della manutenzione di hardware e dell'infrastruttura di rete fattori in completa discrezione del provider del servizio.

Esistono due modelli principali che vengono offerti:

- **PaaS** (Platform as a Service): viene messa a disposizione una piattaforma software che include solitamente il sistema operativo, l'environment di esecuzione di un linguaggio di programmazione, un DBMS e un web server. Non si ha quindi la possibilità di cambiare questi componenti, ma solo di eseguire un'applicazione compatibile con quelli offerti.
- **IaaS** (Infrastructure as a Service): viene messa a disposizione una macchina virtuale sulla quale si può eseguire qualsiasi software. Si ha quindi il controllo completo sul sistema operativo, sul file system e anche su alcuni componenti della rete (es. Il firewall dell'host).

In questo caso adotterei il secondo perché, avendo anche un socket, si necessita di aprire un'ulteriore porta oltre a quella dell'HTTP e, con la maggior parte dei provider PaaS, questo non è possibile.

Si dovranno quindi andare a definire le risorse da acquistare. Queste di solito includono la potenza di calcolo, definita in **vCPU**, la dimensione della RAM e dello spazio di archiviazione e la velocità della rete. Una vCPU corrisponde a un CPU virtuale assegnata dall'**hypervisor** alla macchina virtuale. Infatti, non si acquista una macchina server vera e propria, ma solo un'istanza virtuale che viene eseguita insieme a numerose altre sulle macchine del provider. Bisogna anche specificare il sistema operativo desiderato.



	Instance na...	Memory	vCPUs	Network P...	Storage	On-Deman...	CurrentGeneration
◉	t3.xlarge	16 GiB	4	Moderate	EBS only	0.1917	Yes
○	m5.xlarge	16 GiB	4	Up to 10 Giga...	EBS only	0.224	Yes
○	m5d.xlarge	16 GiB	4	Up to 10 Giga...	1 x 150 NVMe...	0.264	Yes
○	r5.xlarge	32 GiB	4	Up to 10 Giga...	EBS only	0.296	Yes
○	r5d.xlarge	32 GiB	4	10 Gigabit	1 x 150 NVMe...	0.336	Yes
○	i3.xlarge	30.5 GiB	4	Up to 10 Giga...	1 x 950 NVMe...	0.362	Yes
○	t3.2xlarge	32 GiB	8	Moderate	EBS only	0.3834	Yes

Una volta effettuato l'acquisto, per il setup del software e per il deploy dell'applicazione bisognerà innanzitutto potersi connettere al server. Solitamente bisogna abilitare il servizio **SSH** dal pannello di controllo messo a disposizione dal provider. SSH (Secure Shell) è un protocollo di comunicazione che consente di aprire una sessione cifrata da remoto tramite riga di comando. Utilizza di solito la porta 22 e permette di installare da linea di comando tutti i software necessari. Invece, i binari del programma verranno trasferiti attraverso **SFTP** (SSH File Transfer Protocol). Questo, essendo basato su SSH, non solo utilizza un canale sicuro e cifrato, ma non richiede l'apertura di ulteriori porte in quanto, a differenza dei protocolli FTP e FTPS, utilizza unicamente la porta 22.

3.3 Modello dei dati

Si deve poi progettare l'architettura della base di dati. Emergono le seguenti entità:

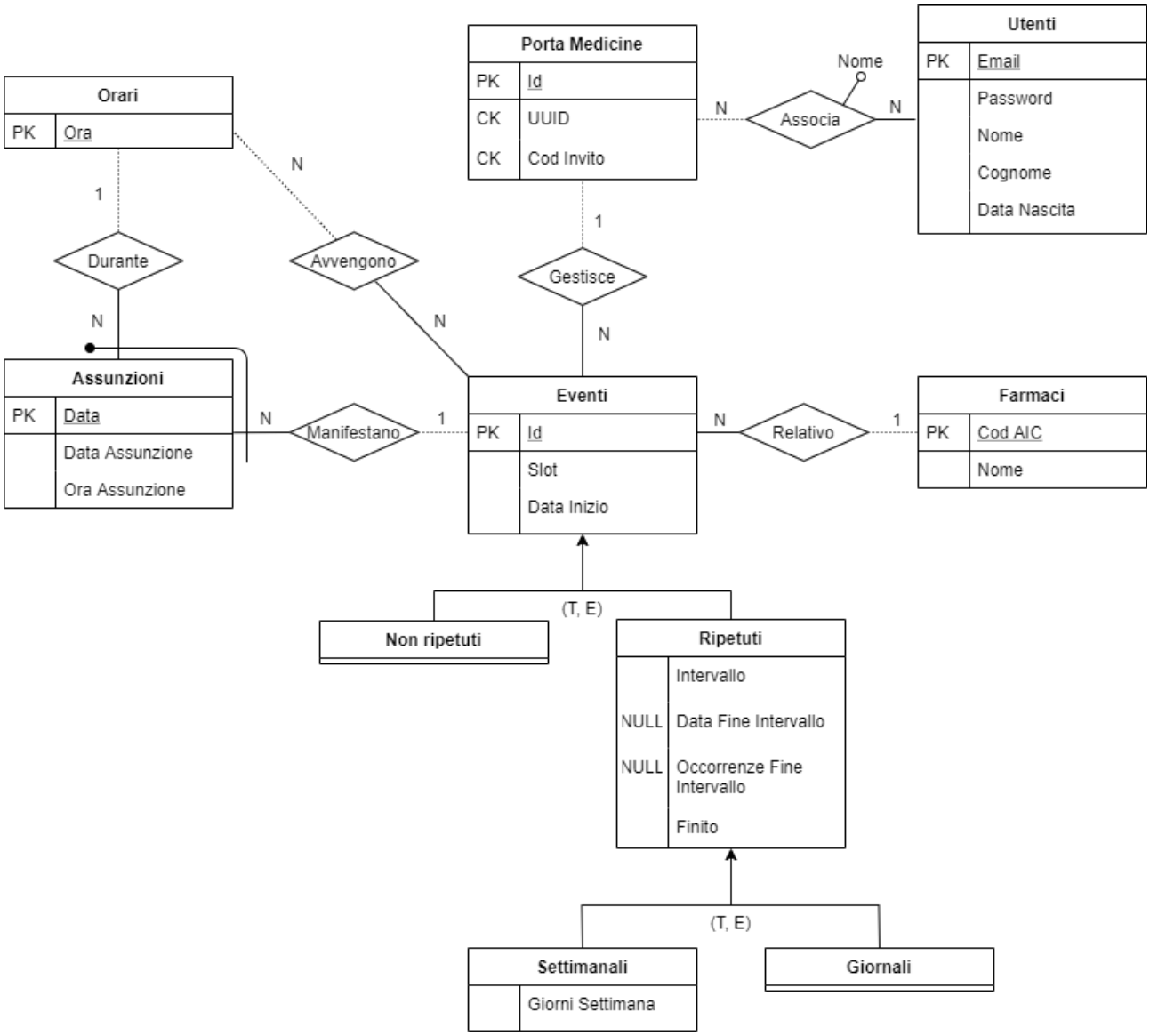
- Porta-medicine: tutti i porta-medicine esistenti
- Utenti: rappresentano gli account degli utenti registrati sul portale ai quali verranno poi associati i porta-medicine
- Farmaci: elenco di farmaci che è possibile assumere
- Eventi: descrive come deve avvenire l'assunzione di un determinato farmaco, specificando la cadenza, la data di inizio e di fine, lo slot in cui sarà posizionato. Un evento può ripetersi nel tempo o avvenire una volta sola, avere una fine ben stabilita per data o per occorrenze o ripetersi fino a quando non viene eliminato da un utente
- Assunzioni: istanza di un evento con una data e un'ora ben precisa

Le associazioni più rilevanti sono:

- Associa: quando un utente associa un porta-medicine al proprio account
- Gestisce: come il porta-medicine dovrà assicurarsi che le assunzioni avvengano
- Manifestano: quando gli eventi generali diventano effettive assunzioni con data e ora

3.3.1 Schema concettuale

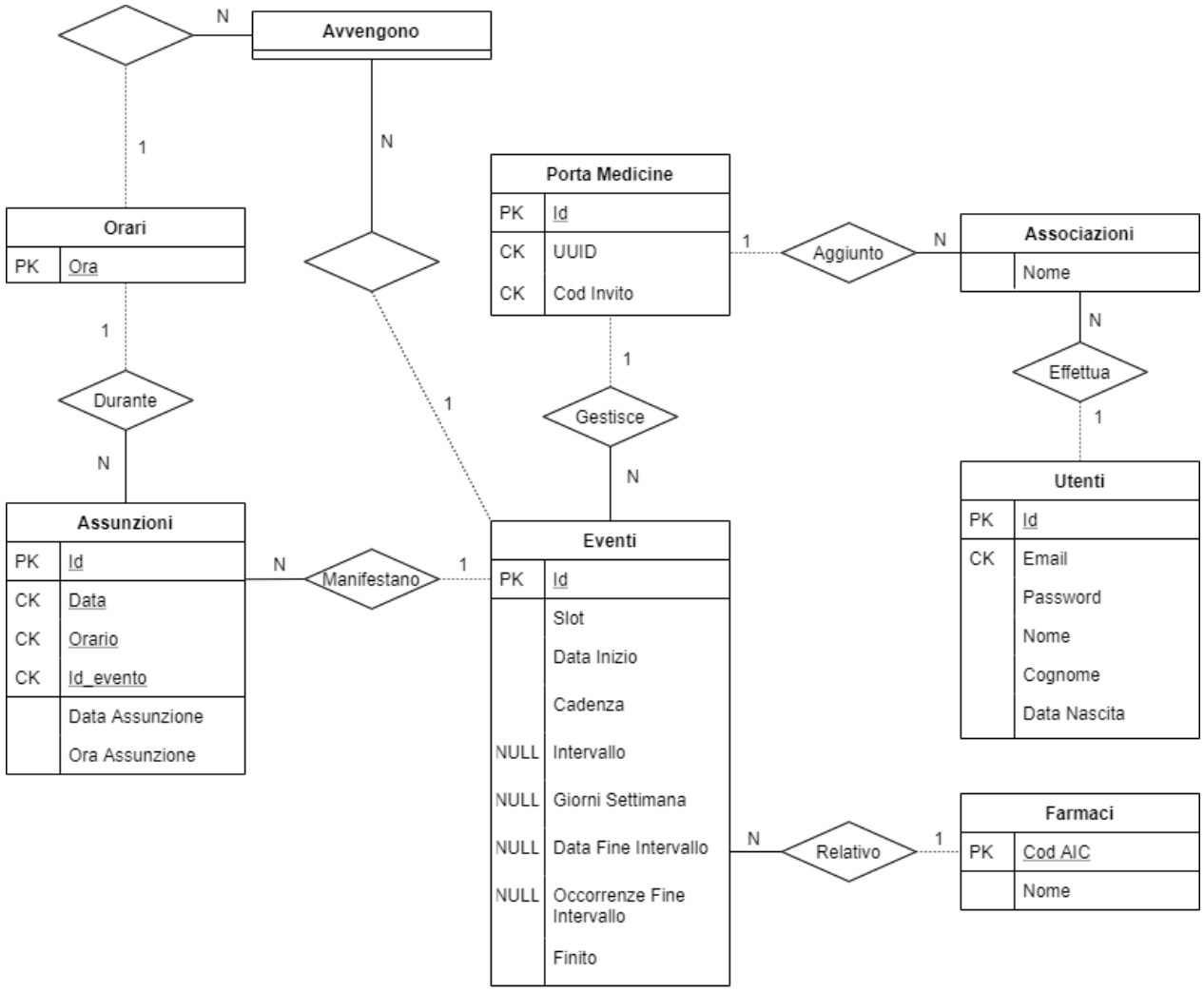
Lo schema ER che descrive il modello concettuale è quindi il seguente:



Prima di effettuare l'azione di mapping vado ad effettuare un'operazione di **ristrutturazione** dello schema:

- Sostituisco le associazioni N-N con altre due entità Associazioni e Avvengono. In entrambe le relazioni le chiavi primarie formate dalle chiavi esterne possono essere ritenute univoche.
- La generalizzazione (**ISA**) su Ripetuti, viene risolta inglobando le entità derivate in quanto totale ed esclusiva e in considerazione del loro poca significatività in termini di associazioni e attributi. Viene quindi aggiunto un attributo "cadenza" per poterle distinguere.
- La generalizzazione (**ISA**) su Eventi è anch'essa totale ed esclusiva e viene risolta inglobando le entità derivate. L'attributo "cadenza", già presente, viene riutilizzato per indicare anche la distinzione tra ripetuti e non ripetuti.
- Le chiavi primarie delle entità Utenti e Assunzioni risultano computazionalmente poco efficienti. Questo perché la prima è un attributo di lunghezza variabile e possibilmente molto grande, mentre la seconda è una chiave composta da più attributi. Vengono quindi introdotte due **chiavi artificiali** ID e le chiavi sostituite diventano **chiavi candidate**.

Lo schema ristrutturato quindi diventa:



3.3.2 Schema logico

Posso ora effettuare il **mapping** per ottenere lo schema logico:

```
Porta_medicine(ID, uuid, cod_invito)
Utenti(ID, email, password, nome, cognome, data_nascita)
Associazioni(ID_PORTA_MEDICINE*, ID_UTENTE*, nome)
Farmaci(COD_AIC, nome)
Eventi(
   ID, slot, data_inizio,
   cadenza, intervallo, giorni_settimana,
   data_fine_intervallo, occorrenze_fine_intervallo, finito,
   id_porta_medicine*, aic_farmaco*
)
Orari(ORA)
Avvengono(ID_EVENTO*, ORARIO*)
Assunzioni(ID, data, data_assunzione, ora_assunzione, orario*, id_evento*)
```

3.3.3 Definizioni in linguaggio SQL

Seguono le definizioni in linguaggio SQL (**DDL**):

```sql
CREATE TABLE porta_medicine (
  id int(11) PRIMARY KEY AUTO_INCREMENT,
  uuid char(32) UNIQUE,
  cod_invito char(32) UNIQUE
);

CREATE TABLE utenti (
  id int(11) AUTO_INCREMENT PRIMARY KEY,
  email varchar(64) NOT NULL UNIQUE,
  password char(60) NOT NULL, -- bcrypt hash
size
  nome varchar(32) NOT NULL,
  cognome varchar(32) NOT NULL,
  data_nascita date NOT NULL,
);

CREATE TABLE associazioni (
  id_porta_medicine int(11),
  id_utente int(11),
  nome varchar(32) NOT NULL,
  PRIMARY KEY (id_porta_medicine, id_utente),
  FOREIGN KEY (id_porta_medicine)
    REFERENCES porta_medicine (id)
      ON DELETE CASCADE
      ON UPDATE CASCADE,
  FOREIGN KEY (id_utente)
    REFERENCES utenti (id)
      ON DELETE CASCADE
      ON UPDATE CASCADE
);

CREATE TABLE farmaci (
  cod_aic int(9) unsigned zerofill PRIMARY
KEY,
  nome varchar(64) NOT NULL
);

CREATE TABLE orari (
  ora time PRIMARY KEY
);


CREATE TABLE eventi (
  id int(11) AUTO_INCREMENT PRIMARY KEY,
  slot int(11) NOT NULL,
  id_porta_medicine int(11) NOT NULL,
  aic_farmaco int(9) unsigned zerofill NOT
NULL,
  data_inizio date NOT NULL,
  cadenza ENUM('non ripetuta', 'settimanale',
'giornaliera') NOT NULL DEFAULT 'non
ripetuta',
  intervallo int DEFAULT NULL,
  giorni_settimana tinyint(7) UNSIGNED DEFAULT
NULL,
  data_fine_intervallo date DEFAULT NULL,
  occorrenze_fine_intervallo int DEFAULT NULL,
  finito tinyint(1) NOT NULL DEFAULT 0,
  FOREIGN KEY (aic_farmaco)
    REFERENCES farmaci (cod_aic)
      ON DELETE RESTRICT
      ON UPDATE CASCADE,
  FOREIGN KEY (id_porta_medicine)
    REFERENCES porta_medicine (id)
      ON DELETE CASCADE
      ON UPDATE CASCADE
);

CREATE TABLE avvengono (
  id_evento int(11),
  orario int(11),
  PRIMARY KEY (id_evento, orario),
  FOREIGN KEY (id_evento)
    REFERENCES eventi (id)
      ON DELETE CASCADE
      ON UPDATE CASCADE,
  FOREIGN KEY (orario)
    REFERENCES orari (ora)
      ON DELETE RESTRICT
      ON UPDATE RESTRICT
);

CREATE TABLE assunzioni (
  id int(11) NOT NULL PRIMARY KEY,
  id_evento INT(11) NOT NULL,
  data date NOT NULL,
  orario time NOT NULL,
  data_assunzione date,
  orario_assunzione time,
  FOREIGN KEY (orario)
    REFERENCES orari (ora)
      ON DELETE RESTRICT
      ON UPDATE RESTRICT,
  FOREIGN KEY (id_evento)
    REFERENCES eventi (id)
      ON DELETE CASCADE
      ON UPDATE CASCADE,
  CONSTRAINT unique_assunzioni
    UNIQUE (id_evento, data, orario)
);
```

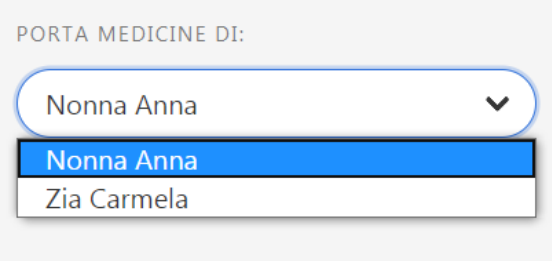
3.4 Segmento significativo dell'applicazione

Il portale avrà una struttura molto semplice: sarà composto da una pagina di login, una di registrazione e una pagina principale accessibile una volta loggati. Qui sarà presente il calendario con tutti gli eventi dei propri porta-medicine. Tutte le operazioni di aggiunta, modifica e cancellazione saranno accessibili da questa pagina mediante dei modal e utilizzeranno **AJAX** per effettuare le richieste.

La pagina quindi conterrà inizialmente solo la struttura di base (con relativo HTML e CSS), mentre i contenuti e i dati verranno caricati e impaginati da JavaScript dinamicamente una volta che la pagina è stata caricata.

Un esempio semplice di questo può essere osservato nel seguente frammento di codice:

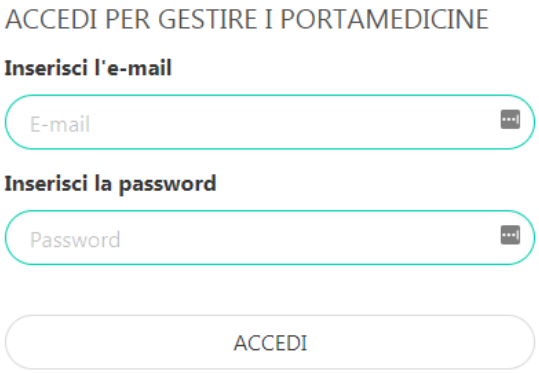
```
<div class="field">
    <div class="control">
        <div class="select is-rounded is-fullwidth">
            <!-- Template da clonare -->
            <select class="is-hidden">
                <option class="porta-medicine-template"></option>
            </select>
            <!-- Container che conterrà gli elementi caricati -->
            <select id="porta-medicine-container"></select>
        </div>
    </div>
</div>
```




```
<script>
const container = document.querySelector('#porta-medicine-container');
const templateClass = 'porta-medicine-template';
const template = document.querySelector('.' + templateClass);
// Quando la pagina ha finito di caricare il DOM,
// carico i dati dall'endpoint dell'API Rest
fetch('/api/porta_medicine').then(async response => {
    // Controllo che la risposta sia positiva
    if(!response.ok)
        throw response.status + ": " + (await response.text());
    // Faccio il parsing con la libreria JXON di Mozilla
    return JXON.stringToJs(await response.text());
}).then(devices => {
    let obj = devices.porta_medicine.porta_medicina;
    if(!obj)
        return [];
    if(!obj.forEach)
        return [ obj ];
    return obj;
}).then(devices => devices.forEach(device => {
    // Copio il template scritto in html
    const deviceNode = template.cloneNode(true);
    deviceNode.classList.remove(templateClass);
    // Modifico i campi rilevanti con i dati caricati
    deviceNode.value = device.id;
    deviceNode.textContent = device.nome;
    // Aggiungo
    container.appendChild(deviceNode);
}));
</script>
```

Mentre un esempio di form (ad esempio per il login):

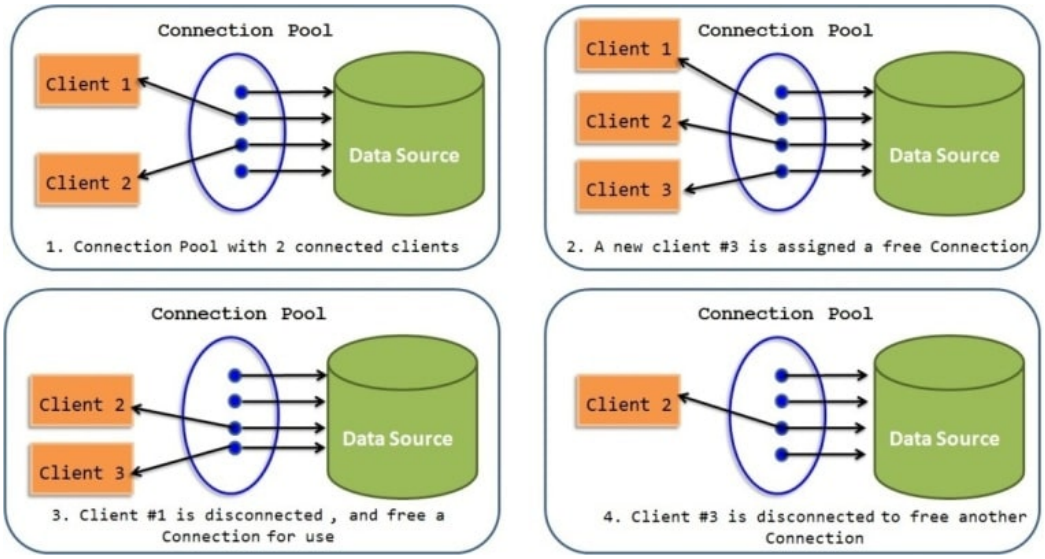
```
<form method="GET" action="/api/utenti">
    <label>Inserisci l'e-mail</label>
    <input type="email" placeholder="E-mail" id="email" required/>

    <label>Inserisci la password</label>
    <input type="password" placeholder="Password" id="password" required/>

    <input type="submit" value="ACCEDI"/>
</form>
```




```
<script>
document.querySelector("form").addEventListener('submit', event => {
    // Stop the form default behaviour
    event.preventDefault();
    // Input checks
    // ...
    const form = event.target;
    const ema = document.getElementById("email").value;
    const psw = document.getElementById("password").value;

    return fetch(form.action + "?email=" + ema + "&password=" + psw, {
        method: form.method
    }).then(async response => {
        const code = response.status;
        if (code === 200) {
            // logged
            location.href = '/home';
        } else if(code === 401) {
            // email not found or password wrong
            // Show error to the user
            // ...
        } else if(!response.ok) {
            throw response.status + ": " + (await response.text());
        }
    }).catch(ex => {
        // Show error to the user
        // ...
        console.error(ex);
    });
};
</script>
```

Il server invece dovrà come prima cosa poter comunicare con il database. Ci sono diverse strategie, tra cui:

- Aprire la connessione per ogni richiesta. Questo significa che l'utente dovrà aspettare su ogni richiesta il tempo di connessione al database. Inoltre, si potrebbe potenzialmente aprire un numero di connessioni spropositato, uguale al numero di richieste contemporanee effettuate.
- Utilizzare il **connection pooling**. Una connessione, una volta creata e usata, viene messa in una pool e riutilizzata per altre richieste.



Di seguito lo snippet di codice per la creazione della pool:

```
final PoolProperties properties = new PoolProperties();
// Driver di MySQL
properties.setDriverClassName("com.mysql.cj.jdbc.Driver");
// Stringa di connessione, contentente ip (localhost), porta (3306) e nome dello schema
(medicine_dealer)
properties.setUrl("jdbc:mysql://localhost:3306/medicine_dealer");
// Nome dell'utente. Deve avere i privilegi minimi necessari
// per visualizzare, inserire, modificare e cancellare
properties.setUsername("medicine_dealer_server");
// Password dell'utente
// Dovrebbe essere caricata da un file o variabili d'ambiente e non nel codice
properties.setPassword("bufsdubbulsDIGUiYBJK987T");
// Pool parameters
properties.setInitialSize(15);
properties.setMinIdle(15);
properties.setMaxActive(20);

final DataSource dataSource = new DataSource();
dataSource.setPoolProperties(properties);
// Context usato per fare le query
final DSLContext ctx = DSL.using(dataSource, SQLDialect.MARIADB);
```

3.5 AUTENTICAZIONE

L'API Rest inoltre, essendo stateless, ovvero non conservando informazioni sullo stato dei client, deve gestire le **sessioni** dei client per poter autorizzare gli utenti ad accedere a diverse parti dell'API. Una volta verificata la validità del login è quindi essenziale che venga inizializzata una sessione.

```
private final Key key = ...;

@GET
@Consumes(MediaType.APPLICATION_XML)
public Response login(@Context UriInfo uriInfo,
                      @NotNull @QueryParam(value = "email") String email,
                      @NotNull @QueryParam(value = "password") String password) {
    final Optional<Record3<Integer, String, String>> record = ctx
            .select(Tables.UTENTI.ID, Tables.UTENTI.EMAIL, Tables.UTENTI.PASSWORD)
            .from(Tables.UTENTI)
            .where(Tables.UTENTI.EMAIL.equal(email))
            // Il ciclo di fetch diventa uno stream
            .fetch()
            .stream()
            .filter(r -> passwordEncoder.matches(password, r.get(Tables.UTENTI.PASSWORD)))
            .findFirst();
    // Se non c'è alcun record con l'id e la password data, restituisci status code 403
    if(!record.isPresent())
        return Response
                .status(Response.Status.UNAUTHORIZED)
                .build();

    final int id = record.get().get(Tables.UTENTI.ID);
    final String actualEmail = record.get().get(Tables.UTENTI.EMAIL);
    // Genera un token di sessione e restituiscilo come cookie
    final String token = Jwts.builder()
                .claim("id", id)
                .setSubject(actualEmail)
                .setIssuer(uriInfo.getAbsolutePath().toString())
                .setIssuedAt(new java.util.Date())
                // Durata della sessione
                .setExpiration(java.util.Date.from(ZonedDateTime.now().plusHours(1L).toInstant()))
                .signWith(key, SignatureAlgorithm.HS512)
                .compact();
    return Response.ok()
            .cookie(makeAuthCookie(token))
            .build();
}

private NewCookie makeAuthCookie(String token) {
    return new NewCookie(
            "session_cookie",
            token,
            null, // the URI path for which the cookie is valid
            null, // the host domain for which the cookie is valid
            NewCookie.DEFAULT_VERSION, // the version of the specification to which the cookie
complies
            null, // the comment
            // No max-age and expiry set, cookie expires when the browser gets closed
            NewCookie.DEFAULT_MAX_AGE, // the maximum age of the cookie in seconds
            null, // the cookie expiry date
            true, // specifies whether the cookie will only be sent over a secure connection
            true // if {@code true} make the cookie HTTP only
    );
}
```

I token generati saranno dei **JWT** (JSON Web Tokens), ovvero delle stringhe di testo in base 64 contenente informazioni sull'utente (nel nostro caso id ed e-mail). Questi token vengono poi inseriti in un cookie di sessione che il server manderà all'utente. Il browser si occuperà quindi di salvarlo e di mandarlo in ogni successiva richiesta effettuata dall'utente.

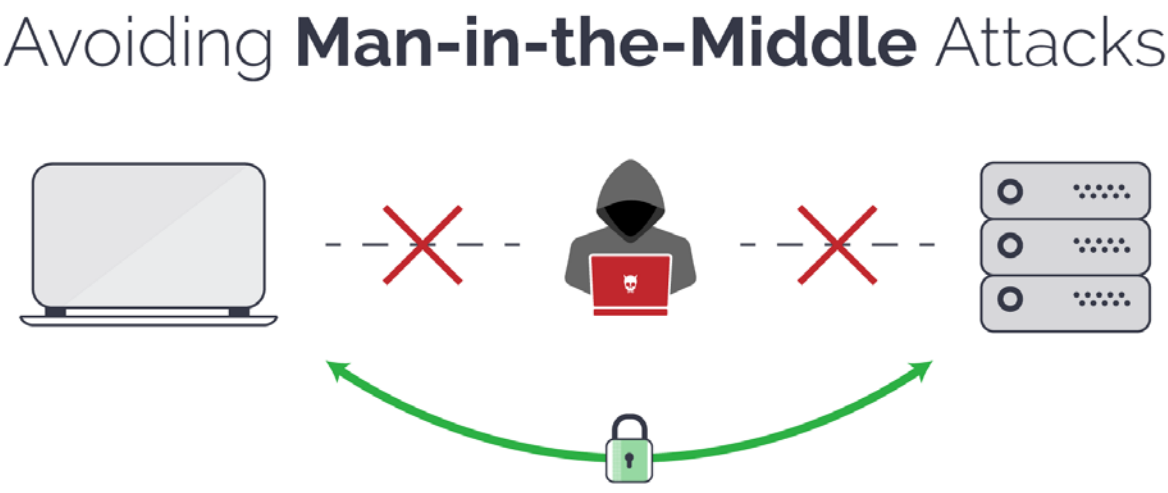
```
final ContainerRequestContext requestContext = ...; // Richiesta effettuata dall'utente
final Cookie cookie = requestContext.getCookies().get("session_cookie");
if(cookie == null)
    throw new NotAuthorizedException("A session cookie must be provided");

final Map<String, Object> sessionVariables;
try {
    // Recupera il token dal cookie
    final String token = cookie.getValue();
    sessionVariables = authenticationService.authenticate(token);
} catch (AuthenticationException e) {
    throw new NotAuthorizedException("A session cookie must be provided");
}
// Variabili di sessione
final int id = (Integer) sessionVariables.get().get("id");
final String email = (String) sessionVariables.get().get("email");
```

3.6 Sicurezza

Particolare attenzione dovrà essere posta sulla sicurezza dell'applicazione. Esistono infatti molti tipi di attacchi che possono essere compiuti su questo tipo di applicazioni, primi tra tutti attacchi di **MITM** (Man in the middle), di **eavesdropping** e di **tampering**. Utilizzando un protocollo non sicuro come http infatti si hanno due problemi:

- **L'autenticazione del server**: il client non sa se sta comunicando con il server effettivo o con qualcuno che finge di esserlo. Infatti, negli attacchi MITM l'attaccante si pone in mezzo alla comunicazione tra client e server, facendo credere a ciascuno di stare comunicando direttamente con l'altra parte.
- I dati in transito sono in **chiaro**, rendendo possibile a chiunque di leggerli e manometterli.

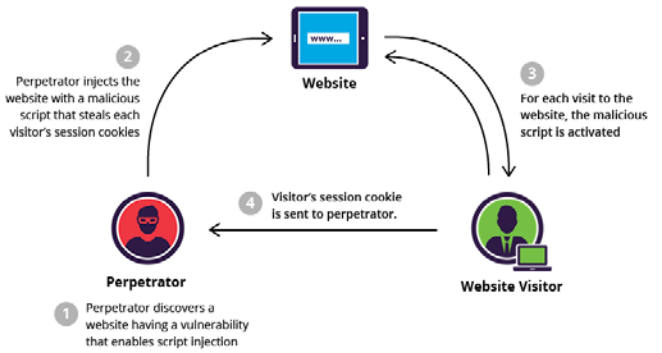
Questi due tipi di problemi possono essere risolti utilizzando un protocollo di comunicazione sicuro come **HTTPS**, ovvero HTTP over TLS (Transport Layer Security). Il server dovrà avere un **certificato digitale** firmato da una **certificate authority** fidata, in modo che questa possa garantire la legittimità del sito e autenticarlo. Inoltre, l'intero payload del pacchetto HTTP verrà criptato prima di essere spedito sul canale di comunicazione.



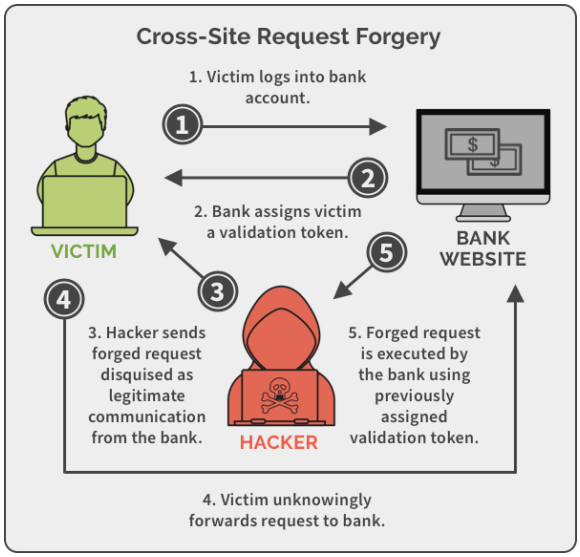
Bisogna porre anche particolare attenzione alla verifica dei token di sessione. Infatti, anche se in encoding base64, sono delle stringhe di testo in chiaro mandate al client. Un malintenzionato potrebbe modificare tale stringa presente nel cookie, facendo credere al server di essere un altro utente. Per questa ragione, sui token JWT viene posta una **firma digitale** dal server, utilizzando una chiave privata di cui è l'unico a esserne in possesso. In questo modo viene garantito che il token sia stato rilasciato dal server e viene assicurata anche l'**integrità** del contenuto.

Altri attacchi da cui bisogna proteggersi sono quelli di **cookie hijacking**, che si verificano quando un attaccante entra in possesso di un cookie valido di un utente. Questo attacco non può essere completamente impedito, ma solo mitigato. Si può solo cercare di prevenire il furto dei cookie e, in caso avvenga, di limitare i danni che può causare. Il furto solitamente avviene attraverso l'utilizzo di altri tipi di attacchi legati alla sicurezza dei siti web, come ad esempio:

- **XSS** (Cross-site Scripting): vulnerabilità nel quale gli attaccanti iniettano degli script nelle pagine web visualizzate dall'utente. Ottiene così accesso a tutti i dati dell'utente salvati dal browser, tra cui anche il cookie di sessione. Per mitigare questo tipo di attacco esiste un parametro del cookie **httpOnly** che, se settato, istruisce il browser di non rendere accessibile il cookie da codice JavaScript.

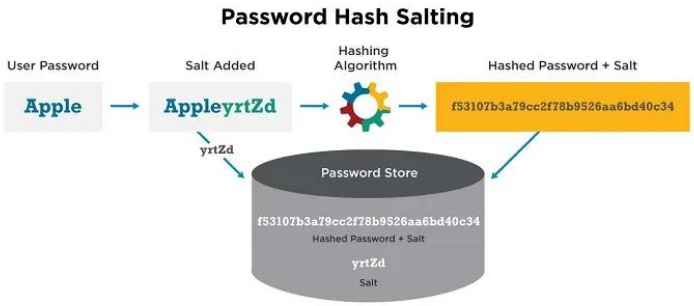


- **CSRF** (Cross-site Request Forgery): tipo di attacco dove l'attaccante fa eseguire ad un utente autenticato comandi non autorizzati. Solitamente, l'attaccante deve identificare un URL che esegue l'azione desiderata, per esempio cambiare la password dell'account dell'utente. Dovrà poi fare aprire tale link all'utente, per esempio posizionando il link all'interno di un sito web da lui controllato. Quando l'utente aprirà il link, il browser includerà automaticamente il cookie di sessione di tale sito e spedirà la richiesta al server, che non sarà in grado di distinguerla da una richiesta reale. Per prevenire questo tipo di attacco è stato aggiunto un ulteriore parametro ai cookie, chiamato **SameSite**, che permette di chiedere al browser di non inserire il cookie in richieste provenienti da altri siti.



Per limitare i possibili danni che possono avvenire nel caso il cookie venga rubato è invece importante impostare una durata massima alla sessione in modo che, anche se rubato, possa essere utilizzato solo per un periodo di tempo limitato.

Infine, bisogna anche salvare in maniera appropriata le password degli utenti. Questo perché, nel caso ci sia un **data breach**, bisogna prevenire che le password possano essere utilizzate da un attaccante per autenticarsi sul sito stesso o anche su siti differenti dove l'utente utilizza la medesima password. Bisogna quindi utilizzare una funzione di **hashing**, ovvero una funzione matematica one-way (irreversibile) che dando in input una stringa di lunghezza variabile restituisce un'hash (anche detta digest), una stringa a lunghezza fissa di caratteri incomprensibili che non possa essere ricondotta a quella originale. Questo dovrebbe obbligare l'attaccante a utilizzare attacchi di **brute force** per ottenere la password originale, processo che



richiede molto tempo (aumenta esponenzialmente tanto più la password è lunga). Tuttavia, l'uso di una funzione di hashing da sola non basta: esistono infatti delle tabelle di hash già computate di password molto usate chiamate **rainbow table**. Queste permettono di risalire dall'hash alla password in chiaro in poco tempo. Per combattere l'uso di queste tabelle si inserisce all'interno della password in chiaro, prima di esser passata alla funzione di hashing, una stringa di caratteri randomica molto lunga, diversa per ogni utente, chiamata **salt**. Questo fa sì che anche password molto comuni diventino complesse e quindi non trovabili nelle rainbow table. Il salt sarà poi salvato nel database insieme all'hash della password, in modo che quando viene ricevuta una richiesta di login, il server possa:

- Prendere il salt dell'utente dal database;
- Generare una hash con la password data dall'utente e il salt appena preso;
- Confrontare l'hash generata con quella salvata nel database, autenticando l'utente se le due corrispondono.