



StayHealthy

Giacomo Rocchetti & Alessandro Liscio

Cos'è StayHealthy?

StayHealthy è una PWA (Progressive Web Application) che ha lo scopo di connettere attraverso una piattaforma comune dottori e pazienti.

L'applicazione mette a disposizione dei pazienti servizi di accoppiamento, ottenimento e visualizzazione, attraverso grafici e tabelle, delle informazioni dei bracciali **Miband** (versioni 2 e 3), nonché compilazione di questionari e servizio messaggistica con il proprio dottore di riferimento.

Un dottore, dall'altro lato, potrà visualizzare tutte le informazioni dei propri pazienti, attraverso grafici e selezione a calendario. Potrà inoltre inviare messaggi ai propri pazienti, oltre a visualizzare e rispondere ai messaggi di questi ultimi.

Suddivisione del lavoro

Per lo sviluppo di questo progetto sono state individuate due fasi principali, corrispondenti con la suddivisione del lavoro:

- sviluppo del front-end, concernente la creazione dell'interfaccia grafica, la gestione delle interazioni tra utente e applicazione nonché la gestione della connessione tra applicazione e bracciale MiBand;
- sviluppo del back-end, concernente la creazione del server ospitante l'applicazione, lo sviluppo del database, la creazione chiamate API e la gestione delle strategie di autenticazione dell'utente e sicurezza delle informazioni.

Front-end

Creazione delle pagine

Il primo passo affrontato per lo sviluppo del front-end è stato quello della realizzazione delle varie pagine componenti l'applicazione. Avendo deciso di sviluppare una Single Page Application, abbiamo scelto di lavorare in [Angular](#), framework sviluppato da Google per l'implementazione di applicazioni web performanti: è infatti una particolarità di Angular il concetto di routing, il quale permette di navigare tra le pagine dell'applicazione senza subire il caricamento della nuova pagina, aumentando notevolmente le performance.



Angular utilizza il linguaggio [Typescript](#), un super-set di Javascript che permette la tipizzazione delle variabili e, quindi, un controllo maggiore sui parametri e valori di ritorno delle funzioni.

Per le pagine web, Angular utilizza il concetto di *componente*, che racchiude la view, intesa come output che si interfaccia all'utente, e il controller, inteso come elemento che gestisce le azioni relative alla view.

Questa fase di sviluppo, durata cinque giorni, ci ha permesso di sviluppare le interfacce grafiche dei seguenti componenti:

- **App**: componente principale. La direttiva `<router-outlet>` permette di inserire le view dei vari componenti nella medesima pagina, evitando il cambio di pagina e, di conseguenza, il caricamento;
- **Navbar**: componente presente nella parte superiore delle pagine, permette la navigazione dell'applicazione;
- **Login**: pagina iniziale con cui l'utente si interfaccia;
- **MiBand**: home page del paziente, utilizzata per la connessione al bracciale MiBand;
- **Survey**: pagina che permette al paziente di compilare il questionario;
- **History**: pagina utilizzata dal paziente per visionare i dati del bracciale e il questionario compilato in un determinato giorno.
- **Doctor-History**: stessa struttura del componente *History*, ma ha un dropdown aggiuntivo che permette al dottore di selezionare uno specifico paziente a cui controllare lo storico;
- **Messages**: permette di visualizzare i messaggi in arrivo, di rispondere a questi e di scriverne nuovi;
- **Page-Not-Found**: comparirà nel caso in cui l'url inserito non è corretto.

Per lo styling è stato scelto [Bootstrap](#), una libreria che contiene componenti e script per lo sviluppo di pagine full responsive, che si adattano ovvero alla dimensione della finestra.

Routing Angular

La fase successiva riguarda la gestione del routing tra le pagine. Come già detto precedentemente, Angular permette il cambio di view attraverso l'url immesso. E' stato quindi associato un url ad ogni componente e inserito il meccanismo di routing nel componente navbar. Fortunatamente Angular fornisce uno strumento molto semplice per la gestione di questa funzionalità, quindi è stato necessario meno di un giorno per implementarlo. Fatto questo, lo scheletro dell'applicazione è stato ufficialmente completato.

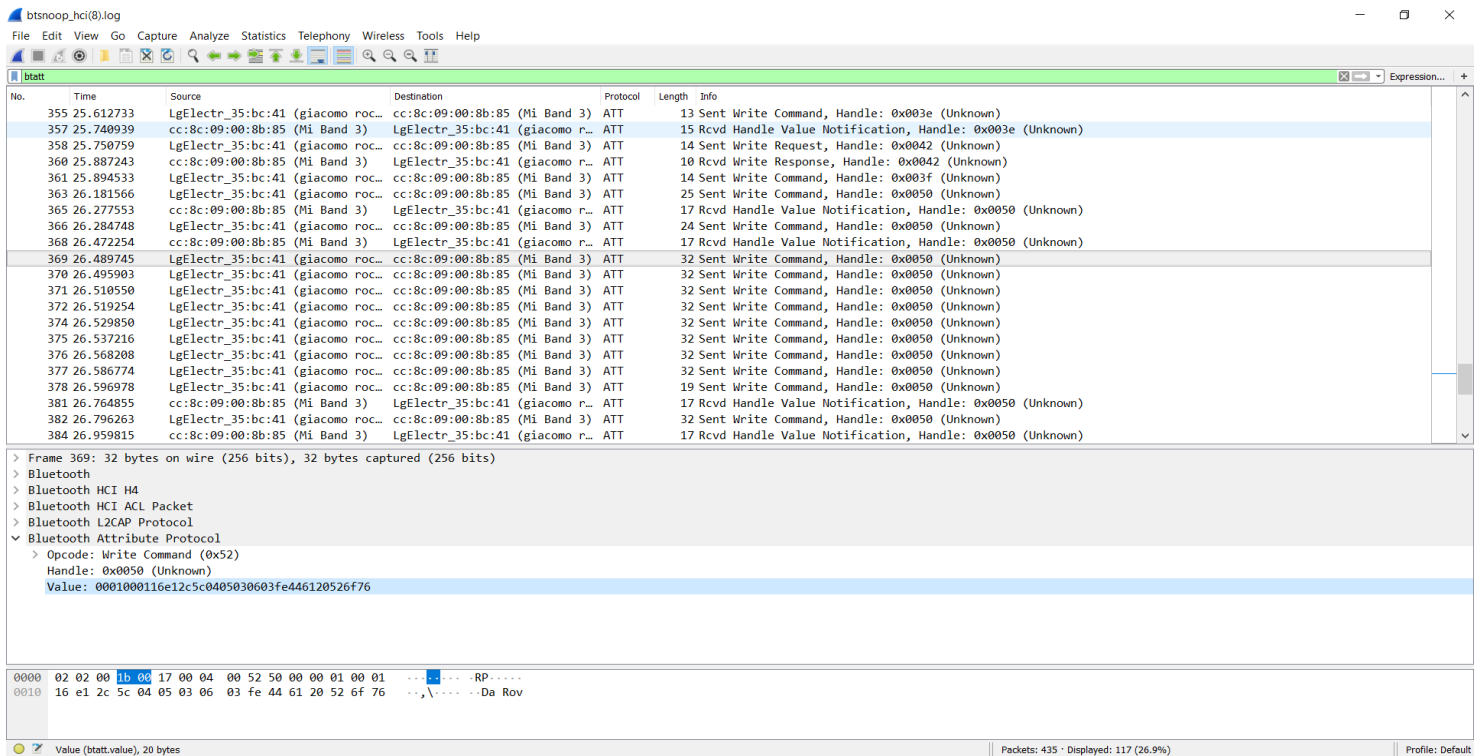
Chiamate API al server Node.js

La fase successiva è stata l'aggiunta delle richieste al server e la gestione delle sue risposte. E' buona norma non lasciare al componente stesso la mansione di gestire i dati e i modelli, ma di farli gestire dai *servizi*, ovvero classi che si occupano di funzionalità specifiche come, appunto, la gestione dei dati provenienti dal server. Questi servizi sono poi *iniettabili*, ovvero qualunque componente può utilizzarli. I servizi implementati sono i seguenti:

- **authorization.service**: permette di controllare se un utente è autorizzato a navigare in una determinata pagina (ad esempio, un paziente non può accedere alla pagina *doctor/messages*);
- **doctor.service**: contiene i metodi per reperire dal server un utente con ruolo dottore e i relativi pazienti;
- **patient.service**: come doctor.service, contiene le chiamate per ottenere un utente con ruolo paziente e il medico curante;
- **history.service**: permette di ottenere i dati presi dal bracciale di un'intera giornata passando in input la data che si vuole controllare;
- **login.service**: contiene le funzioni che permettono di reperire i dati di interesse di un utente al suo login;
- **messages.service**: permette di prendere messaggi dal server, inviarli ed eliminarli;
- **miband.service**: contiene le funzioni relative alla ricerca e connessione con un bracciale MiBand. Permette inoltre di inviare al server i dati delle attività;
- **survey.service**: contiene metodi per reperire il modello del questionario da compilare, per inviarne uno appena compilato o di reperire le risposte dei giorni precedenti.

MiBand

Il passo successivo è stato l'implementazione del bluetooth per connettere il bracciale MiBand all'applicazione web. E' stato quindi scelto di utilizzare la tecnologia **Web Bluetooth**, che permette la comunicazione di dispositivi BLE (Bluetooth Low Energy) ad una pagina web. Abbiamo trovato una libreria su GitHub (<https://github.com/vshymanskyi/miband-js>), sviluppata in Node.js, che utilizza questa tecnologia per comunicare con il bracciale, ma è subito sorto un problema: dovendo noi sviluppare una Progressive Web Application, dovevamo permettere all'applicazione di funzionare anche offline: non era quindi possibile implementare il meccanismo di Web Bluetooth nel server Node.js, poiché non avrebbe funzionato offline. Abbiamo quindi deciso di implementare il Web Bluetooth direttamente nell'applicazione Angular, scrivendo una libreria in Typescript quasi da zero. Purtroppo le documentazioni riguardo le API della MiBand sono scarse, quindi abbiamo dovuto prendere i file di log del bluetooth dallo smartphone a cui abbiamo associato il nostro bracciale, compiere alcune azioni e analizzarlo con **Wireshark**, un software open-source per l'analisi dei pacchetti, solitamente utilizzato per la risoluzione dei problemi di networking e per lo sviluppo dei protocolli di comunicazione.



Avendo quindi reperito le informazioni da Wireshark (in bytes) e da alcuni progetti presenti su GitHub, è stato effettuato un reverse-engineering del bracciale, permettendo di individuare le funzionalità necessarie per la nostra applicazione:

- **autenticazione:** si tratta di un meccanismo di autenticazione a chiave simmetrica che permette al bracciale di accoppiarsi con un altro dispositivo;
- **informazioni batteria:** permette di avere informazioni riguardante il livello della batteria, la data dell'ultimo caricamento e il livello raggiunto dall'ultimo caricamento;
- **abilitazione monitor battito cardiaco:** permette di attivare il sensore del battito cardiaco in modo da avere una misurazione ogni minuto;
- **lettura delle attività:** principale funzionalità del bracciale, permette di ottenere i dati partendo da una data specifica e, per ogni minuto, si hanno le seguenti informazioni:
 1. **codice attività:** codice numerico che il bracciale associa all'attività in base all'orientamento del dispositivo, dei passi e del movimento;
 2. **intensità:** intensità del movimento svolto in quel minuto;
 3. **passi:** passi effettuati in quel minuto;
 4. **battito cardiaco:** battito cardiaco misurato in quel minuto.

Questa fase è stata la più impegnativa del progetto, durata circa 4 settimane.

Implementazione MiBand in StayHealthy

Avendo quindi trovato come comunicare con il bracciale per reperire le informazioni descritte nel paragrafo precedente, abbiamo quindi implementato queste funzionalità nell'applicazione principale (cartella *miband*). Sono state inserite tutte le costanti necessarie per la comunicazione con il bracciale (sequenze specifiche di bytes) e implementata la classe *MiBand* che gestisce la connessione con esso. La classe è poi utilizzata dal componente *MiBand* per rendere le funzionalità accessibili all'utente.

Programmazione a eventi

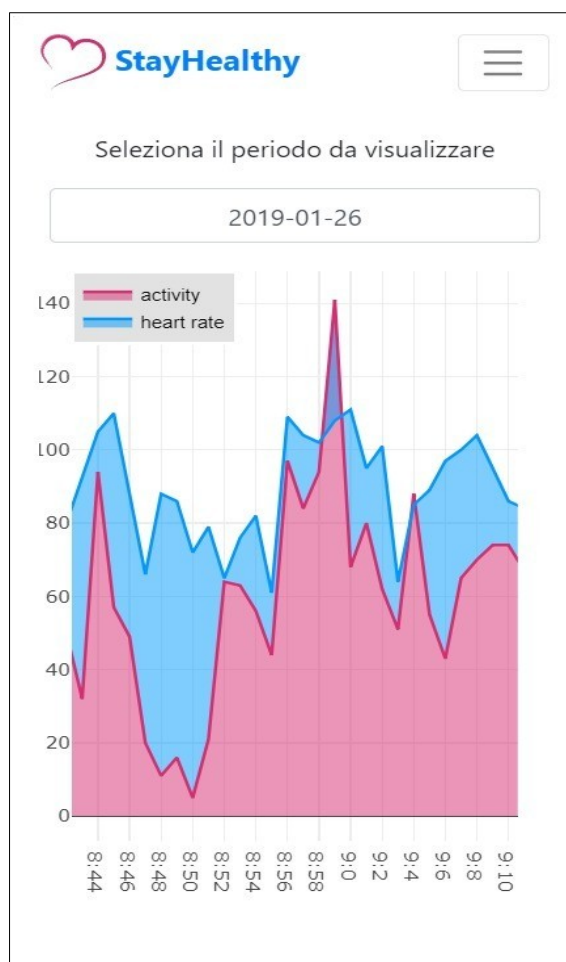
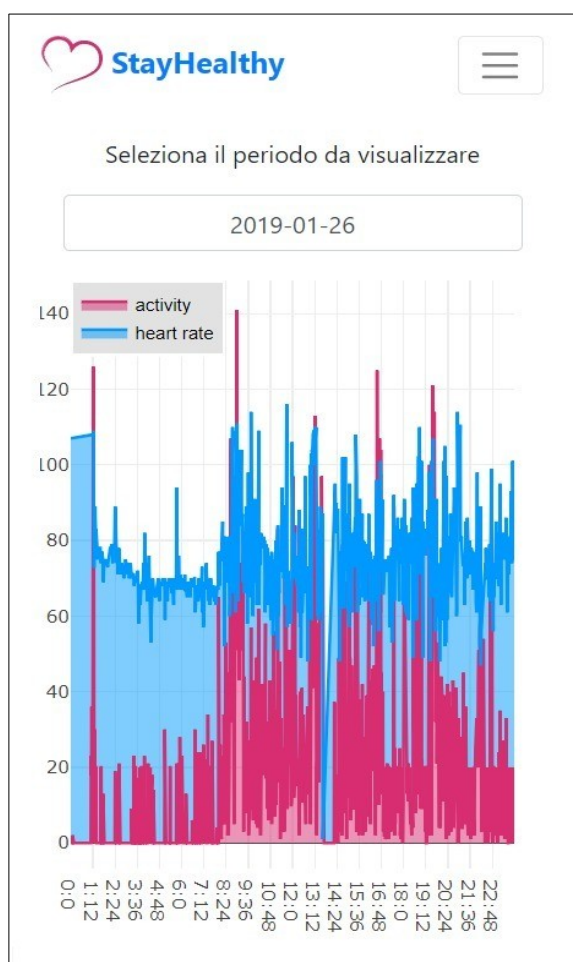
Arrivati a questo punto ci siamo resi conto di un problema non trascurabile: il progetto fin'ora creato non permetteva l'indipendenza dei componenti e, di conseguenza, non poteva essere estendibile. Abbiamo quindi deciso di implementare un approccio ad *eventi*: non è prerogativa di un componente comunicare il proprio stato, ma sta agli altri componenti *osservare* lo stato del primo e agire di conseguenza. Questo approccio è stato implementato attraverso la libreria *Reactive Extensions for JavaScript (RxJS)* e permette una semplice gestione delle funzioni asincrone e delle rispettive funzioni di callback. Prendiamo come esempio la connessione al bracciale: la schermata che permette di inviare i dati al server non dovrà comparire fino a che l'applicazione e il bracciale non avranno finito l'accoppiamento. Il componente *MiBand* si *sottoscrive* alla variabile di tipo Subject *is_authenticated\$* della classe *MiBand*. Nel momento in cui la variabile cambierà in *'AUTHORIZED'*, il componente farà comparire la schermata per la gestione delle azioni del bracciale. Questo meccanismo è stato applicato a tutti i componenti dell'applicazione in modo da avere totale indipendenza dei componenti e una conseguente estendibilità delle funzionalità.



Storico

Il passo successivo è stato quello di implementare il grafico per la schermata dello storico. Avendo gli storici del paziente e del dottore funzionalità in comune, abbiamo deciso di creare la classe astratta *GraphDrawer* che verrà poi estesa da questi due componenti. Inizialmente abbiamo deciso di utilizzare le librerie [Ng2-charts](#) e [Chart.js](#), le quali contengono direttive Angular per la creazione semplice di grafici. Si è presentato però un problema: non potevamo scorrere il grafico orizzontalmente per controllare i dati delle ore successive o precedenti. E' stato quindi necessario modificare direttamente la libreria *Ng2-charts* per permettere questo comportamento.

Questa soluzione ha però un'importante limitazione: è possibile solo scorrere il grafico in un intervallo di tempo prestabilito, ma non si possono effettuare operazioni di zoom. Avendo quindi il problema della libreria modificata e quello dello zoom, abbiamo deciso di cambiare libreria e di utilizzare [Plotly.js](#), che permette l'ingrandimento dei grafici in modo accessibile anche da dispositivi touchscreen.



Survey

Il passo successivo è stato implementare il meccanismo di costruzione di un questionario partendo dalle domande contenute nel database. Il componente *survey* chiederà quindi a *survey.service* di reperire tutte le domande del questionario dal database per poi costruirle in un formato json compatibile con la libreria *SurveyJS*, la quale permette la creazione di questionari e quiz da una struttura json. Inoltre fornisce anche il meccanismo di acquisizione delle risposte al completamento del questionario, sempre in formato json, che verrà poi inviato al server.

Nel caso in cui il paziente tenti di compilare il questionario per la seconda volta in un giorno, l'applicazione non permetterà la ricompilazione del questionario e mostrerà un messaggio di avviso. Successivamente è stato richiesto di cambiare il formato del questionario giornaliero, consentendo di valutare in una scala da 0 a 5 il proprio stato di depressione, dolore e stato di salute utilizzando la scala di *Wong Baker*. Utilizzando quindi delle immagini, non è stato possibile implementarlo con la libreria *SurveyJS*, ma è stato implementato a livello componente utilizzando una struttura dati per la gestione delle immagini da visualizzare e i valori ad esse associati. Le domande e la struttura del questionario vengono reperite sempre dal database.



Come ti senti oggi?

Depressione



Dolore

Stato di salute





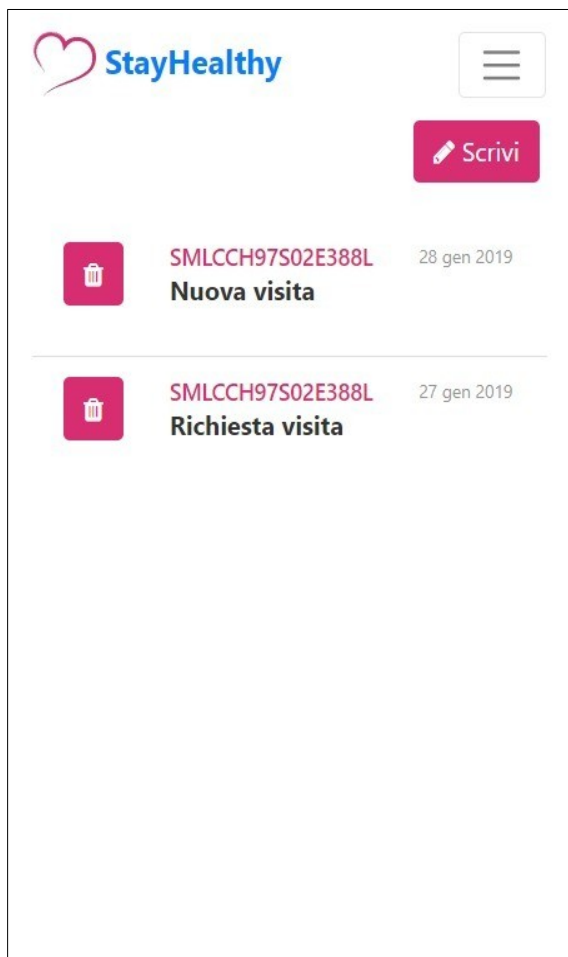
Il questionario è già stato compilato oggi, torna domani!

Messages

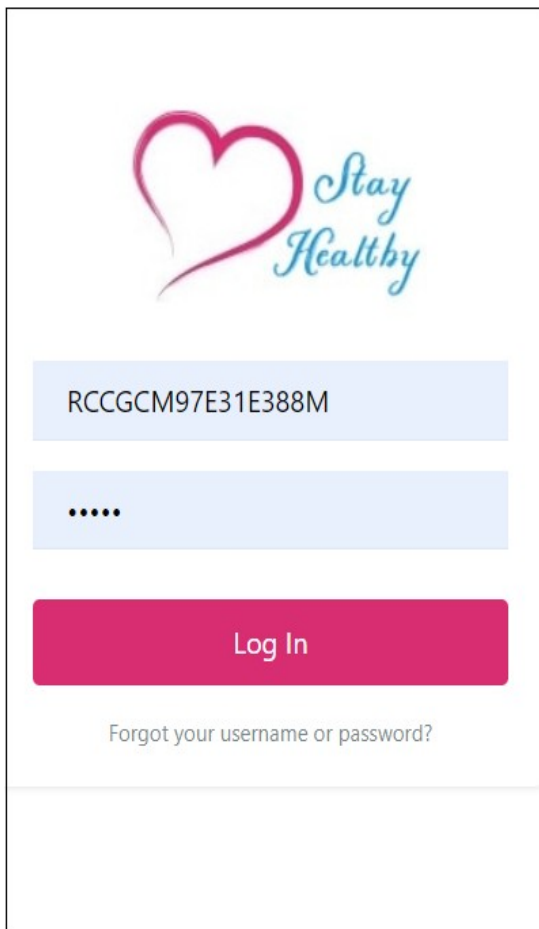
La fase successiva è stata l'implementazione della messaggistica. In questo caso, il componente *messages* è comune tra paziente e dottore, ma i due ruoli hanno funzionalità diverse: mentre un paziente può ricevere messaggi solamente dal medico curante e può inviarli soltanto a quest'ultimo, un dottore può ricevere messaggi da tutti i suoi pazienti e può selezionare a quale di questi inviare un messaggio.

Questi controlli vengono fatti direttamente dalle direttive Angular nella view *messages.html*. Il componente utilizza il servizio *messages.service* per l'invio, la ricezione e l'eliminazione dei messaggi. Ogni messaggio è formato dalle seguenti parti:

- **uuid**: utilizzato come chiave primaria nel database;
- **sender**: mittente del messaggio;
- **receiver**: destinatario del messaggio;
- **object**: oggetto del messaggio;
- **date**: data di invio;
- **content**: corpo del messaggio.



Login



The login form features a logo at the top with a pink heart outline and the text 'Stay Healthy' in a blue script font. Below the logo are two light blue input fields: the first contains the text 'RCCGCM97E31E388M' and the second contains five dots. A pink 'Log In' button is positioned below the password field. At the bottom of the form, there is a link that reads 'Forgot your username or password?'.

Successivamente abbiamo lavorato alla gestione del login. La prima decisione da prendere è stata con quale meccanismo effettuare l'autenticazione ed è stato scelto *passport.js* (che verrà spiegato nel dettaglio nella parte del back-end). Il *login.service* si occupa delle richieste da fare al server per richiedere l'autenticazione e i passi sono i seguenti:

1. l'utente inserisce username e password nel form della pagina;
2. *login.service* richiede il ruolo dell'utente al server inviando una richiesta contenente username e password appena immessi;
3. se l'autenticazione è andata a buon fine, il server risponde collocando un **cookie** per il controllo della sessione (in modo da non far scadere la sessione nel caso di chiusura della pagina senza logout) e ritornando il ruolo dell'utente:
 - se l'utente è un **patient**, l'applicazione chiede al server che le vengano inviati i dati relativi a sé stesso e al medico curante, che verranno salvati nel **localStorage** del browser;
 - se l'utente è un **doctor**, l'applicazione chiede al server che le vengano inviati i dati relativi a sé stesso e a tutti i pazienti che ha in cura, sempre salvandoli nel **localStorage**.
4. una volta ricevuti i dati, l'utente viene indirizzato nella relativa schermata home (*miband* nel caso del paziente, *doctor-history* nel caso del medico).

Utilizzando questo meccanismo, si fa in modo che i dati di base (ad esempio codice fiscale, elenco dei pazienti ecc.) non vengano reperiti ogni volta dal database, risparmiando il numero di chiamate e, nel caso di un database ospitato in cloud, soldi.

Questa parte del progetto ha richiesto discreto tempo per la comprensione del meccanismo di autenticazione e per la ricerca del metodo ottimale per il reperimento e il mantenimento dei dati utili all'utente.

Progressive Web Application

Concluso lo sviluppo del front-end e del server, è giunto il momento di trasformare questa semplice applicazione web in una vera e propria PWA.

Seguenti sono i requisiti che abbiamo dovuto seguire per trasformare l'applicazione in una PWA:

1. Il sito è ospitato in **https**: come verrà spiegato nella parte del back-end, tutto il traffico http è stato reindirizzato ad https.
2. Le pagine sono **responsive** per dispositivi come tablet e smartphones: questo requisito è stato soddisfatto grazie all'utilizzo di Bootstrap.
3. Tutti gli URLs dell'applicazione vengono caricati anche **offline (service worker)**: e' stato possibile aggiungere il service worker grazie alla libreria di Angular [@angular/pwa](#). Un service worker è uno script che viene eseguito in un browser e gestisce il meccanismo di caching dell'applicazione: abbiamo quindi fatto in modo che tutte le risposte da parte del server vengano salvate in cache e, se l'applicazione si trova in modalità offline, viene reperito dalla cache l'ultimo risultato della chiamata che si vuole eseguire.
4. Aggiungere dei metadati per l'installazione dell'applicazione in una home screen: la libreria [@angular/pwa](#) fornisce anche un file chiamato **manifest.json**: qui vengono salvate le informazioni relative all'installazione dell'applicazione, quali nome dell'app e icone di varie dimensioni in modo da poter essere visualizzate correttamente da tutti i dispositivi.
5. Il primo caricamento deve essere veloce anche con il 3G (<10 secondi): questo requisito è stato testato con il tool [Performance](#) fornito da Google Chrome per analizzare il rendering delle pagine e il relativo tempo, simulato in una connessione 3G. Inizialmente il caricamento superava di gran lunga i 10 secondi, ma grazie a meccanismi di compressione delle risposte dal server e minifying dei file Javascript, Html e CSS, siamo riusciti a raggiungere l'obiettivo imposto da questo requisito.
6. L'applicazione funziona su più piattaforme (Chrome, Safari, Firefox...): questo requisito è necessario per poter utilizzare l'applicazione in tutti i dispositivi, che possono utilizzare browser diversi.
7. La transizione tra pagine non deve essere bloccante: essendo la nostra applicazione una Single Page Application, non si incorre nel caricamento durante le transizioni tra una pagina e l'altra, quindi non è stato necessario apportare alcuna modifica.
8. Ogni pagina ha un URL: questa opzione era già stata impostata nel routing di Angular.
9. L'utente deve essere avvertito nel caso in cui Javascript sia disabilitato: se l'utente non ha Javascript attivato, l'applicazione web non può essere eseguita: questo avvertimento è aggiunto nel file *index.html* aggiungendo il tag `<noscript>` con allegata la frase "Please enable Javascript to run this application."

Per testare i requisiti della PWA, abbiamo utilizzato il tool [LightHouse](#) fornito da Google Chrome, il quale testa l'applicazione eseguendo azioni che controllano la presenza di soluzioni adatte a soddisfare i requisiti imposti. Il tool assegnerà un punteggio da 0 a 100 in base ai requisiti soddisfatti.



Audits

Identify and fix common problems that affect your site's performance, accessibility, and user experience. [Learn more](#)



Device

☒ Mobile

☐ Desktop



Audits

☒ Performance

☒ Progressive Web App

☐ Best practices

☐ Accessibility

☐ SEO



Throttling

☒ Simulated Fast 3G, 4x CPU Slowdown

☐ Applied Fast 3G, 4x CPU Slowdown

☐ No throttling

☒ Clear storage

Run audits

Progressive Web App

These checks validate the aspects of a Progressive Web App, as specified by the baseline [PWA Checklist](#).



Additional items to manually check

3 audits



Passed audits

12 audits



100

Back-end

Server

Il primo passo è stato quello di determinare che tipo di server utilizzare e, dopo aver effettuato svariate ricerche, abbiamo deciso di implementare un server [Node.js](#). Questa scelta è stata fatta sulla base di due motivazioni: la prima fu l'aver trovato una API per MiBand implementata, appunto, in Node.js (scoperta che poi si rivelerà inutile, per problemi di compatibilità con la struttura della nostra applicazione); la seconda è la grande quantità di documentazione presente nella rete non solo per Node.js di per sé, ma anche per applicazioni che implementano Node.js assieme a [postgreSQL](#) e [Passport.js](#), che sono altre le altre due tecnologie utilizzate per lo sviluppo del back-end, rispettivamente riguardanti il database utilizzato e la middleware per l'autenticazione e per la sicurezza delle informazioni scambiate tra back-end e front-end (tratteremo la motivazione dietro la loro scelta più avanti).



Una volta determinato il tipo di server da utilizzare, ci sono voluti un paio di giorni per implementare un primo esempio di server funzionante che potesse ricevere e rispondere adeguatamente a dei primi esempi di chiamate http.

Non avendo allora ancora un client operativo, abbiamo deciso di utilizzare [Postman](#), un ottimo software per effettuare delle simulazioni di chiamate su RESTful API.

Era stato quindi creato un primo esempio di server, rappresentato dal file ["server.ts"](#) ed un primo servizio di routing delle richieste, rappresentato dal file ["routes.ts"](#).

Database

Il secondo passo è stato quello di costruire una corretta struttura dati nel quale conservare le informazioni riguardanti gli utenti dell'applicazione (dottori e pazienti).

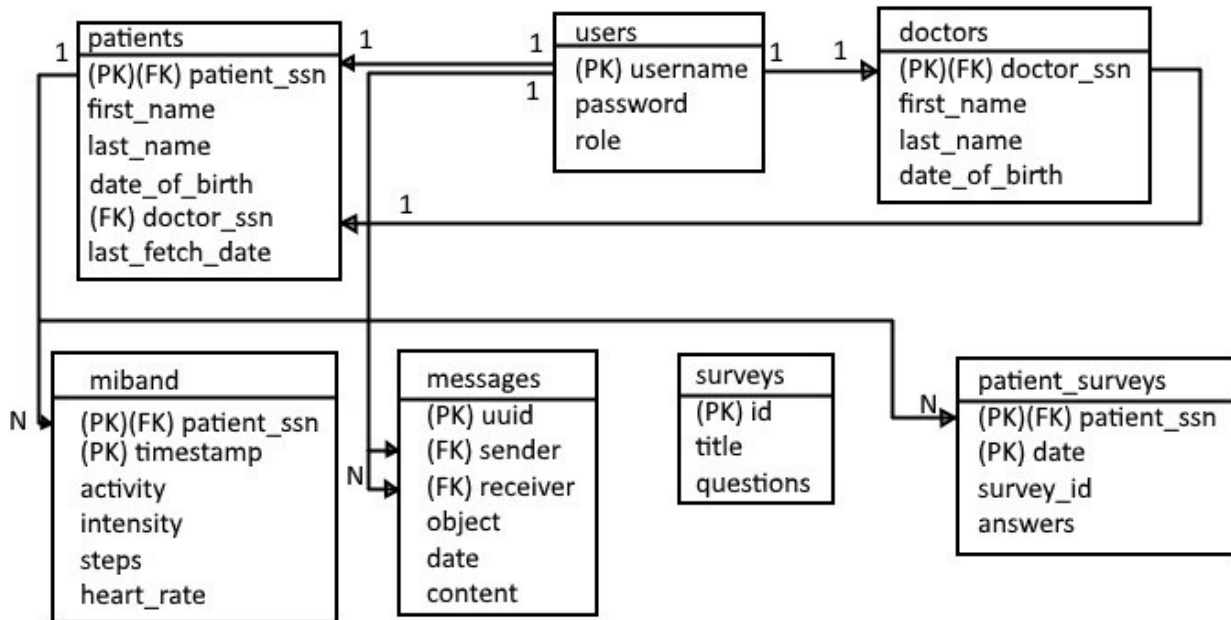
La ricerca del corretto database ha richiesto tre giorni di tempo, poiché cercavamo un database che offrisse le funzionalità di un database relazionale (per le relazioni utenti-pazienti e utenti-dottori, dottori-pazienti, pazienti-questionari, pazienti-MiBand, utenti-messaggi), ma che allo stesso tempo offrisse una gestione funzionale di serie temporali, rappresentate dai dati reperiti dai bracciali MiBand.

La ricerca è terminata con la scelta di utilizzare un database relazionale [postgreSQL](#) dotato di estensione [TimescaleDB](#) per la gestione delle serie temporali.



Timescale fornisce l'interfaccia di una singola tabella continua riguardante tutte le serie temporali, anche se in realtà essa è divisa in tanti singoli "chunks", con ogni chunk dimensionato automaticamente a runtime dal sistema per ottimizzare le performance in base alla capacità della macchina ed alle chiamate che vengono solitamente effettuate (esempio: selezione di dati miband nel range di 24h).

Struttura



Il terzo passo è stato quello di modificare il server in funzione della struttura del database, portando alla creazione delle cartelle "models" e "controllers".

In **"models"** sono presenti tutti i modelli di dati che vengono utilizzati per l'elaborazione e la gestione dei dati che vengono rinvenuti dalle chiamate al database ("patient.ts" contiene una classe rappresentante la struttura della tabella "patients", "doctors.ts" contiene una classe rappresentante la tabella "doctors" ecc.).

In **"controllers"** sono presenti delle classi che vengono utilizzate per estrapolare i parametri dalla chiamata http inviata dal client, creare la query che dovrà essere fornita al database assieme ai parametri appena estratti e ritornare a **"routes.ts"** le risposte restituite dal database, il quale si occuperà di inviarle al client, sia in caso di errore che di successo, dopo averle elaborate.

Ogni controller è rappresentato da una classe che estende la superclasse "TableCtrl", che fornisce alle sottoclassi i campi "dbManager", "sql", "params", "result" ed "error", che rappresentano la struttura base (e nel più dei casi, totale) dei vari controller. "TableCtrl" implementa inoltre due funzioni per il controllo della correttezza del codice fiscale e la sua correlazione con il nome utente (username), poiché nella nostra applicazione i due campi coincidono.

Per gestire le chiamate rivolte al database e le risposte restituite è stata implementata una classe apposita contenuta nel file **"dbmanager.ts"**: essa si occupa di effettuare un controllo sui parametri passati dai controller, aprire e chiudere la connessione con il database e gestire la risposta da esso restituita, generando eventualmente un errore da inviare al client.

Routes

Il quarto passo è stato quindi quello di creare tutte le routes necessarie per l'utilizzo completo dell'applicazione all'interno del file **"routes.ts"**, nonché l'implementazione delle funzioni che gestiscono l'elaborazione delle risposte inviate dal lato server al lato client, gestite in base al tipo di chiamata effettuata dal client (GET, POST e DELETE) e dal tipo di risposta del server (successo o errore).

Avevamo ora quindi un server correttamente funzionante e capace di funzionare come canale di comunicazione fra il lato client ed il database contenente tutti i dati necessari per delle prime

simulazioni di utilizzo.

Sicurezza

Il quinto ed ultimo passo è stato quello di assicurare la sicurezza nello scambio delle informazioni fra lato client e lato server tramite l'utilizzo di una sessione utente.

Per questo compito, abbiamo deciso di utilizzare [Passport.js](#), un middleware di autenticazione per Node.js che può essere utilizzato in ogni applicazione web basata su [Express](#), quale il nostro server.

Per l'implementazione di questo strumento è stato necessario definire una nuova strategia di autenticazione e le funzioni di serializzazione e deserializzazione dell'utente definite nel file ["passport.js"](#).



L'autenticazione viene effettuata tramite una chiamata al database con una query di tipo GET dalla tabella "users" utilizzando come parametro di ricerca l'username inviato dal client; viene poi utilizzata una funzione [bcrypt](#) per effettuare la comparazione tra la password inviata dal client e quella salvata nello stesso record dell'username (se esistente) trovato nel database.

Se l'autenticazione viene eseguita con successo, allora l'utente viene serializzato, tenendo salvati in sessione solamente username e ruolo (paziente o dottore), al fine di autenticare l'utente ogni volta che il client effettua una chiamata al server, poiché le chiamate che ogni tipo di utente è autorizzato a fare sono diverse (si prenda per esempio il caso in cui un paziente cerchi di ottenere la lista di tutti i pazienti del proprio dottore, o ancora, un dottore che cerchi di ottenere la lista di tutti i pazienti di un altro dottore); è stato quindi necessario creare delle funzioni di autorizzazione all'interno del file "routes.ts":

- **loginRedirect**: controlla se un paziente che sta effettuando una chiamata di POST login abbia già una sessione attiva ed in caso positivo restituisce un errore al client
- **isLoggedIn**: controlla se un paziente che sta cercando di effettuare una chiamata non di POST login abbia già una sessione attiva ed in caso negativo restituisce un errore al client.
- **isAuthorized**: controlla se l'utente è autorizzato ad effettuare una determinata chiamata al server, controllando l'attributo "role" della sessione e richiamando opportunamente le funzioni *findInDoctorPatientsList* e *checkPatientDoctor*
- **findInDoctorInPatientsList**: controlla se il paziente cercato da un utente dottore rientra nella propria lista dei pazienti
- **checkPatientDoctor**: controlla se il dottore passato come parametro di una chiamata di un paziente è effettivamente il dottore assegnato a quel paziente.

La deserializzazione dell'utente avviene nel momento in cui il client invia una chiamata http di GET logout.

E' stata infine effettuata una modifica alla configurazione del server (agendo su "server.ts") reindirizzando tutto il traffico http in https, per assicurare lo scambio sicuro dei dati.

Di seguito un grafico rappresentante il funzionamento generale del back-end:

