ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

PROGETTAZIONE E SVILUPPO DI UN'OFFLINE WEB APPLICATION CON ANGULAR SERVICE WORKER E INDEXEDDB

Elaborato in
PROGRAMMAZIONE AD OGGETTI

Relatore
Prof. MIRKO VIROLI

Co-relatore
Dott. STEFANO ZOFFOLI

Presentata da
LUCA RAGAZZI

Seconda Sessione di Laurea
Anno Accademico 2017 – 2018

PAROLE CHIAVE

Offline

Angular

Service Worker

IndexedDB

PWA

a chi crede in me,
alla mia famiglia.

Sommario

In questa tesi si trattano la progettazione e lo sviluppo di un'Offline Web Application utilizzabile come applicazione gestionale per le aziende. In particolare con questa applicazione sarà possibile gestire clienti, prodotti e listini dei prezzi. Nato su richiesta del committente dell'azienda Librasoft, il sistema si pone l'obiettivo di dimostrare come i dipendenti di un'azienda possono beneficiare di applicazioni che, con le opportune tecnologie, possono funzionare senza alcuna connessione Internet, permettendo all'utente di lavorare in modo sicuro e stabile, senza preoccuparsi dello stato della rete. Il software trova applicazione in contesti aziendali, ma le tecnologie prese in esame per il funzionamento offline possono essere riutilizzate in qualsiasi sistema. L'interfaccia utente è stata sviluppata per essere semplice ed intuitiva, ma allo stesso tempo elegante e funzionale.

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Uno dei primi compiti che mi sono stati assegnati durante il tirocinio curriculare presso l'azienda Librasoft[1] è lo studio di un framework[2] JavaScript[3]. Scelsi di studiare Angular, un framework JavaScript per lo sviluppo di Single Page Application e successivamente incominciai ad applicare le sue basi metodologiche e tecnologiche per la realizzazione di semplici applicazioni gestionali che sfruttassero la tecnologia sopra citata delle Single Page Application.

Avendo valutato positivamente il mio lavoro, il referente di Librasoft mi ha proposto di continuarlo e di integrarlo per un progetto di tesi. Lo scopo era quello di combinare le potenzialità di una Single Page Application con le caratteristiche di una Progressive Web Application, cioè la creazione di un'applicazione che potesse funzionare con scarsa connessione di rete e addirittura in assenza.

Nasce così *Jester*, applicativo aziendale per la gestione di clienti, prodotti e listini prezzi.

Seguendo un percorso suddivisibile in due macro-fasi, la tesi si sviluppa attraverso cinque capitoli. La prima fase, dedicata all'approfondimento delle tecnologie utilizzate per affrontare le problematiche dell'offline, è descritta nel primo capitolo. Gli ultimi quattro capitoli, invece, si focalizzano sul caso di studio, descrivendo la fase relativa all'analisi, alla progettazione e allo sviluppo di Jester avente le tecnologie introdotte nel primo capitolo della tesi.

[1] **Librasoft**: azienda Librasoft snc di Golinucci Thomas, Massari Piergiorgio e Zoffoli Stefano.

[2] **Framework**: architettura logica di supporto su cui un software può essere progettato e realizzato, facilitandone lo sviluppo da parte del programmatore.

[3] **JavaScript**: linguaggio di scripting utilizzato nella programmazione web lato client per la creazione di effetti dinamici interattivi.

Capitolo 1 Il primo capitolo esplora e descrive le tecnologie utilizzatesi nel progetto di tesi.

Capitolo 2 La seconda fase del percorso di tesi, che inizia con il secondo capitolo, affronta la fase di analisi dei requisiti relativi al caso di studio. In particolare, si mostrano le caratteristiche funzionali e non che il sistema dovrà implementare.

Capitolo 3 Il terzo capitolo affronta la progettazione e l'architettura del sistema precedentemente analizzato, risaltando i pattern progettuali utilizzatisi.

Capitolo 4 Il quarto capitolo entra nella fase di sviluppo e implementazione dell'applicazione presa in esame.

Capitolo 5 Il quinto, ultimo capitolo della tesi, tratta la fase di testing e validazione sul sistema, effettuando prove sperimentali su diversi scenari reali.

Capitolo 1

Background

Internet ha avuto un peso decisivo nella vita delle persone e nello sviluppo aziendale, tanto da poter essere visto come il nostro alleato quotidiano. Internet è il motore che consente di avere una connessione, in tempo reale, con il mondo esterno, per questo motivo la connettività è fondamentale per chiunque, dallo studente che vuole ascoltare la musica in streaming[1], al lavoratore che vuole proseguire il lavoro ovunque egli sia. Per queste considerazioni l'offline è un problema temuto da tutti.

E se esistessero tecniche per ovviare a questo problema e per far in modo che l'applicazione utilizzata dall'utente sia trasparente da una connessione di rete instabile?

In questo capitolo si fornisce una panoramica delle tecnologie utilizzate nello sviluppo del progetto di tesi, al fine di una migliore comprensione dei capitoli successivi.

La Sezione 1.1 tratta le tecnologie web utilizzate per l'implementazione dell'applicazione mediante approfondimento dei linguaggi di programmazione e dei framework frontend[2] e backend [3] utilizzatisi.

La Sezione 1.2 descrive brevemente il caso di studio su cui si concentra le tesi.

[1]**Streaming**: flusso di dati audio/video trasmessi da una sorgente a uno o più destinatari che viene riprodotto man mano che arriva a destinazione.

[2]**Frontend**: parte client di un sistema software che gestisce l'interazione con l'utente.

[3]**Backend**: parte server di un sistema software che ne gestisce l'amministrazione elaborando i dati generati dal frontend.

1.1 Single Page Application

Figura 1.1: Principali framework JavaScript per lo sviluppo di SPA [1]

Nel corso della mia esperienza ho utilizzato svariate applicazioni web. Tuttavia, queste condividevano la medesima caratteristica: un'interfaccia grafica che cambiava e si ricaricava ogni volta che si cliccasse su una specifica funzionalità.

Il limite del caricamento delle pagine interne è stato superato: all'interno di un'unica pagina web sarà presente l'intero software, per cui niente più attese per il caricamento delle pagine web.

Con **Single Page Application** (**SPA**) si intende un'applicazione web o un sito web che può essere usato o consultato su una singola pagina web con l'obiettivo di fornire un'esperienza utente più fluida e simile alle applicazioni desktop dei sistemi operativi tradizionali. In una SPA le risorse e il codice necessario (HTML[4], JavaScript e CSS[5]) sono caricati dinamicamente e aggiunti alla pagina quando necessario, di solito dopo determinate azioni compiute dagli utenti. Ricaricare la pagina ad ogni interazione provoca, oltre ad un peggioramento della User Experience[6], anche un'inutile ritrasmissione di elementi che

[4]**H**yper**T**ext **M**arkup **L**anguage: linguaggio di markup per la formattazione e impaginazione di documenti ipertestuali disponibili nel web.

[5]**C**ascading **S**tyle **S**heets: linguaggio usato per definire la formattazione di documenti HTML.

[6]**User Experience**: esperienza d'uso dell'utente che interagisce con il sistema. Include anche le sue percezioni personali circa l'utilità, l'efficienza e la semplicità d'utilizzo del sistema.

all'atto dell'invio sono già presenti nella memoria del browser.

Le SPA risolvono questi problemi delegando a JavaScript il rendering[7] dell'HTML e la gestione dei dati, eliminando così la necessità di ricaricare la pagina durante la sessione dell'utente. Un'applicazione con queste caratteristiche può godere di diversi vantaggi, tra cui massima produttività, massima flessibilità ed interoperabilità.

Le SPA sono costituite da un'architettura interna articolata e completa, simile al pattern MVC[8] o la sua derivazione più nuova MVVM[9].

Esistono diversi framework JavaScript in grado di offrire tutto il necessario a queste applicazioni, tra i più noti ricordo Angular, React e Vue.js. La nascita di questi framework ha permesso la creazione di pagine web fluide come le applicazioni desktop.

Angular e React sono i due framework frontend più popolari al momento e supportati, infatti sono stati sviluppati dalle due maggiori realtà web odierne, rispettivamente Google e Facebook. I due condividono la caratteristica di essere progetti open source[10]. Oggigiorno Angular conta su una community[11] molto più ampia del suo rivale React, soprattutto perché è stato rilasciato prima, rispettivamente nel 2010 e nel 2013.

Il fatto che Angular sia stato sviluppato da Google e che ha dietro le spalle una base d'utenza molto estesa, mi ha permesso di decidere di studiare ed applicare questo framework per lo sviluppo di Jester, caso di studio della tesi.

[7] **Rendering**: processo di generazione dei dati all'interno dell'interfaccia grafica.

[8] **M**odel-**V**iew-**C**ontroller: pattern architetturale della programmazione orientata agli oggetti in grado di separare la logica di presentazione dei dati dalla logica di business.

[9] **M**odel-**V**iew-**V**iew**M**odel: pattern architetturale, simile a MVC, che esalta la logica di presentazione dei dati, delegandole anche una parte di gestione della logica di business.

[10] **Open source**: software i cui autori rendono pubblico il codice sorgente, permettendo a chiunque di apportarvi modifiche per suo il miglioramento.

[11] **Community**: gruppo di persone con interessi comuni.

1.1.1 Angular

Figura 1.2: Logo di Angular [2]

Come già introdotto all'inizio, **Angular** è un framework JavaScript lato client per la progettazione di Single Page Application. È scritto in TypeScript[12] e supporta SASS[13]. Di seguito spiego nel dettaglio il funzionamento e l'architettura di questa tecnologia.

Nella realizzazione di un nuovo progetto è estremamente importante la creazione di un ambiente di sviluppo, operazione che potrebbe risultare complicata a causa della quantità di pacchetti software da integrare e per la definizione di diversi aspetti che potrebbero cambiare nel tempo, quindi di dipendenze software.

Per semplificare l'attività di setup dell'ambiente di sviluppo, il team di Angular ha sviluppato **Angular CLI**, un ambiente a riga di comando per creare la struttura di un'applicazione Angular già configurata. La configurazione di Angular CLI necessita l'utilizzo di un package manager[14]. Nello specifico è

[12] **TypeScript**: superset di JavaScript orientato agli oggetti che permette l'utilizzo di classi, interfacce e moduli.

[13] **S**yntactically **A**wesome **S**tyle **S**heets: estensione del linguaggio CSS per la realizzazione di fogli di stile, con l'ulteriore aggiunta di variabili, mixin, nesting, ereditarietà e altro ancora.

[14] **Package manager**: collezione di strumenti che automatizzano il processo di installazione e di configurazione di un software.

stato utilizzato **npm**, un package manager di JavaScript basato su nodejs[15].

Angular CLI può essere così installato con il seguente comando:

```
$ npm install -g angular-cli
```

Una volta installato l'ambiente a riga di comando, si può creare una nuova applicazione Angular digitando semplicemente:

```
$ ng new Jester
```

Questo comando genererà una cartella Jester con la seguente struttura:

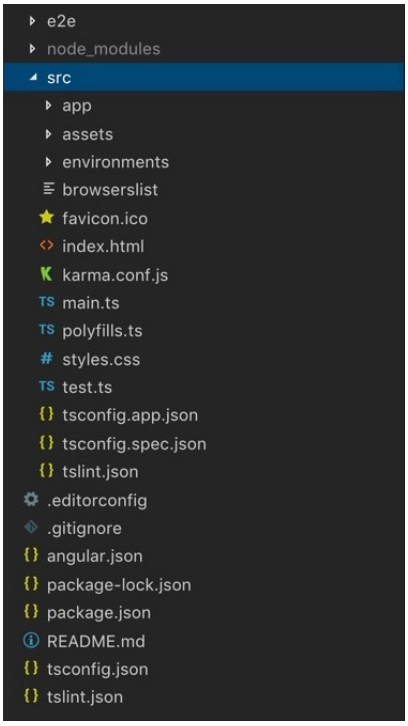


Figura 1.3: Struttura di un progetto Angular creato con Angular CLI

[15]**Nodejs**: piattaforma open source per l'esecuzione di codice JavaScript lato server.

Nella cartella *Jester* si trovano i file di configurazione fondamentali. Di seguito si fa un elenco delle cartelle e dei file principali generati da Angular CLI:

- **e2e**: cartella che contiene i test end-to-end di Protractor[16];
- **node_modules**: cartella che contiene i moduli nodejs necessari per il setup e l'esecuzione dell'applicazione;
- **src**: cartella che contiene il codice sorgente dell'applicazione;
- **protractor.conf.js**: file situato dentro la cartella e2e, contiene la configurazione di Protractor;
- **karma.conf.js**: file che contiene la configurazione di Karma[17];
- **tslint.json**: file che contiene la configurazione di TSLint[18];
- **angular.json**: file che contiene la configurazione del progetto Angular;
- **package.json**: file che contiene i riferimenti alle dipendenze e alle librerie del progetto;

Angular CLI consente di ottenere un'anteprima dell'applicazione e di visualizzarla in un browser tramite il comando:

```
$ ng serve -o
```

Esso manda in esecuzione un web server che consentirà l'accesso all'applicazione all'indirizzo *http://localhost:4200*.

Oltre alla visualizzazione in anteprima dell'applicativo, si ha a disposizione il *live reloading*, cioè la compilazione e il caricamento automatico dell'applicazione in seguito a modifiche ai file sorgenti, perciò non sorge il problema della ricarica della pagina.

L'utilità di Angular CLI si estende anche nella possibilità di generare interi frammenti di codice, tra cui generazione di componenti, servizi, classi e tanto altro.

[16]**Protractor**: framework di Angular per i test end-to-end.

[17]**Karma**: test runner di Angular.

[18]**TSLint**: strumento di analisi che controlla il codice TypeScript per la leggibilità, la manutenibilità e gli errori di funzionalità. È ampiamente supportato dai moderni editor e sistemi di compilazione.

Componenti

Essendo Angular un framework costituito da una solida architettura, uno dei concetti fondamentali su cui si basa è il concetto di *componente*. I componenti sono gli elementi costitutivi fondamentali delle applicazioni Angular. Oltre ad avere il controllo di una porzione dello schermo, ne visualizzano i dati e agiscono in base all'input dell'utente. Un'applicazione Angular è un insieme di componenti che interagiscono tra di loro. Ci sarà sempre un *root component* che, per convenzione, prende il nome di *AppComponent*.

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {
  title = 'Jester';
}
```

Listato 1.1: app.component.ts

Il decoratore[19] *@Component* identifica la classe immediatamente sotto esso come un componente e specifica i relativi metadati:

- **selector**: selettore che dice ad Angular di creare e inserire un'istanza di questo componente ovunque trovi il tag corrispondente nel file HTML;
- **templateUrl**: percorso relativo al modulo del file HTML di questo componente;
- **styleUrls**: percorso relativo al modulo del file SCSS[20] di questo componente;

Il codice sopra mostra "AppComponent". Bisogna specificare che una classe non è un componente vero e proprio finché non viene contrassegnata con il decoratore @Component.

Per quanto riguarda il template di un componente, non è altro che un file HTML che può contenere meccanismi e direttive Angular, che alterano l'HTML in base alla logica definita nell'applicazione.

[19] **Decoratore**: funzione che prende una classe e la arricchisce di specifiche funzionalità, dette meta-informazioni.

[20] **S**assy **CSS**: estensione della sintassi CSS basata su SASS.

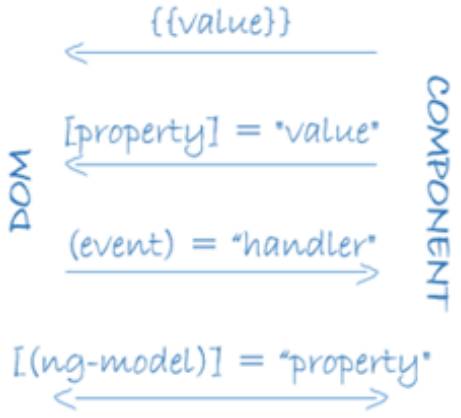
```
<div>
  <h1> Welcome to {{ title }}! </h1>
</div>
```

Listato 1.2: app.component.html

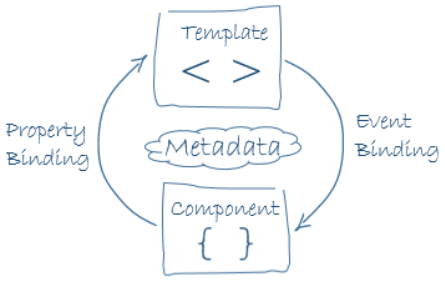
Nell'immagine riportata si può notare un meccanismo molto importante supportato da Angular, il *two-way data binding*, utilizzato per far comunicare un componente con il proprio template. Il data binding è composto da:

- **Event binding**: aggiornamento dei dati applicativi in seguito ad azioni di input dell'utente;
- **Property binding**: interpolazione dei dati di un componente visualizzabili nella view;

Di seguito sono mostrate due figure sul meccanismo del two-way data binding, prese dalla documentazione ufficiale di Angular.



(a) Tipologie di data binding



(b) Interazione tra componenti e template mediante data binding

Figura 1.4: Two-way data binding[3]

Alla luce di quanto detto, possiamo rappresentare il sistema come un albero di componenti, ognuno aventi le proprie funzionalità specifiche, per una migliore separazione della logica applicativa.

Moduli

Di notevole importanza, oltre l'AppComponent, è l'*AppModule*, che rappresenta il modulo di partenza dell'applicazione, il cosiddetto *root module*, il quale caricandosi permette l'avvio di tutti i componenti dell'applicazione.

Un'applicazione può essere composta da più moduli, ma l'importante è che sia presente il modulo di partenza, perché senza esso non è possibile avviare un'applicazione Angular. Un *modulo* è un contenitore di funzionalità che consentono di organizzare il codice di un'applicazione.

Per una migliore comprensione riporto il codice dell'AppModule generato da Angular CLI.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';

@NgModule({
    declarations: [ AppComponent ],
    imports: [ BrowserModule ],
    providers: [],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

Listato 1.3: app.module.ts

Come è possibile notare, un modulo non è altro che una classe a cui è stato applicato un decoratore *@NgModule* avente diverse proprietà tra cui:

- **declarations**: dichiara i componenti che appartengono al modulo;
- **imports**: contiene i moduli necessari per i componenti dichiarati;
- **providers**: dichiara tutti i servizi per renderli accessibili in tutte le parti dell'applicazione;
- **bootstrap**: vista principale dell'applicazione;

Servizi

Si può considerare un servizio come una classe implementata con uno scopo ben definito, che gestisce un specifica logica applicativa per il funzionamento dell'applicazione.

Angular distingue i componenti dai servizi per il semplice scopo di aumentare la modularità e la riusabilità all'interno del sistema da sviluppare. Per questo motivo un componente delega ai servizi determinate attività, per esempio gestire la comunicazione con il server e reperire i dati da esso.

Ogni servizio utilizza il decoratore *@Injectable()* che permette ai componenti di includere il servizio per poterlo utilizzare. La figura seguente rappresenta il concetto importante della *dependency injection*:

Figura 1.5: Dependency injection[4]

Il componente dovrà dichiarare all'interno del suo costruttore il servizio o i servizi di cui necessita.

1.2 Progressive Web Application

Le applicazioni web si pongono come valida alternativa alle tradizionali applicazioni native per diversi motivi:

- **Nessun software da installare e facilità di aggiornamento**: un'applicazione web si trova sul server per cui l'aggiornamento effettuato sul server è reso automaticamente disponibile a tutti gli utenti;

- **Accesso multipiattaforma**: l'accesso all'applicazione non dipende dal tipo di hardware e di sistema operativo utilizzato dagli utenti, per cui l'interfaccia è immediatamente disponibile ovunque;

- **Compatibilità con tutti i browser moderni**: l'applicazione può girare su qualsiasi browser moderno;

- **Scalabilità**: un'applicazione web non riscontra particolari problemi alle esigenze di un'azienda in crescita, se vi è stata effettuata una buona progettazione;
- **Sicurezza grazie ad HTTPS**: un'applicazione web servita da HTTPS[21] gode di una navigazione web sicura;

Nonostante le applicazioni web possano avere vantaggi rispetto alle applicazioni native, queste ultime rappresentano ancora la maggioranza delle applicazioni che vengono scaricate ogni giorno. Le applicazioni native sono applicazioni sviluppate specificamente per un sistema operativo, per esempio un'applicazione Mac non funzionerà su un sistema Windows e viceversa. Questo succede perché il linguaggio di programmazione è differente da un sistema operativo all'altro. Di seguito si elencano i punti di forza delle applicazioni native:

- **User Experience**: essendo veloci, affidabili e reattive assicurano un'ottima esperienza utente;
- **Accesso semplificato alle funzionalità del telefono**: comunicazione facilitata con accelerometro, fotocamera, etc...;
- **Notifiche push**: notifiche che permettono di avvisare gli utenti;
- **Connectionless**: molte applicazioni native non necessitano di Internet per funzionare. Questo può essere visto come un grande vantaggio, perché, nonostante sia il 2018, molte zone non sono coperte in maniera ottimale dalla rete Internet e il fatto di permettere agli utenti di accedere all'applicazione senza connessione è un vantaggio enorme da non dimenticare;

Entrambe le applicazioni web e le applicazioni native hanno pro e contro ed è impossibile decretare quale delle due sia meglio. Tutto dipende dalla situazione e dall'obiettivo del progetto.

Le **Progressive Web Application** (**PWA**) sono un ibrido tra le normali pagine web e le applicazioni native. Combinano, infatti, la versatilità del web con la velocità delle applicazioni mobile, ottenendo ottimi risultati sulla User Experience. Si utilizza il termina "Progressive" in quanto l'esperienza utente migliora con l'utilizzo dell'applicazione, in modo appunto progressivo.

Il browser proporrà all'utente di inserire l'icona del sito nella home screen, proprio come un'applicazione nativa, per poi poter navigare il sito con velocità

[21] **H**yper**T**ext **T**ransfer **P**rotocol **S**ecure: estensione del protocollo HTTP che garantisce una comunicazione sicura grazie ad una connessione criptata.

decisamente maggiore e soprattutto anche in assenza di connettività, grazie all'utilizzo di tecnologie che verranno elencate e discusse. Lo scopo è quello di consentire un'esperienza di navigazione eccellente ove prima non lo era.

Per procedere alla creazione di una PWA si necessita di avere due file da caricare nel sito: il **Manifest** e il **Service Worker**.

1.2.1 File Manifest

Il *Manifest* è un semplice file di formato JSON[22] che spiega al browser l'applicazione e il comportamento che essa deve avere quando è installata sul desktop o sulla home page di un device mobile. Un tipico Manifest include informazioni riguardo il nome dell'applicazione, le icone da utilizzare per installarla e l'URL[23] principale per avviarla.

```
{
  "name": "Jester",
  "short_name": "Jester",
  "theme_color": "#1976d2",
  "background_color": "#fafafa",
  "display": "standalone",
  "scope": "/",
  "start_url": "/",
  "description": "Gestionale PWA in Angular",
  "icons": [
    {
      "src": "assets/icons/icon-72x72.png",
      "sizes": "72x72",
      "type": "image/png"
    }
  ]
}
```

Listato 1.4: manifest.json

Come detto in precedenza, il file Manifest da informazioni sull'applicazione in un file JSON, in particolare i dettagli del sito web installato sulla home screen del device, o sul desktop, fornendo agli utenti un accesso più rapido e un'esperienza più ricca.

[22] **J**ava**S**cript **O**bject **N**otation: formato molto utilizzato per lo scambio dei dati in applicazioni client-server.

[23] **U**niform **R**esource **L**ocator: indirizzo di una risorsa in Internet, tipicamente presente su un host server.

Il Manifest sopra illustrato definisce diverse caratteristiche dell'applicativo:

- **name e short-name**: fornisce un nome per il sito quando viene visualizzato all'utente, *short-name* è utilizzato quando non c'è sufficiente spazio per mostrare il nome intero dell'applicazione;
- **background-color e theme-color**: definiscono rispettivamente il colore di background e il colore del tema di default dell'applicativo, sono usufruiti solo quando non si è ancora caricato il foglio di stile CSS o SCSS;
- **display**: definisce la modalità di visualizzazione preferita dagli sviluppatori per il sito web;
- **scope**: definisce il contesto di navigazione del sito web. In questo modo si limitano le pagine che possono essere visualizzate mentre viene applicato il Manifest;
- **start-url**: l'URL che viene caricato quando un'utente carica l'applicazione, per esempio quando viene aggiunta nella schermata iniziale;
- **description**: fornisce una descrizione generale di ciò che fa il sito web;
- **icons**: specifica una serie di immagini che possono fungere da icone delle applicazioni, per esempio possono essere utilizzati per rappresentare l'applicazione;

1.2.2 Service Worker

Le PWA devono essere veloci, installabili, grazie al file Manifest, ma soprattutto devono poter funzionare offline e con connessione lenta e instabile. Per raggiungere questo obiettivo, è necessario memorizzare nella cache[24] la shell dell'applicazione utilizzando un *Service Worker*, in modo che essa sia sempre rapidamente disponibile e affidabile.

Il Service Worker è un file JavaScript che il browser esegue in background. Esso può compiere diverse operazioni, tra cui la sincronizzazione in background, le notifiche push e la gestione delle richieste di rete per mezzo della cache.

Nelle sezioni successive si descrive il suo funzionamento e il suo ciclo di vita all'interno di un'applicazione.

[24] **Cache**: area di memoria estremamente veloce con basse capacità di memoria che ha lo scopo di velocizzare l'esecuzione dei programmi.

Funzionamento e ciclo di vita

Il funzionamento di un Service Worker è simile a quello di un server proxy, ovvero fa da intermediario tra il client e il server in modo da rendere migliore la User Experience, immagazzinando i dati di una pagina web in una cache, consentendo esperienze offline agli utenti che interagiscono con il sistema.

Sono necessari due requisiti affinché un Service Worker possa installarsi in un browser e funzionare correttamente:

- **Protocollo HTTPS**: l'utilizzo di tale protocollo garantisce che non ci sia alcun rischio di sicurezza per il Service Worker, siccome può eseguire delle operazioni molto potenti sul browser. Se la pagina web è in HTTP, il Service Worker non può essere installato. Un'altro modo per installarlo è utilizzando il localhost[25];

- **Supporto del browser**: al momento i browser che permettono il supporto dei service worker sono molteplici, tra cui Google Chrome, Mozilla Firefox, Opera e Microsoft Edge;

Di seguito un'immagine sul funzionamento del Service Worker.

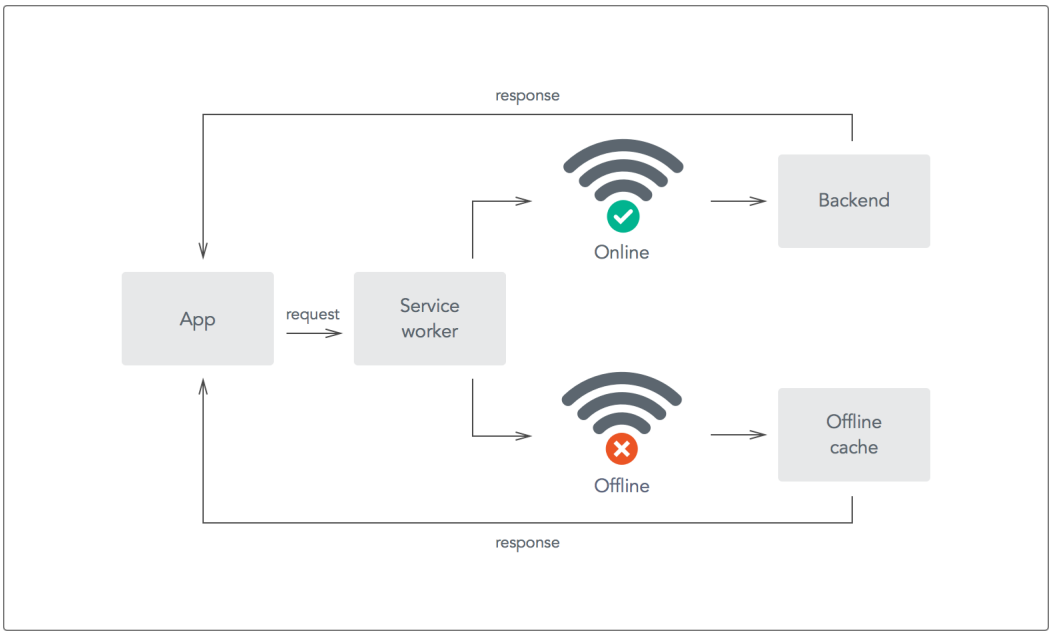


Figura 1.6: Service Worker: Funzionamento[5]

[25]**Localhost**: nome associato all'indirizzo della macchina locale su cui sono in esecuzione i programmi.

Il Service Worker ha un ciclo di vita diverso e completamente separato dalla pagina web. Il ciclo di vita avviene in quattro fasi:

1. **Registrazione**: operazione effettuata nel codice JavaScript dell'applicazione. La registrazione farà avviare al browser l'installazione in background del Service Worker;

2. **Installazione**: diverse risorse statiche sono memorizzate nella cache. Se tutti i file vengono correttamente memorizzati nella cache allora il Service Worker verrà installato, altrimenti non verrà avviato (in tal caso l'installazione sarà ritentata la volta successiva);

3. **Attivazione**: gestione del funzionamento delle vecchie cache;

4. **Fetch**: il Service Worker ottiene il controllo di tutte le pagine che rientrano nel suo ambito e gestirà gli eventi che si verificano dall'applicazione (come le richieste di rete), oppure sarà chiuso per risparmiare memoria, entrando nella fase **Terminated**;

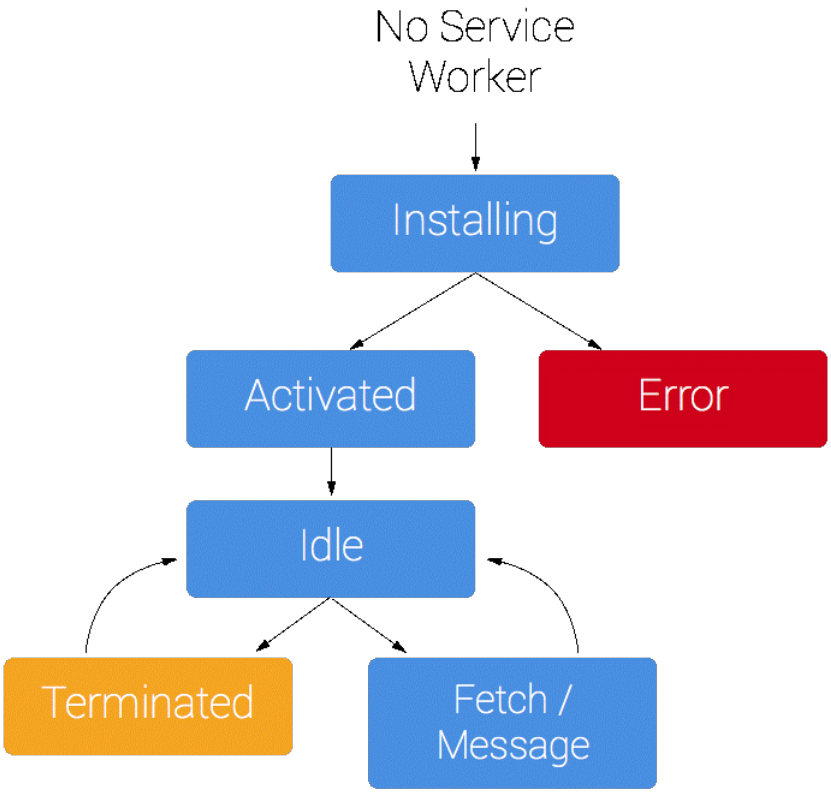


Figura 1.7: Service Worker: Ciclo di vita [6]

1.2.3 Service Worker in Angular

Le applicazioni Angular essendo delle SPA beneficiano particolarmente dei vantaggi del Service Worker. L'*Angular Service Worker* è progettato per ottimizzare l'esperienza utente al fine di utilizzare un'applicazione su una connessione di rete lenta o inaffidabile. I suoi obiettivi sono i seguenti:

- Caching dell'applicazione;
- Aggiornamento in background della versione dell'applicazione;
- Caricamento dell'ultima versione disponibile nella cache al momento del refreshing[26] dell'applicazione;
- Scaricamento delle risorse solo se modificate, per cui conservazione della larghezza di banda quando possibile;

Per supportare questi comportamenti, l'Angular Service Worker carica un file Manifest dal server, precedentemente generato dall'Angular CLI.

La versione 6 di Angular e la 1.6 di Angular CLI contengono già il framework e gli strumenti di sviluppo per generare il file Manifest e aggiungere il Service Worker nell'applicativo.

Per aggiungere il Service Worker ad un'applicazione che si sta creando da zero, bisogna utilizzare il seguente comando di Angular CLI:

```
$ ng new Jester --service-worker
```

In alternativa, se l'applicazione è già presente, è possibile aggiungere il Service Worker mediante il comando:

```
$ ng add @angular/pwa --project Jester
```

Una volta mandato in esecuzione il comando nella shell, Angular CLI crea in autonomia il file Manifest, contenente tutte le informazioni circa l'applicazione, e registra il Service Worker.

L'Angular Service Worker sarà disponibile solo in modalità di produzione, la cossiddetta *production mode*. Per questo motivo si necessita di eseguire un ulteriore comando:

```
$ ng build --prod
```

Questo comando renderà l'applicazione disponibile dentro la cartella *dist*.

Una domanda può sorgere spontanea: cosa ha creato esattamente l'Angular CLI?

[26] **Refreshing**: aggiornamento della pagina web.

I file più importanti dentro la cartella "dist" sono essenzialmente due:

- **ngsw-worker.js**: file dell'Angular Service Worker contenente tutte le direttive e il ciclo di vita. Utilizza il modulo *ServiceWorkerModule* che attiverà il caricamento di questo file, chiamando il metodo *navigation.serviceWorker.register()*;
- **ngsw.json**: file di configurazione che verrà utilizzato dall'Angular Service Worker. All'interno ci sono tutte le informazioni circa il metodo di caching, ovvero il service worker saprà quali file dovrà salvare nella cache e quali no. Il caricamento di questi file avverrà in background. La volta successiva in cui l'utente aggiorna la pagina, l'Angular Service Worker intercetterà le richieste HTTP e utilizzerà i file memorizzati nella cache anziché recuperarli dalla rete;

Di seguito un frammento del file di configurazione, dove si mostrano le risorse esterne da salvare nella cache:

```
{
  "dataGroups": [
    {
      "name": "",
      "patterns": [
        "\\/angular-jestergest-new\\/products\\.json",
        "\\/angular-jestergest-new\\/products\\/view\\/:id\\.json",
        "\\/angular-jestergest-new\\/price_lists\\.json",
        "\\/angular-jestergest-new\\/price_lists\\/view\\/:id\\.json",
        "\\/angular-jestergest-new\\/clients\\.json",
        "\\/angular-jestergest-new\\/clients\\/view\\/:id\\.json"
      ],
      "strategy": "performance",
      "maxAge": 0,
      "version": 1
    }
  ]
}
```

Listato 1.5: ngsw.json

1.3 Tecnologie Web: Client

Si espongono di seguito le principali tecnologie web utilizzate per la realizzazione frontend del progetto.

1.3.1 Angular Material

Angular Material è una collezione di componenti grafici di Material Design[27] progettati direttamente da Google. Possiamo pensare ad Angular Material come un framework simile a Bootstrap[28], ma ideato interamente per applicazioni Angular e questo permette un notevole vantaggio.

Con il rilascio di Angular 6 l'utilizzo di questo framework è diventato molto più semplice grazie a diverse features introdotte nell'Angular CLI, dal quale è possibile creare diversi componenti grafici in modo molto intuitivo. Prima di tutto bisogna installare il pacchetto mediante l'Angular CLI con il seguente comando:

```
$ npm install --save @angular/material @angular/cdk
    @angular/animations
```

Successivamente bisogna aggiungerlo nel progetto:

```
$ ng add @angular/material
```

Per utilizzare i diversi componenti grafici di Angular Material è necessario importarli tutti in una classe modulo.

Di seguito si riporta l'esempio specifico dell'applicazione Jester, che utilizza un numero elevato di componenti di Angular Material (non elencati tutti per motivi di spazio):

```
import {
  MatToolbarModule,
  MatButtonModule,
  MatSidenavModule,
  ...
  MatSnackBarModule,
  MatDatepickerModule,
  MatNativeDateModule,
} from '@angular/material';
```

[27] **Material Design**: design sviluppato da Google. Le sue regole di progettazione si concentrano su layout a griglia, animazioni, transizioni ed effetti di profondità.

[28] **Bootstrap**: framework per uno sviluppo rapido di applicazioni web grazie all'inclusione di frammenti di codice HTML e CSS già creati.

```
import { NgModule } from '@angular/core';

@NgModule({
  imports: [
    MatToolbarModule,
    MatButtonModule,
    MatSidenavModule,
    ...
    MatSnackBarModule,
    MatDatepickerModule,
    MatNativeDateModule
  ]
})

export class MaterialModule {}
```

Listato 1.6: material.module.ts

Un altro passo importante è quello di includere un tema nell'applicazione. Angular Material prevede quattro temi di default, già utilizzabili con l'importazione nel file di stile principale dell'applicazione: *style.scss*:

```
@import '~@angular/material/prebuilt-themes/indigo-pink.css';
```

Listato 1.7: style.scss

1.3.2 IndexedDB

IndexedDB non è un database relazionale ma un Object Store. Mentre un database relazionale organizza i dati in tabelle composte da righe e colonne e prevede un linguaggio specializzato per gestire i dati (SQL[29]), un Object Store consente la permanenza diretta di oggetti e prevede l'utilizzo di indici per recuperarli in modo efficace. Tra le altre caratteristiche, c'è anche la disponibilità di API[30] sincrone e asincrone.

Un rivale dell'IndexedDB è il Web Storage, che include due meccanismi diversi di archiviazione:

- **sessionStorage**: salva le coppie chiave-valore per la durata della sessione del browser;

[29] **S**tructured **Q**uery **L**anguage: linguaggio standard per database relazionali, progettato per creare e modificare schemi all'interno del database.

[30] **A**pplication **P**rogramming **I**nterface: insieme di procedure che permettono agli sviluppatori software di accedere a determinate funzioni o dati.

- **localStorage**: stesso funzionamento di sessionStorage, ma i dati salvati permangono anche tra una sessione e l'altra;

Il fatto che Web Storage è sincrono e che ha il limite di poter salvare solo stringhe, ha permesso di scegliere IndexedDB come storage offline dell'applicazione.

Nel progetto si utilizza l'IndexedDB per memorizzare tutti i dati (prodotti, clienti e listini prezzi) in un database locale, il quale ha consentito di poter effettuare le più comuni operazioni, anche in assenza di connessione di rete.

1.3.3 RESTful API

Dovendo gestire una componente client avente molte richieste da inviare al server, si è reso necessario l'utilizzo di API.

Le API non sono altro che un insieme di tecniche che consentono di collegarsi ad un server esterno ed eseguire alcune operazioni sui dati che esso contiene.

Per **RESTful** (**REpresentational State Transfer**), invece, si intende una collezione di una serie di approcci che definiscono le regole con cui i dati sono trasmessi dal client al server.

REST descrive ogni tipo di interfaccia capace di trasmettere dati per mezzo del protocollo HTTP, senza l'appoggio di tecnologie ausiliarie come cookie[31] o protocolli per lo scambio dei messaggi. Questo ha permesso la creazione della comunicazione tra client e server rendendola completamente stateless[32], dove ogni richiesta è svincolata dalle altre.

Le API consentono di effettuare una qualsiasi delle seguenti operazioni, denominate **CRUD**:

- **Create**: creazione di un elemento mediante richiesta **POST**;
- **Read**: lettura di un elemento mediante richiesta **GET**;
- **Update**: modifica di un elemento mediante richiesta **PUT**;
- **Delete**: eliminazione di un elemento mediante richiesta **DELETE**;

Queste operazioni funzionano in modo del tutto trasparente rispetto al database sul quale vengono inviate.

In Jester sono state inserite delle API per tutte le operazioni CRUD delle entità gestite (clienti, prodotti e listini prezzi).

[31] **Cookie**: utilizzato dalle applicazioni web lato server per recuperare informazioni sul lato client.

[32] **Stateless**: tipologia di diversi protocolli di rete, come HTTP. La connessione viene chiusa una volta terminata lo scambio richiesta/risposta senza il salvataggio dei dati della sessione di comunicazione.

1.4 Tecnologie Web: Server

Si espongono di seguito le principali tecnologie web utilizzate per la realizzazione backend del progetto.

1.4.1 CakePHP

Una delle caratteristiche di Jester è poter comunicare con il server aziendale, già presente e creato dall'azienda Librasoft.

Per modellare il server dell'applicazione è stato utilizzato **CakePHP**.

CakePHP è un framework basato su PHP[33], il quale:

- Potenzia e velocizza lo sviluppo di applicazioni web;
- Semplifica l'interfacciamento con il database;
- Si basa sull'architettura MVC;

Come database, con gli opportuni metodi di connessione, è possibile scegliere tra Mssql, Oracle, SQLite e infine MySQL, quello utilizzato nel progetto.

MySQL è un RDBMS[34] open source composto da un client a riga di comando e da un server. Esso è uno dei DBMS più utilizzati, per il semplice motivo che è supportato da moltissimi sistemi e linguaggi di programmazione. Inoltre, esistono piattaforme come WAMP[35] e LAMP[36] che incorporano MySQL per l'implementazione di server per gestire siti web dinamici. Uno strumento fondamentale nell'interazione con i DBMS relazionali è il linguaggio SQL. Si tratta di un formalismo che permette di indicare le operazioni che il DBMS deve svolgere sul database. Tramite SQL si può attivare qualsiasi tipo di operazione, ma le principali e le più frequenti sono tipicamente: inserimento, lettura, modifica ed eliminazione dei dati.

Il server è stato progettato con un'architettura REST, un vero e proprio standard per la creazione di Web API.

In questa tesi non si entra nello specifico sul funzionamento di CakePHP, poiché non si è implementato il backend. Nonostante ciò, riporto le caratteristiche del Middleware e della sicurezza.

[33]**PHP** **H**ypertext **P**reprocessor: linguaggio di scripting interpretato e originariamente concepito per la programmazione di pagine web dinamiche. Il suo nome è un acronimo ricorsivo.

[34]**R**elational **D**ata**B**ase **M**anagement **S**ystem: sistema per la gestione di basi di dati basato sul modello relazionale.

[35]**WAMP**: piattaforma software di sviluppo web realizzata con Windows, Apache, MySQL e PHP.

[36]**LAMP**: piattaforma software di sviluppo web realizzata con Linux, Apache, MySQL e PHP.

Un *Middleware* funziona da wrapper[37] per un'applicazione, con lo scopo di gestire determinate richieste e generare le relative risposte. Visivamente è possibile vedere l'applicazione avvolta dagli strati del Middleware, come nella figura seguente:

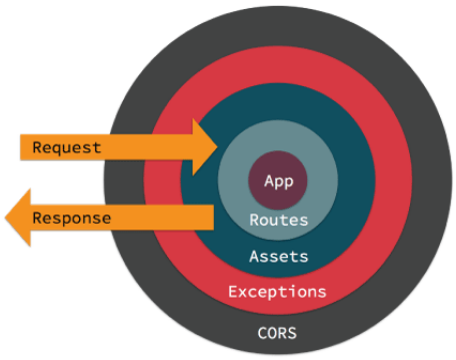


Figura 1.8: Middleware in CakePHP[7]

L'immagine mostra come il Middleware gestisce le richieste prima che arrivano all'applicazione. Il Middleware del routing, per esempio, è incaricato ad applicare il percorso corretto delegando il compito al Controller specifico su cui è indirizzata la richiesta.

CakePHP è caratterizzato dalla protezione Middleware **CSRF** (**Cross Site Request Forgery**), con la quale si proteggono tutte le azioni dell'applicazione. In merito a questo, ogni richiesta HTTP inviata al server deve opportunamente contenere nel suo header un token CSRF per poter comunicare con il server.

Un altro problema della comunicazione client-server di Jester è stato il **CORS** (**Cross-Origin Resource Sharing**). Il problema risiede nel fatto che un'applicazione web in esecuzione su un dominio sta cercando di accedere alle risorse all'interno di un server di un dominio diverso. Per ovviare il problema si è dovuto configurare in maniera opportuna il server e tutte le richiesta HTTP inviate dal client, con l'introduzione di header HTTP addizionali.

[37] **Wrapper**: dal verbo inglese to wrap (avvolgere), indica un modulo software che ne "riveste" un altro.

1.5 Caso di studio: Jester

Jester è un'applicazione gestionale a scopo aziendale ideata per la gestione dei clienti fino a quella dei prodotti con i relativi listini. Essa verrà progettata con le ultime tecnologie web, per renderla efficiente, funzionale e affidabile sia con la connessione di rete che soprattutto in assenza.

Per il frontend del sistema verrà utilizzato il framework Angular abbinato al Service Worker e all'IndexedDB, tale da rendere Jester una vera e propria applicazione comparabile a quelle native, avente le caratteristiche di una Single Page Application e di una Progressive Web Application.

Capitolo 2

Analisi

Scopo della tesi è lo studio e l'implementazione della parte client di un gestionale aziendale per l'amministrazione interna dell'azienda, in particolare per la gestione di clienti, prodotti e listini prezzi. Tale applicazione deve poter funzionare correttamente anche in assenza di connessione Internet, per cui in modalità offline, per poter garantire agli utenti e ai dipendenti una sessione di lavoro sicura e indipendente dallo stato della rete.

2.1 Analisi dei requisiti

Lo sviluppo dell'applicazione inizia con un elenco dei requisiti che il sistema, come prodotto finito, deve soddisfare. Tali requisiti sono stati commissionati dal mio referente dell'azienda Librasoft, dove ho sostenuto il tirocinio per tesi.

Il sistema che si vuole realizzare, per raggiungere lo scopo fissato, dovrà rispettare determinati requisiti. Questi saranno elencati di seguito, divisi in requisiti funzionali (funzionalità che il sistema dovrà mettere a disposizione), requisiti non funzionali (proprietà del sistema non direttamente correlate al suo comportamento funzionale che esprimono dei vincoli o delle caratteristiche di qualità), requisiti implementativi (proprietà del processo di sviluppo del sistema) e requisiti tecnologici (tecnologie utilizzate per lo sviluppo del sistema).

2.1.1 Requisiti funzionali

Autenticazione al sistema. Il sistema deve poter identificare la sessione di lavoro dell'utente come attuale utente del sistema. In particolare, l'utente per poter iniziare la sessione di lavoro deve autenticarsi mediante invio di credenziali di login (email e password). Inoltre, deve esserci la possibilità di effettuare il logout al termine della sessione lavorativa.

Interazione con la dashboard. In seguito all'autenticazione, l'utente potrà visionare la propria dashboard, dalla quale è possibile interagire con il sistema. Nella dashboard devono essere presenti scorciatoie alle operazioni di creazione di clienti, prodotti e listini, oltre alla visualizzazione del numero di ogni entità salvata nel database.

Visualizzazione dei clienti, prodotti e listini. L'applicazione deve permettere all'utente di poter visualizzare, in pagine separate, i propri prodotti, clienti e listini. In ogni pagina di visualizzazione l'utente deve poter cercare un elemento specifico mediante filtri di ricerca e poter ordinare la visualizzazione in base alle caratteristiche degli elementi.

Operazioni CRUD su clienti, prodotti e listini. Per ogni entità è possibile applicare delle operazioni di base. In particolare, possibilità di effettuare le seguenti azioni: creazione, modifica ed eliminazione.

Sessione offline. L'utente deve poter continuare la sessione di lavoro iniziata, anche in assenza di connessione Internet. In particolare, dovrà potersi salvare localmente tutte le modifiche effettuate offline e sincronizzarle con il server non appena la rete ritorna disponibile.

2.1.2 Requisiti non funzionali

Interoperabilità. Abilità del sistema di coesistere e cooperare con altri sistemi. L'applicazione, per funzionare correttamente, deve comunicare con il server aziendale. A tale proposito bisogna risolvere i problemi legati all'interfacciamento tra due applicazioni situate in domini diversi. In particolare, risoluzione dei problemi legati al CORS e utilizzo di token CSRF specifici.

Affidabilità e Robustezza. L'applicativo dovrà comportarsi in modo ragionevole anche in circostanze non previste dalle specifiche di progetto. L'utente dovrà poter dipendere dal software, che dovrà produrre risultati corretti in ogni situazione. In particolare, in condizioni di connessione Internet lenta o

assente. Inoltre, l'applicativo dovrà garantire l'integrità dei dati, per cui gestire la sicurezza nel migliore dei modi, tramite uso di hashing delle password e token.

Usabilità. L'applicazione dovrà poter essere utilizzata in modo semplice e immediato. Andrà posta molta attenzione sulla realizzazione di un'interfaccia utente che permetta un'interazione e un'apprendimento del funzionamento che sia il più rapido possibile, in modo da facilitare gli utenti, anche quelli meno esperti nel settore informatico. Il sistema deve offrire quindi diversi servizi all'utente, tra cui:

- **Apprendibilità**: semplicità nell'apprendimento e nell'utilizzo del sistema;
- **Velocità**: velocità con la quale si effettuano le operazioni desiderate;
- **Soddisfazione**: soddisfazione del cliente nell'utilizzo dell'applicativo;
- **Facilità di navigazione**: semplicità di orientarsi dentro l'applicazione;
- **Memorabilità**: possibilità di riutilizzare l'applicativo dopo svariato tempo senza ulteriore training;
- **Prevenzione degli errori**: il software non deve produrre errori incorreggibili, non prodotti direttamente dall'utente;

Portabilità. L'applicazione dovrà poter funzionare su più piattaforme. Trattandosi di un'applicazione web dovrà poter essere eseguita sui più comuni browser. Inoltre, l'applicativo deve poter essere salvato ed eseguito dalla home page del proprio dispositivo, per un avvio del software più semplice e immediato.

2.1.3 Requisiti implementativi

Manutenibilità. Il sistema dovrà essere realizzato in modo da facilitarne la sua manutenzione al fine di semplificare la ricerca degli errori, adattarlo ai cambiamenti del dominio applicativo e permettere l'eventuale aggiunta di nuove funzionalità. Affinché questo sia possibile il software dovrà essere caratterizzato da modularità e quindi composto da un insieme di moduli[1] e implementato con i più comuni pattern progettuali, con lo scopo di separare i concetti della logica di business.

[1]**Moduli**: componenti di base del sistema con funzionalità strettamente legate.

Riusabilità. Il software deve poter essere utilizzato, in tutto o in parte, anche in altri sistemi, per cui è fondamentale rendere gli aspetti della logica applicativa in un'ottica di riutilizzo, non concentrandosi sul singolo caso, ma sulla generalizzazione della soluzione.

2.1.4 Requisiti tecnologici

Il sistema da sviluppare deve essere una Web Application e inoltre deve possedere le caratteristiche di una Single Page Application e di una Progressive Web Application, con lo scopo di dimostrare quanto una Web Application, avente le caratteristiche menzionate, sia potente, efficiente e versatile.

2.2 Modello del dominio

A partire dai requisiti individuati precedentemente, è utile formalizzare il modello del dominio che può essere preso come punto di partenza per lo sviluppo del software. Un modello del dominio descrive le varie entità che fanno parte nel sistema e le loro relazioni, risaltando i concetti fondamentali.

Siccome il sistema si basa su un'architettura REST, il modello del dominio è formalizzato tramite il concetto di risorsa. Viene perciò presentato sotto forma di diagramma ad albero mostrando un'organizzazione gerarchica delle risorse dell'applicazione, rappresentate dagli URI[2].

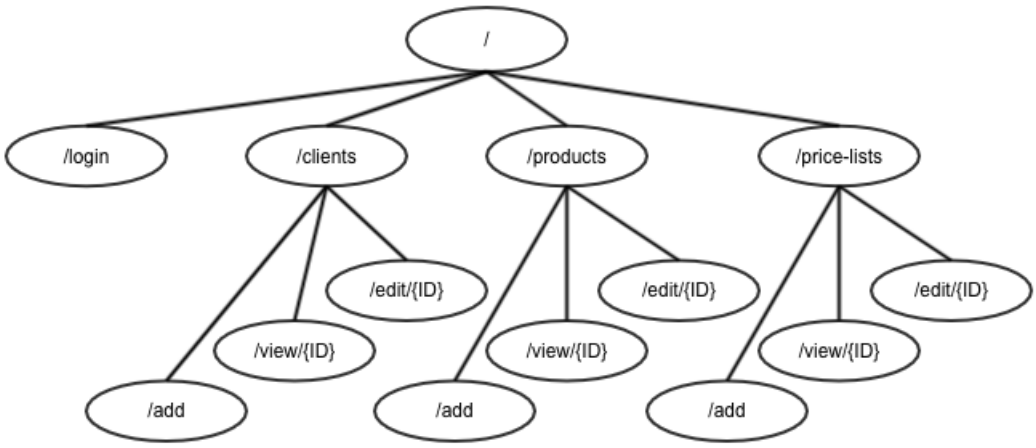


Figura 2.1: Modello del dominio del sistema

[2]**U**niform **R**esource **I**dentifier: sequenza di caratteri che identifica univocamente una risorsa.

2.3 Casi d'uso

In UML i diagrammi dei casi d'uso sono dedicati alla rappresentazione dei requisiti funzionali di un sistema dal punto di vista dell'utilizzatore, noto come attore.

Lo studio dei requisiti di sistema precedentemente analizzati ha permesso la creazione del seguente diagramma dei casi d'uso:

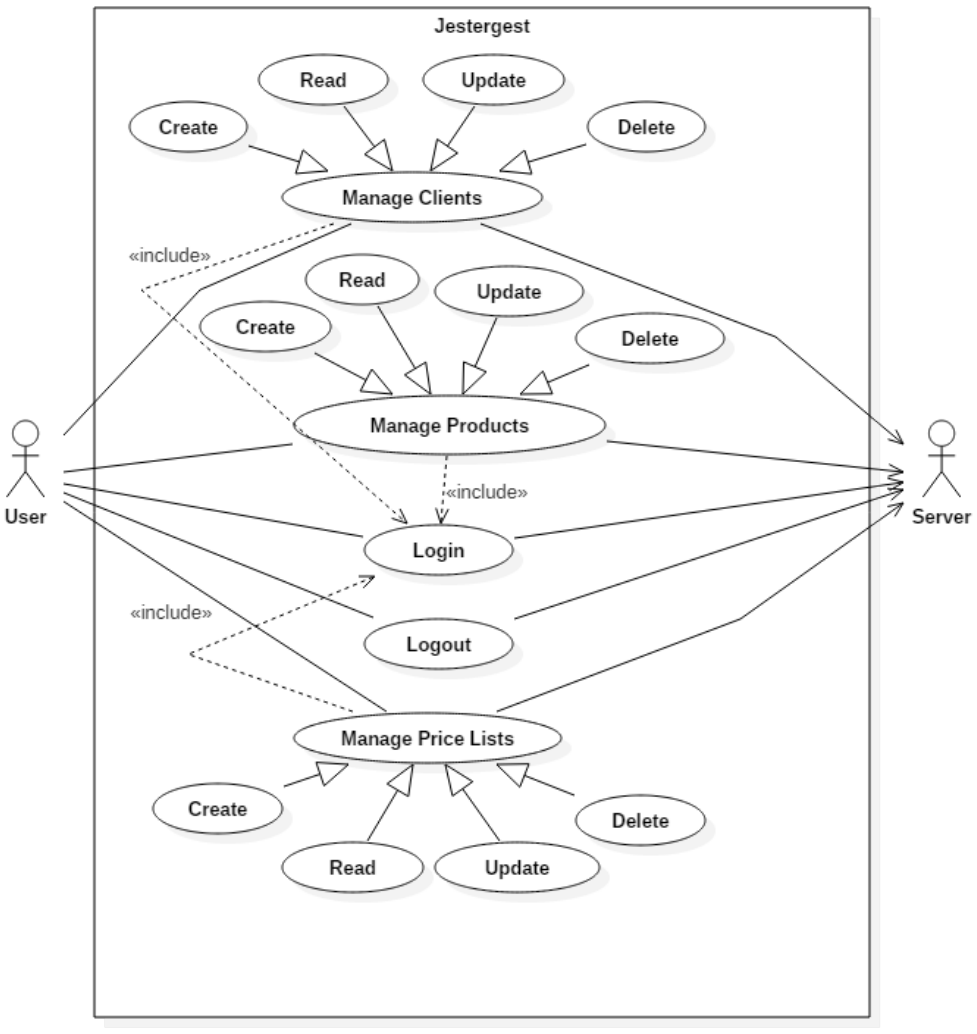


Figura 2.2: Jester: Diagramma dei casi d'uso

Seguendo ciò che è stato descritto nella fase di analisi dei requisiti, il sistema dovrà permettere all'utente una fase di autenticazione per poter accedere all'applicazione vera e propria.

Una volta autenticatosi, l'utente potrà interagire con il sistema ed eventualmente effettuare il logout della sessione. In particolare, l'utente potrà gestire clienti, prodotti e listini e per ognuno di essi effettuare operazioni di creazione, visualizzazione, modifica ed eliminazione.

2.4 Scenari

Gli scenari rappresentano delle astrazioni di utilizzo del sistema, da testare in fase di validazione del sistema.

In questo progetto sono stati considerati tre scenari in quanto rappresentativi. Ognuno di essi dovrà essere validato in fase di testing. Le operazioni da testare nei diversi scenari sono semplici, ma sufficienti per garantire il corretto funzionamento del software.

Scenario 1 - Sessione di lavoro completamente online

Tabella 2.1: Scenario 1

Scenario 1	Sessione di lavoro completamente online
Descrizione	L'utente deve poter effettuare una sessione di lavoro in condizioni normali di connessione di rete
Attore	Utente
Precondizioni	Connessione Internet presente

Il primo scenario descrive la sessione di lavoro dell'utente effettuata con condizioni normali di connessione Internet. In particolare, in questo scenario, l'utente dovrà effettuare le seguenti operazioni:

1. Creazione di un cliente
2. Modifica di un cliente
3. Eliminazione di un cliente

Scenario 2 - Sessione di lavoro sia online che offline

Tabella 2.2: Scenario 2

Scenario 2	Sessione di lavoro sia online che offline
Descrizione	L'utente deve poter effettuare una sessione di lavoro in condizioni di connessione di rete sia presente che assente
Attore	Utente
Precondizioni	Connessione Internet presente

Il secondo scenario descrive la sessione di lavoro dell'utente iniziata con condizioni normali di connessione Internet e successivamente diventata offline. In particolare, in questo scenario, l'utente dovrà effettuare le seguenti operazioni:

1. Creazione di un prodotto
2. Modifica di un prodotto
3. Eliminazione di un prodotto

Scenario 3 - Sessione di lavoro completamente offline

Tabella 2.3: Scenario 3

Scenario 3	Sessione di lavoro completamente offline
Descrizione	L'utente deve poter effettuare una sessione di lavoro in condizioni di connessione di rete assente
Attore	Utente
Precondizioni	Connessione Internet assente e autenticazione al sistema avvenuta con successo

Il terzo e ultimo scenario, descrive la sessione di lavoro dell'utente effettuata con condizioni assenti di connessione Internet. In particolare, l'utente dovrà iniziare ad utilizzare il sistema in uno stato completamente offline della rete ed effettuare le seguenti operazioni:

1. Creazione di un listino
2. Modifica di un listino
3. Eliminazione di un listino

Capitolo 3

Design e progettazione

In questo capitolo si descrivono le fasi di design e progettazione effettuate su Jester. Inoltre, vengono trattati aspetti riguardanti l'architettura del sistema e delle singole parti che lo compongono.

La sezione 3.1 presenta il sistema con i suoi componenti, descrivendo l'interazione e la comunicazione che avviene tra essi.

La sezione 3.2 descrive i pattern progettuali utilizzati e la loro architettura.

La sezione 3.3 affronta la modellazione del comportamento di due componenti del sistema.

La sezione 3.4 tratta lo studio della User Interface dell'applicazione.

3.1 Architettura del sistema e interazione tra i componenti

Trattandosi di un'applicazione web, i due componenti principali sono il client e il server. Il client utilizzerà il framework JavaScript Angular abbinato al database locale IndexedDB.

Lo scopo principale è presentare all'utente un sistema per la gestione dei clienti, prodotti e listini dei prezzi, offrendo un'interfaccia semplice, veloce e soprattutto robusta in condizioni di instabilità della rete. Grazie ad API apposite avverrà la comunicazione con il server.

Il server, invece, è implementato con il framework PHP CakePHP abbinato a MySQL e si occupa del salvataggio dei dati in un database e soprattutto nell'utilizzo di tecniche di sicurezza per l'integrità dei dati mediante token e hashing delle password.

3.2 Design del frontend

Avendo utilizzato Angular come framework di progettazione lato client, l'architettura del frontend dell'applicativo seguirà il design architetturale tipico di un'applicazione Angular.

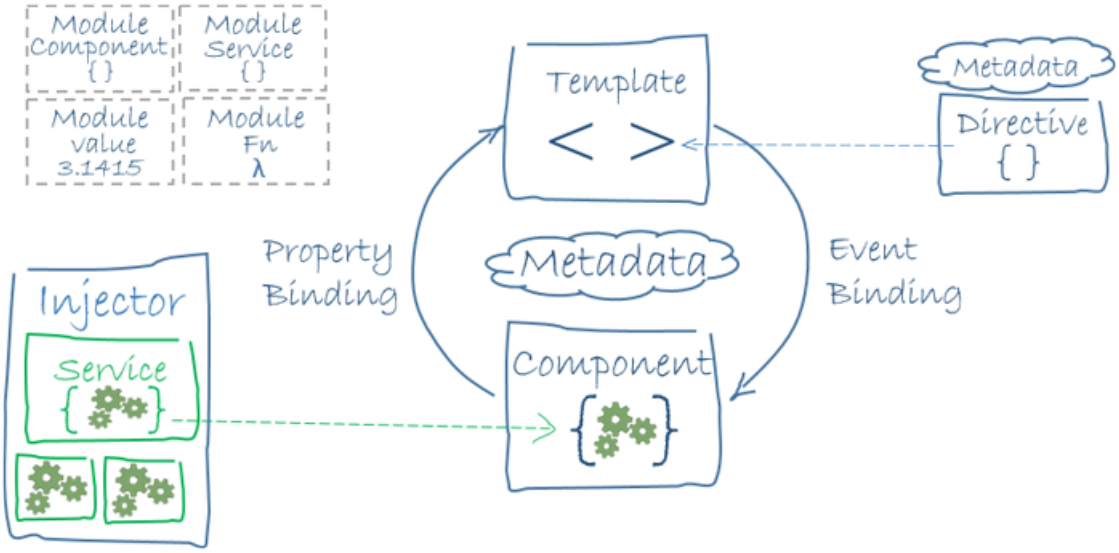


Figura 3.1: Panoramica dell'architettura Angular [9]

Angular, come già descritto nella sezione1.1.1, è caratterizzato da un'architettura molto precisa e particolare:

- **Moduli**: sono i principali blocchi architetturali che costituiscono e formano un'applicazione Angular. Definiscono il contesto di dominio per i componenti;
- **Componenti**: ogni componente definisce una classe che contiene dati e logica dell'applicazione ed è associato ad un template che definisce una interfaccia visiva dell'applicativo. I componenti utilizzano i servizi per funzionalità non direttamente correlate alle viste;
- **Servizi**: sono classi create appositamente per gestire la logica di business, necessaria per il funzionamento dell'applicazione e per la condivisione di informazioni tra i vari componenti;

Ogni pagina dell'applicazione sarà realizzata attraverso uno o più componenti, che si occuperanno di gestire le interazioni dell'utente. Le funzionalità

specifiche, per la gestione della comunicazione con il server e con l'IndexedDB, saranno implementate nei servizi.

Ogni servizio è stato creato per assolvere particolari compiti. Per semplificare la logica inserita nei componenti si è deciso di ricorrere ad un unico servizio che inglobasse tutti i servizi del sistema. A tale scopo è stato utilizzato il **Facade Pattern**, un pattern di progettazione software tipico delle applicazioni Angular, che fornisce un'interfaccia attraverso la quale accedere a sottosistemi che implementano funzionalità complesse, in questo caso i servizi.

Di seguito viene mostrato lo schema architetturale di Jester dal punto di vista dell'interazione tra componenti e servizi.

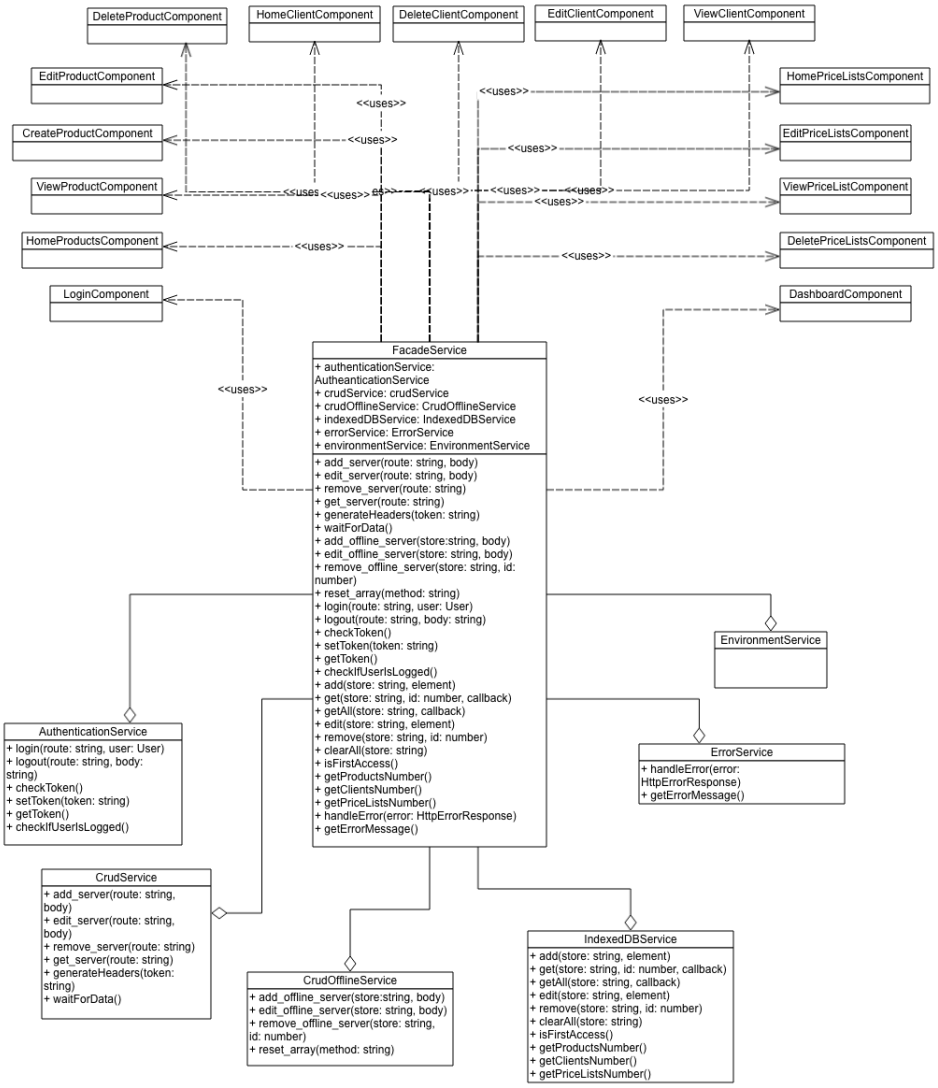


Figura 3.2: Schema architetturale dell'applicazione Jester

3.2.1 Architettura di Jester

Per la progettazione di Jester si è utilizzata una suddivisione come nelle specifiche Angular, creando cartelle diverse, contenenti le logiche architetturali del sistema.

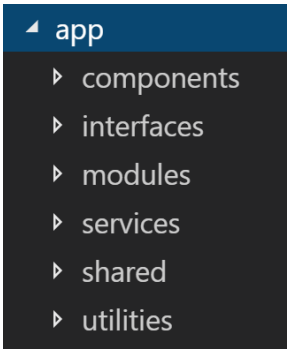
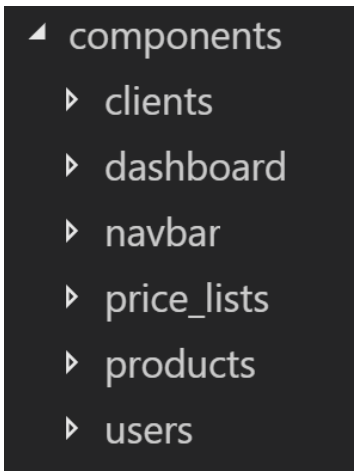


Figura 3.3: Struttura dell'applicazione.

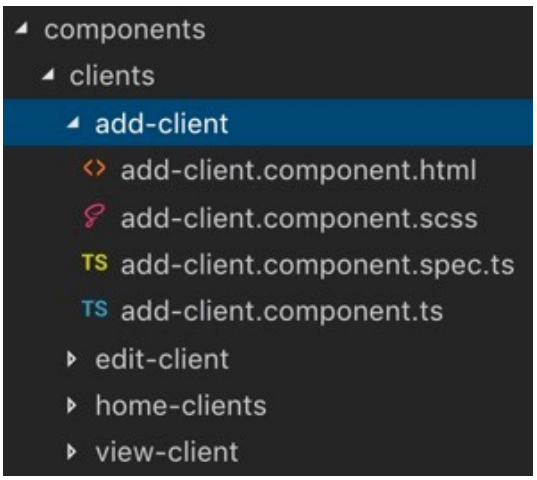
La cartella *app* è formata da sotto-directory che meglio descrivono il loro contenuto all'interno del progetto.

3.2.2 Components

La cartella *components* racchiude al suo interno tutti i componenti del sistema:



(a) Cartella components



(b) Cartella clients/add-client

Figura 3.4: Struttura delle cartelle dei componenti di Jester

I componenti principali sono: clienti, prodotti, listini prezzi e utenti. Oltre a questi è possibile notare anche la presenza del componente dashboard e del componente della barra di navigazione (navbar) che costituiscono comunque un mattone importante nel sistema complessivo.

Per ognuno di essi sono presenti due file TypeScript, uno per la logica e uno per il testing, un file HTML per il template e un file SCSS per lo stile. Ogni directory è composta da sotto-directory con funzionalità specifiche che meglio descrivono quel particolare componente. Tutti i sotto-componenti, a parte la dashboard e la barra di navigazione, contengono la logica legata alle operazioni CRUD:

- **add-component**: adibito alla creazione di un elemento;
- **edit-component**: adibito alla modifica di un elemento;
- **home-component**: adibito alla visione di tutti gli elementi di una data collezione;
- **view-component**: adibito alla visione di un singolo elemento;

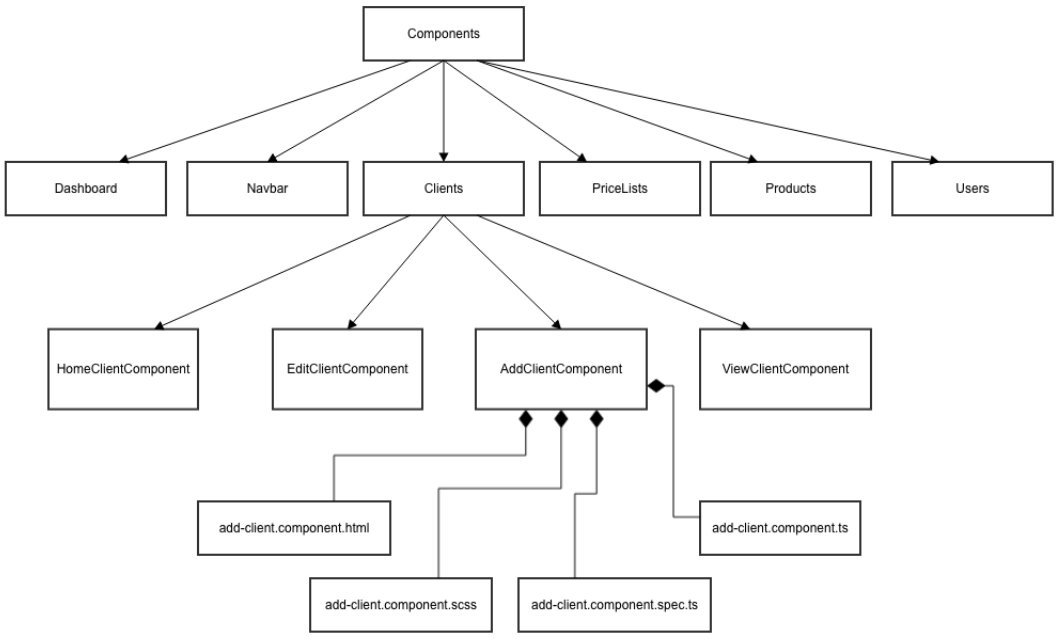


Figura 3.5: Schema architetturale dei componenti di Jester

3.2.3 Services

La cartella *services* contiene tutti i servizi di cui necessita il sistema:

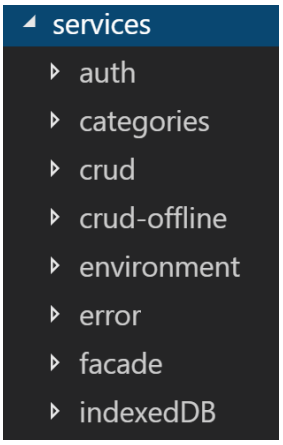


Figura 3.6: Cartella services

All'interno è presente l'intera logica dell'applicazione legata alla comunicazione con il server e alla condivisione di informazioni tra i componenti.
In particolare sono stati progettati i seguenti servizi:

- **authService**: gestione dell'autenticazione degli utenti;
- **crudService**: gestione della comunicazione con il server tramite REST API;
- **crudOfflineService**: gestione della logica applicativa in modalità offline;
- **environmentService**: gestione dell'URL a seconda della tipologia dell'esecuzione del software, può essere in modalità di produzione o di sviluppo;
- **errorService**: gestione degli errori lato client e lato server;
- **facadeService**: servizio che offre un'interfaccia di tutti i servizi. Rappresenta il design pattern Facade;
- **indexedDBService**: gestione del salvataggio dei dati nel database locale IndexedDB;

3.2.4 Modules

Nella cartella *modules* ci sono i moduli di Angular, fondamentali per l'esecuzione dell'intera applicazione.

Come è possibile notare dalla seguente immagine è stato creato un modulo che gestisce interamente le importazioni dei componenti di Angular Material:

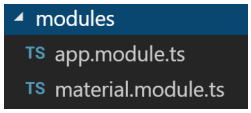


Figura 3.7: Cartella modules

3.2.5 Shared

Nella cartella *shared* sono presenti i file principali condivisi da tutta l'applicazione, ovvero l'AppComponent, che rappresenta il root component da cui discendono tutti i componenti creati, e la gestione del routing, mediante il quale l'utente può navigare tra le diverse pagine e quindi tra i diversi componenti del sistema:

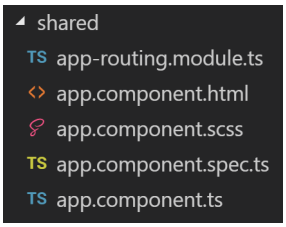


Figura 3.8: Cartella shared

3.2.6 Utilities

Infine è stata creata una cartella *utilities* nella quale sono presenti tutte le classi di utilità del sistema, dalla gestione dei modal, delle pagine di errore, fino al controllo del routing interno:

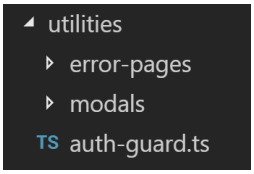


Figura 3.9: Cartella utilities

3.3 Comportamento del sistema

Per modellare il comportamento del sistema si è deciso di utilizzare una tipologia di diagramma UML[1], il diagramma delle attività. Questo diagramma modella il flusso di lavoro di un determinato caso d'uso e rappresenta il flusso delle operazioni, evidenziando le relazioni tra gli attori e i punti di decisione che caratterizzano il caso d'uso.

Di seguito sono mostrati due diagrammi delle attività legati alla fase di autenticazione del sistema e alla fase di creazione di una nuova entità.

3.3.1 Fase di autenticazione

Per quanto concerne la fase di autenticazione, il sistema verificherà l'esistenza di un utente che ha appena inserito l'indirizzo email e la password nell'appropriato form della pagina di login. Nel caso in cui le credenziali siano corrette, il sistema reindirizza l'utente nella pagina principale dell'applicativo, la dashboard, altrimenti viene avvisato tramite un messaggio di errore.

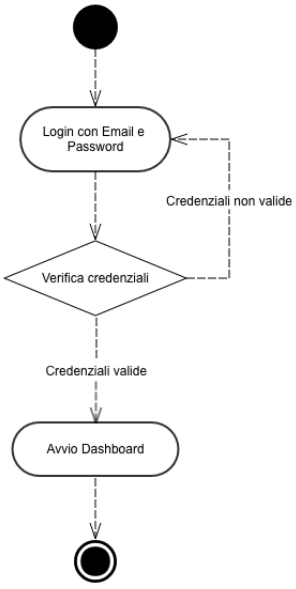


Figura 3.10: Diagramma delle attività per la fase di autenticazione

[1] **U**nified **M**odeling **L**anguage: linguaggio di modellazione basato sul paradigma orientato agli oggetti.

3.3.2 Fase di creazione

Una volta autenticato nel sistema, l'utente può effettuare diverse operazioni, tra cui la creazione, la modifica e l'eliminazione di un prodotto, cliente o listino. Di seguito si prende in esame la creazione di un nuovo prodotto.

L'utente, per poter creare un nuovo prodotto correttamente, deve compilare alcuni campi obbligatori. È mostrata in figura la modellazione del comportamento della creazione di un nuovo prodotto.

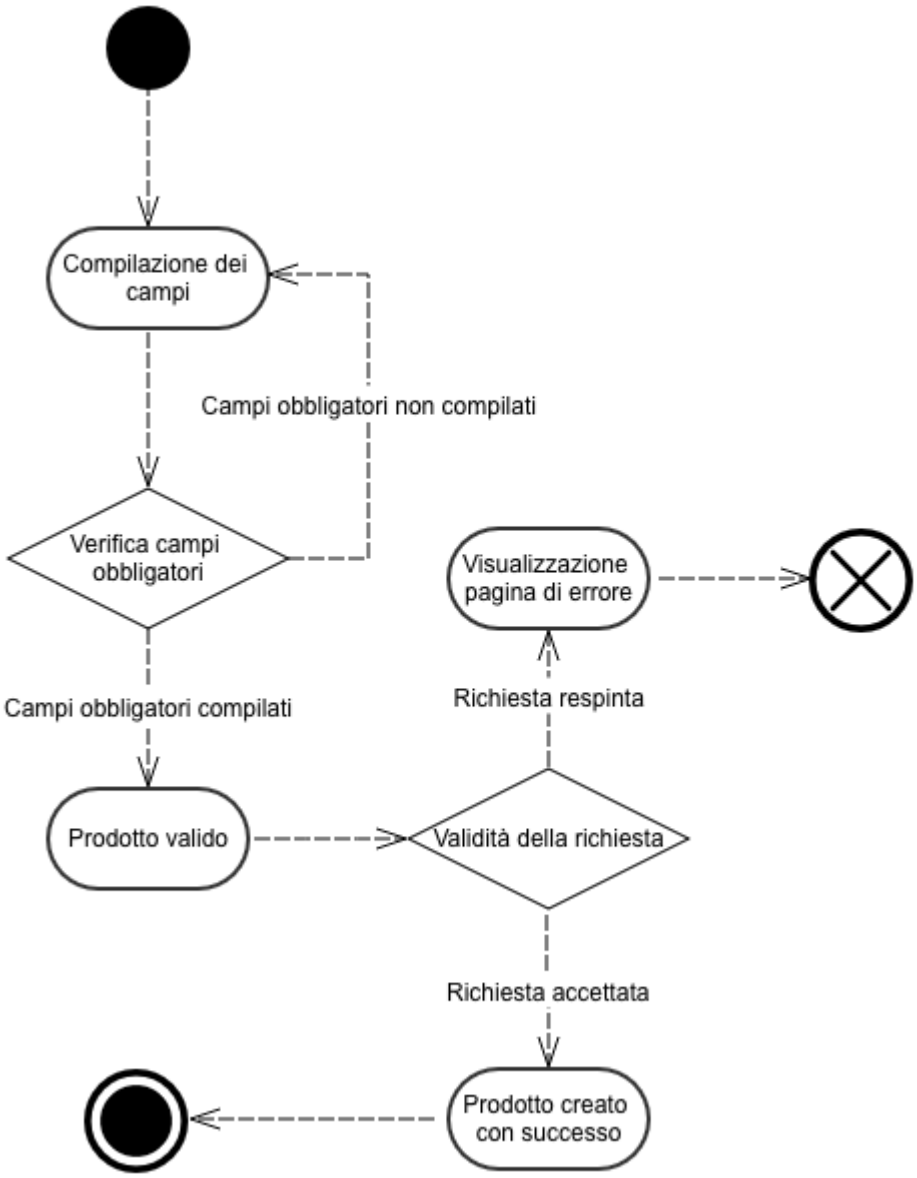


Figura 3.11: Diagramma delle attività per la fase di creazione di un prodotto

3.4 User Interface

L'interfaccia utente è stata progettata con l'uso di mockup[2], grazie ai quali si sono potuti sviluppare i diversi template mediante file HTML e SCSS. Si è prestata particolare attenzione alla progettazione di un'interfaccia semplice da utilizzare, per venire incontro agli utenti meno esperti.

3.4.1 Login

Come prima pagina iniziale l'applicativo sarà caratterizzato da una schermata di login, all'interno della quale gli utenti possono autenticarsi al sistema ed accedere alla propria dashboard.

Siccome vengono trattati dati sensibili, la sicurezza risulta fondamentale, per questo motivo la password è crittografata lato server da opportune tecniche di hashing. Invece, per quanto concerne la comunicazione client-server, si assume che il trasferimento dei dati avvenga mediante un protocollo sicuro HTTPS che assicuri l'integrità dei dati e la loro protezione contro violazioni esterne.

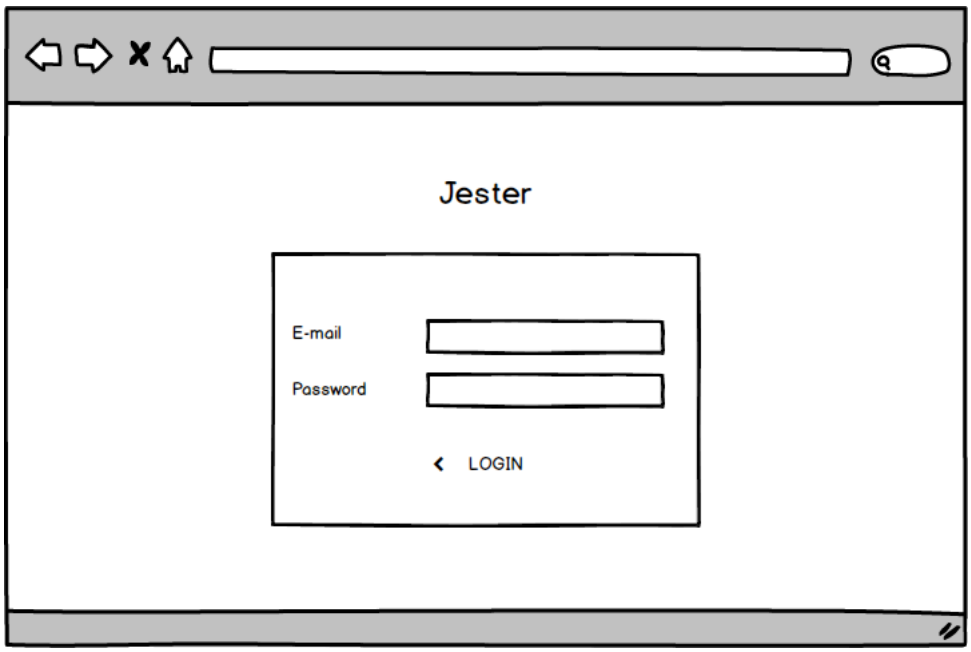


Figura 3.12: Mockup della schermata iniziale di Jester: la pagina di Login

[2]**Mockup**: riproduzione di un oggetto o modello in scala ridotta. Attività utile in fase di progettazione.

3.4.2 Dashboard

La pagina "Dashboard" può essere considerata come l'home page dell'applicativo. In essa l'utente potrà svolgere diverse operazioni:

- Visualizzazione del numero di clienti, prodotti e listini salvati nel database;
- Utilizzo di scorciatoie per l'aggiunta di nuovi elementi;

Un elemento fondamentale risulta essere la barra di navigazione, presente in ogni pagina, tramite la quale sarà possibile navigare e scegliere le operazioni da effettuare.

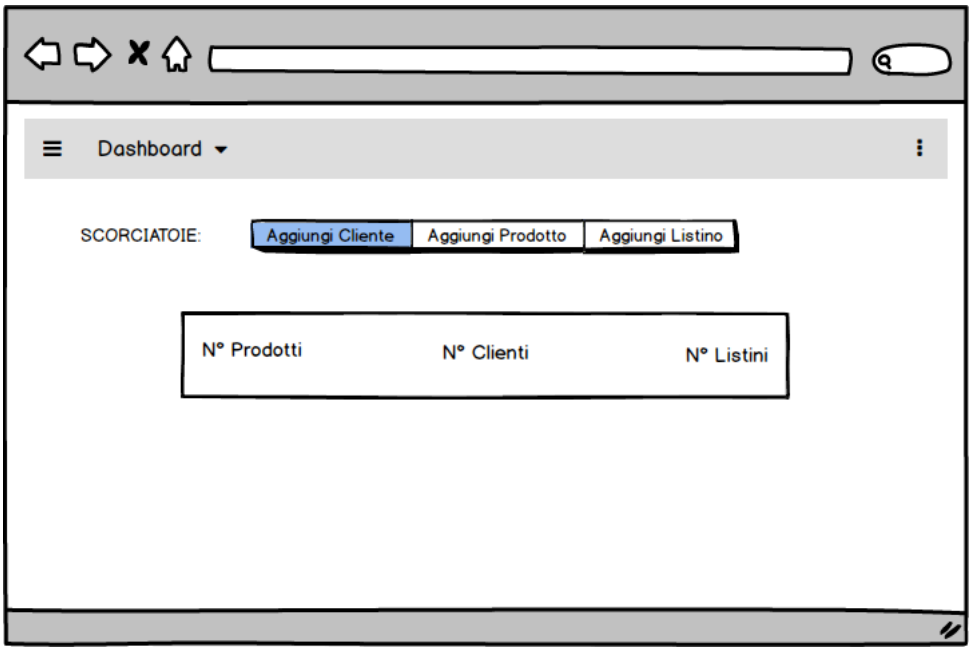


Figura 3.13: Mockup della home page di Jester: la pagina di Dashboard

3.4.3 HomeComponent

"HomeComponent" rappresenta la pagina di visualizzazione di tutti gli elementi di una data collezione, per cui clienti, prodotti e listini.

Nel mockup seguente è stato realizzato un prototipo della pagina inerente ai prodotti. L'utente potrà visualizzare tutti i prodotti salvati nel database e interagirvi mediante tre diverse operazioni presenti nella colonna della tabella "Azioni". In particolare, potrà visualizzare un singolo prodotto, con tutte le sue caratteristiche, modificarlo o eliminarlo. Altre funzionalità disponibili sono quelle di ricercare un prodotto digitando qualsiasi sua caratteristica, non solo il nome o il codice, e poter ordinare i prodotti tramite tecniche di filtraggio, come ad esempio l'ordine alfabetico.

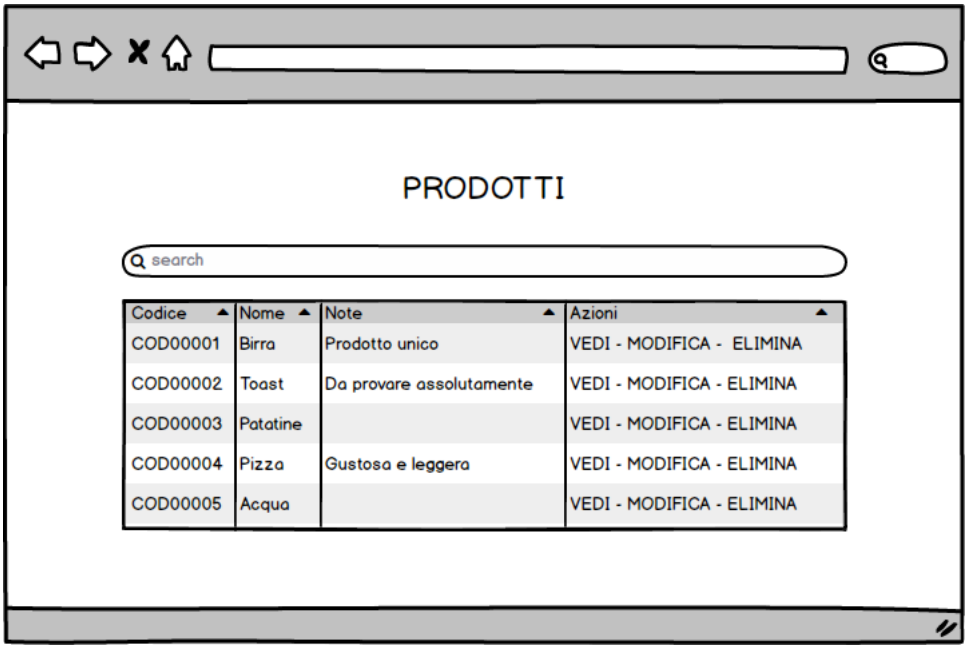


Figura 3.14: Mockup della pagina di visualizzazione dei prodotti

3.4.4 ViewComponent

In questa pagina è possibile visualizzare un singolo elemento che, nel caso seguente, è un prodotto.

L'utente potrà leggere tutte le caratteristiche del prodotto e, se lo desidera, potrà effettuare delle modifiche grazie al "button" "Modifica Prodotto" che lo reindirizzerà alla pagina di modifica.

Figura 3.15: Mockup della pagina di visualizzazione di un singolo prodotto

3.4.5 CreateComponent

In questa schermata l'utente potrà creare e aggiungere un nuovo elemento nel database.

Nella figura seguente viene mostrato il caso relativo all'aggiunta di un prodotto. La creazione avviene mediante completamento dei campi di input suddivisi in determinate sezioni.

Figura 3.16: Mockup della pagina di creazione di un prodotto

Capitolo 4

Sviluppo e implementazione

In questo capitolo vengono messe in evidenza le scelte strategiche per lo sviluppo delle funzionalità del sistema.

La sezione 4.1 descrive l'implementazione dei componenti Angular di Jester.

La sezione 4.2 analizza nel dettaglio l'implementazione dei servizi del sistema.

4.1 Sviluppo dei Componenti di Jester

In questa sezione viene presentata l'implementazione di due componenti principali di Jester, *HomeProductsComponent* e *AppComponent*.

4.1.1 HomeProductsComponent

La pagina di visualizzazione rappresenta un punto fondamentale di Jester, perché interagendo con essa è possibile effettuare qualsiasi operazione su un certo insieme di elementi. Nello specifico si presenta il componente "HomeProductsComponent", destinato al rendering di tutti i prodotti.

home-products.component.ts

Ogni componente Angular è rappresentato da un file TypeScript, che contiene la logica, ed è collegato tramite data binding ad un file HTML per il rendering dei dati. Mediante il decoratore *@Component* si specifica che la classe sarà un componente del sistema. Nel costruttore è possibile notare la presenza del servizio *FacadeService*, unico servizio del sistema "iniettato" dentro i componenti. Grazie ad esso "HomeProductsComponent" può comunicare con qualsiasi servizio per gestire la sua logica applicativa.

```
/** Componente che si occupa della visualizzazione di tutti i
    prodotti. */
@Component({
  selector: 'app-home-products',
  templateUrl: './home-products.component.html',
  styleUrls: ['./home-products.component.scss']
})
export class HomeProductsComponent implements OnInit {
  private title: String = 'Prodotti';
  private products: Product[];
  private store: string = "products";
  private displayedColumns: String[] = ['code', 'name',
      'description', 'actions'];
  private dataSource = new MatTableDataSource<Product>();
  @ViewChild(MatPaginator) paginator: MatPaginator;
  @ViewChild(MatSort) sort: MatSort;

  constructor(
    private titleService: Title,
    public dialog: MatDialog,
    private facadeService: FacadeService
  ) {}
```

Listato 4.1: home-products.component.ts

"HomeProductsComponent" ha il compito di scaricare tutti i prodotti salvati nell'IndexedDB e popolare la tabella presente all'interno della pagina. Ciò è reso possibile nel modo seguente:

```
/** Metodo che scarica tutti i prodotti dal database locale. */
  private getProductsFromCollection() {
    this.facadeService.getAll(this.store, products => {
      this.products = products as Product[];
      this.dataSource = new MatTableDataSource(this.products);
      this.dataSource.paginator = this.paginator;
      this.dataSource.sort = this.sort;
    });
  }
```

Listato 4.2: Metodo *"getProductsFromCollection()"*

"getProductsFromCollection()" gestisce la Promise[1] ritornata dal metodo "getAll(...)" dell'IndexedDB e crea una nuova tabella di Angular Material popolandola con i tutti i prodotti recuperati. Inoltre, la tabella sarà configurata con tecniche di paginazione e filtraggio dei dati:

```
/**
 * Metodo che applica la tipologia di filtraggio all'interno della
    tabella.
 * @param filterValue La stringa da cercare all'interno della
    tabella.
 */
public applyFilter(filterValue: string): void {
    filterValue = filterValue.trim();
    filterValue = filterValue.toLowerCase();
    this.dataSource.filter = filterValue;
}
```

Listato 4.3: Metodo *"applyFilter(...)"*

home-products.component.html

File HTML di HomeProducts che rappresenta il suo template. In esso è possibile vedere il concetto di data binding spiegato nella sezione 1.1.1.

Come detto in precedenza, nella pagina è situata una barra di ricerca con la quale è possibile cercare dinamicamente un dato elemento all'interno della tabella. Questo è possibile con il seguente frammento:

```
<div class="div-filter flex-container">
  <mat-form-field>
    <input matInput (keyup)="applyFilter($event.target.value)">
  </mat-form-field>
</div>
```

Listato 4.4: home-products.component.html

Di seguito è mostrato il codice relativo alla creazione della tabella, con le direttive e le sintassi Angular:

```
<div class="table-container mat-elevation-z8">
  <table mat-table [dataSource]="getDataSource()" matSort >
    <ng-container matColumnDef="code">
```

[1]**Promise**: oggetto usato per computazione asincrona. Una Promise rappresenta un'operazione che non è ancora completata, ma lo sarà in futuro.

```
      <th mat-header-cell *matHeaderCellDef mat-sort-header> Codice
          </th>
      <td mat-cell *matCellDef="let product"> {{product?.code}} </td>
    </ng-container>
    <ng-container matColumnDef="name">
      <th mat-header-cell *matHeaderCellDef mat-sort-header>
          Articolo </th>
      <td mat-cell *matCellDef="let product"> {{product?.name}} </td>
    </ng-container>
    <ng-container matColumnDef="description">
      <th mat-header-cell *matHeaderCellDef mat-sort-header>
          Descrizione </th>
      <td mat-cell *matCellDef="let product">
          {{product?.description}} </td>
    </ng-container>
    <ng-container matColumnDef="actions">
      <th mat-header-cell class="actions" *matHeaderCellDef
          mat-sort-header> Azioni </th>
      <td mat-cell class="actions" *matCellDef="let product">
        <a matTooltip="Vedi" matTooltipPosition="above" mat-mini-fab
            color="primary"
            routerLink="/products/view/{{product.id}}">
          <i class="material-icons">remove_red_eye</i>
        </a>
        <a matTooltip="Modifica" matTooltipPosition="above"
            mat-mini-fab color="accent"
            routerLink="/products/edit/{{product.id}}">
          <i class="material-icons">create</i>
        </a>
        <a matTooltip="Elimina" (click)="openDialog(product.id)"
            matTooltipPosition="above" mat-mini-fab color="warn">
          <i class="material-icons">delete_forever</i>
        </a>
      </td>
    </ng-container>
    <tr mat-header-row *matHeaderRowDef="getDisplayedColumns()"></tr>
    <tr mat-row *matRowDef="let product; columns:
        getDisplayedColumns();"></tr>
  </table>
  <mat-paginator [pageSizeOptions]="[5, 10, 25, 100]"
      [showFirstLastButtons]="true"></mat-paginator>
</div>
```

Listato 4.5: home-products.component.html

Viene inoltre impostata una paginazione che consente all'utente di selezionare un numero preciso di elementi per ogni pagina della tabella.

4.1.2 AppComponent: la radice di tutto

"AppComponent" rappresenta il componente padre: lo si può comparare alla radice di un albero, i cui rami sono gli altri componenti del sistema. Esso permette di visualizzare a monitor il componente giusto a seconda del routing corrente. Inoltre, è la classe che, comunicando con i servizi implementati, gestisce le operazioni effettuate offline e le invia al server non appena ritorna disponibile la connessione Internet.

Siccome è il componente principale, si suddivide la spiegazione della sua logica applicativa in sezioni.

Inizializzazione e popolamento dell'IndexedDB

L'inizializzazione risulta un punto importante del ciclo di vita di ogni componente, soprattutto per l'AppComponent.

Appena viene inizializzato, vengono effettuate diverse operazioni:

1. Viene scaricato il token per la corretta comunicazione con il server con il metodo "checkToken()";
2. Svuota tutte le collezioni dell'IndexedDB ripopolandole con i dati presi direttamente dal server. In questo modo si ha sempre la sicurezza di lavorare con i dati aggiornati;

```
/** Metodo che viene eseguito solo quando il componente viene
   inizializzato. */
 ngOnInit() {
   this.checkToken();
   if (this.facadeService.isFirstAccess()) {
     this.clearCollections();
     this.getDataFromServer();
   }
   this.facadeService.generateHeaders(this.token);
   setInterval(()=> {
       this.checkNetwork();
       this.checkRequestToSend();
   }, 2000);
 }
```

Listato 4.6: Inizializzazione di AppComponent

Controllo della connessione

Periodicamente viene controllato lo stato della rete per avvisare l'utente del passaggio in modalità offline.

```
/** Metodo che controlla se la connessione e' presente. */
  private checkNetwork(): void {
    var self = this;
    if (navigator.onLine && self.isOnline == false) {
      self.isOnline = true;
      this.createSnackBar("Connessione internet presente. Buon
          lavoro da Jestergest!");
    } else if (!navigator.onLine == self.isOnline == true) {
      self.isOnline = false;
      this.createSnackBar("Connessione internet assente. Stai
          lavorando in modalita' offline!");
    }
  }
```

Listato 4.7: Metodo *"checkNetwork()"*

Gestione dell'Offline

Il metodo più importante di "AppComponent è "checkRequestToSend()". Periodicamente si controlla se sono presenti delle richieste effettuate in modalità offline. Non appena si ritorna online, le richieste vengono inviate al server per la sincronizzazione della sessione di lavoro.

Uno dei principali problemi riscontrati è stato progettare come inviare le richieste al server in modo cronologico. Di seguito riporto un esempio:
Si presuppone che, durante la sessione offline, l'utente abbia creato un nuovo prodotto e subito dopo lo abbia rimosso. Al ritorno della connessione Internet le richieste devono essere inviate al server in ordine temporale, ovvero prima la richiesta POST della creazione del nuovo prodotto e successivamente la richiesta DELETE di quest'ultimo. Per ovviare al suddetto problema viene prima controllata la presenza di richieste POST, poi di quelle PUT e infine di quelle DELETE. Di seguito riporto un frammento di codice del metodo in questione "checkRequestToSend()".

```
// Se e' presente una richiesta delete, controllo prima se c'e'
    una richiesta post e poi una richiesta put
if (localStorage.getItem("delete_products") != null) {
  if (localStorage.getItem("post_products") != null)
      this.sendOfflineRequest(this.productsStore,
      this.url_addProduct, this.post);
  if (localStorage.getItem("put_products") != null)
      this.sendOfflineRequest(this.productsStore,this.url_editProduct,
      this.put);
  this.sendOfflineRequest(this.productsStore,
      this.url_deleteProduct, this.delete);
}
```

Listato 4.8: Metodo *"checkRequestToSend()"*

Le richieste sono inviate mediante il seguente metodo (per semplicità è mostrata solo la gestione delle richieste POST):

```
/** Metodo che sincronizza le operazioni offline con il server. */
  private sendOfflineRequest(store: string, url: string, method:
      string) {
    let request;
    let separator: string = "_";
    request = JSON.parse(localStorage.getItem(method + separator +
        store));
    localStorage.removeItem(method + separator + store);
    request.forEach(body => {
      this.facadeService.generateHeaders(this.token);
      this.createSnackBar("E' stata inviata una richiesta in
          background al server!");
      if (method == this.post) {
        this.facadeService.add_server(url, body).subscribe(
          success => this.createSnackBar("La creazione del " +
              this.returnItemNotification(store) + " che hai
              effettuato offline e' avvenuta con successo!"),
          error => console.log(error)
        );
      }
    });
    this.facadeService.reset_array(method);
  }
```

Listato 4.9: Metodo *"sendOfflineRequest(...)"*

4.2 Sviluppo dei Servizi di Jester

In questa sezione verrà presentata l'implementazione dei servizi principali di Jester.

4.2.1 IndexedDBService: salvataggio locale dei dati

Per usufruirne è necessario installare il servizio che si comporta da wrapper del database locale IndexedDB con il seguente comando:

```
$ npm install angular2-indexeddb
```

Il primo passo consiste nell'importare la classe "angular2-indexeddb" nel servizio come una dipendenza:

```
import { AngularIndexedDB } from 'angular2-indexeddb';
```

Listato 4.10: indexedDB.service.ts

Con la creazione di una nuova istanza della classe è possibile iniziare la comunicazione con l'IndexedDB:

```
private db: AngularIndexedDB = new AngularIndexedDB('myDb', 1);
```

Listato 4.11: indexedDB.service.ts

Come primo step bisogna creare le collezioni con le quali si desidera interagire:

```
/** Costruttore del servizio. */
  constructor() {
    this.dbCreatePromise = this.db.openDatabase(1, e => {
      var productsStore = e.currentTarget.result.createObjectStore(
        'products', { keyPath: this.key, autoIncrement: true });
      var clientsStore = e.currentTarget.result.createObjectStore(
        'clients', { keyPath: this.key, autoIncrement: true });
      var priceListsStore = e.currentTarget.result.createObjectStore(
        'price_lists', { keyPath: this.key, autoIncrement: true });
    });
  }
```

Listato 4.12: indexedDB.service.ts

Nello specifico sono state create tre collezioni, rispettivamente per prodotti, clienti e listini.

Con la figura seguente ci si addentra nel cuore del servizio. Le operazioni consentite per comunicare con l'IndexedDB sono le operazioni CRUD. Ognuna

di esse ritorna una Promise che deve essere gestita dall'utilizzatore del servizio. Nella figura seguente si evidenziano solo i metodi di lettura GET e di modifica PUT.

```
/**
 * Metodo che prende dal database locale un elemento con un
    determinato ID.
 * @param store Nome della collezione dove verra' preso l'elemento.
 * @param id ID dell'elemento da prendere dalla collezione.
 * @param callback Dati dell'elemento richiesto.
 */
public get(store: string, id: number, callback): void {
  this.dbCreatePromise.then(
    () => {
      this.db.getByKey(store, id).then(
        element => callback(element),
        error => console.log(error)
      );
    }
  );
}

/**
 * Metodo che aggiorna nel database locale l'elemento modificato.
 * @param store Nome della collezione dove verra' modificato
    l'elemento.
 * @param element Dati dell'elemento modificato.
 */
public edit(store: string, element): void {
  this.db.update(store, element).then(
    success => {},
    error => console.log(error)
  );
}
```

Listato 4.13: Metodo *"get(...)"* e *"edit(...)"*

Con questo servizio risulta semplice comunicare con il database locale, nel quale è possibile salvare qualsiasi elemento che si modifica. Lo scopo è di non dover comunicare in modo sincrono con il server, ma lavorare localmente e aggiornare il server in background.

4.2.2 CrudService: servizio REST

Obiettivo del servizio è la gestione della comunicazione con il server. Per questo motivo CrudService manipolerà le operazioni effettuate dall'utente inviando richieste CRUD al backend. Si riporta solo il metodo che gestisce le richieste POST.

```
/** Costruttore del servizio. */
  constructor(
    private http: HttpClient,
    private env: EnvironmentService
  ) {}

  /**
   * Metodo che invia al server l'elemento creato.
   * @param route URL del percorso per la comunicazione col server.
   * @param body Contiene i dati dell'elemento creato.
   */
  public add(route: string, body): Observable<any> {
    return
        this.http.post(this.createCompleteRoute(this.env.getUrl(),
        route), body, this.generateHeaders(this.token));
  }

  /** Metodo per creare il percorso completo per la comunicazione
      col server. */
  private createCompleteRoute(envAddress: string, route: string):
      string {
    return `${envAddress}/${route}`;
  }
```

Listato 4.14: crud.service.ts

Il metodo privato "createCompleteRoute(envAddress, route)" è una funzione di utility, che ha lo scopo di creare il percorso corretto a seconda della modalità con la quale è stata eseguita l'applicazione, ovvero produzione o sviluppo.

Tutte le richieste ritornano un oggetto Observable[2] che deve essere gestito dall'utilizzatore del servizio con la clausola *"subscribe"*, come nel seguente esempio:

```
/** Metodo che invia al server il prodotto creato. */
  private addProductToServer(): void {
    let url_add: string =
        "angular-jestergest-new/products/add.json?type_id=1";
    this.facadeService.generateHeaders(this.facadeService.getToken());
    this.facadeService.add_server(url_add, this.productForm.value)
      .subscribe(
        success => {
          this.addProductToCollection("Prodotto creato con
              successo!");
        },
        error => {
          this.loading = false;
          this.facadeService.handleError(error);
          this.errorMessage = this.facadeService.getErrorMessage();
        }
      );
  }
```

Listato 4.15: Subscription in add-product.component.ts

4.2.3 CrudOfflineService: gestione dell'offline

Servizio che gestisce le operazioni effettuate dall'utente in condizioni offline. L'obiettivo consiste nel salvataggio di array di richieste, differenziandole per la diversa tipologia CRUD, nel localStorage, da processare non appena la connessione ritorna disponibile.

```
export class CrudOfflineService {
  private post = [];
  private put = [];
  private delete = [];

  /** Costruttore del servizio. */
  constructor() {}

  /**
```

[2]**Observable**: oggetto usato per la computazione asincrona. Indicato per gestire dati inviati dal server in seguito ad una richiesta del client.

```
 * Salva nel LocalStorage le richieste POST.
 * @param store Tipologia di elemento da salvare.
 * @param body Dati dell'elemento creato.
 */
public add_offline(store: string, body): void {
  this.post.push(JSON.stringify(body));
  localStorage.setItem("post_"+ store, JSON.stringify(this.post));
  console.log("POST " + store + " OFFLINE: ",this.post);
}

/**
 * Salva nel LocalStorage le richieste PUT.
 * @param store Tipologia di elemento da salvare.
 * @param body Dati dell'elemento modificato.
 */
public edit_offline(store: string, body): void {
  this.put.push(JSON.stringify(body));
  localStorage.setItem("put_"+ store, JSON.stringify(this.put));
  console.log("PUT " + store + " OFFLINE: ",this.put);
}

/**
 * Salva nel LocalStorage le richieste DELETE.
 * @param store Tipologia di elemento da salvare.
 * @param id ID dell'elemento da eliminare.
 */
public remove_offline(store: string, id: number): void {
  this.delete.push(JSON.stringify(id));
  localStorage.setItem("delete_"+ store,
      JSON.stringify(this.delete));
  console.log("DELETE " + store + " OFFLINE: ",this.delete);
}
```

Listato 4.16: crud-offline.service.ts

4.2.4 AuthService: gestione degli utenti

Servizio che gestisce la sessione degli utenti, con login e logout.

```
/**
 * Metodo che gestisce il login.
 * @param route Percorso della richiesta.
 * @param user Dati dell'utente.
 */
public login(route: string, user: User): Observable<any> {
  this.http_options = {
    headers: new HttpHeaders({ 'Content-Type': 'application/json',
        'X-CSRF-Token': sessionStorage.getItem("token") }),
    withCredentials: true
  };
  return
      this.http.post<any>(this.createCompleteRoute(this.env.getUrl(),
      route), user, this.http_options).pipe(map(res => {
    if (res.data != null) {
      sessionStorage.setItem('currentUser',
          JSON.stringify(user.email));
      return user.email;
    }
    this.isUserLogged = false;
  }));
}

/**
 * Metodo che effettua il logout dell'utente.
 * @param route Percorso della richiesta.
 * @param user Dati dell'utente.
 */
public logout(route: string, body): Observable<User> {
  return
      this.http.post<User>(this.createCompleteRoute(this.env.getUrl(),
      route), body, this.http_options);
}
```

Listato 4.17: auth.service.ts

Altro obiettivo del servizio è reperire il token necessario per comunicare con il server, diventando disponibile per ogni servizio.

```
/** Metodo che scarica il csrf-token prodotto dal server per la
    corretta autenticazione. */
 public checkToken(): Observable <any> {
   return this.http.get<any>(this.url_token, { withCredentials:
       true });
 }

 /** Metodo che setta il token per la comunicazione col server. */
 public setToken(token: string) {
   sessionStorage.setItem("token", token);
   this.token = sessionStorage.getItem("token");
 }
```

Listato 4.18: auth.service.ts

4.2.5 FacadeService: servizio di servizi

Come spiegato nella sezione 3.2, FacadeService offre un'interfaccia tramite la quale gestire tutti i servizi del sistema. Rappresentando il pattern Facade, il primo passo è importare tutti i servizi di Jester:

```
import { AuthService } from '../auth/auth.service';
import { CrudService } from '../crud/crud.service';
import { CrudOfflineService } from
   '../crud-offline/crud-offline.service';
import { ErrorHandlerService } from "../error/error-handler.service";
import { IndexedDBService } from "../indexedDB/indexedDB.service";
```

Listato 4.19: facade.service.ts

Successivamente si inietta ogni servizio dentro il costruttore, seguendo il design pattern Singleton[3] offerto da Angular. In questo modo risulta possibile usufruire di ogni metodo pubblico dei servizi iniettati.

```
/** Costruttore del servizio. */
constructor(
  private crudService: CrudService,
  private crudOfflineService: CrudOfflineService,
  private iDBService: IndexedDBService,
  private authService: AuthService,
  private errorHandlerService: ErrorHandlerService
) {}

/* Metodi di CrudOfflineService */
  public add_offline_server(store:string, body) {
    return this.crudOfflineService.add_offline(store, body);
  }

  public edit_offline_server(store: string, body) {
    return this.crudOfflineService.edit_offline(store, body);
  }

  public remove_offline_server(store: string, id: number) {
    return this.crudOfflineService.remove_offline(store, id);
  }

  public reset_array(method: string) {
    return this.crudOfflineService.reset_array(method);
  }
```

Listato 4.20: facade.service.ts

[3]**Singleton**: design pattern creazionale, con lo scopo di garantire che di una determinata classe venga creata una e una sola istanza.

4.2.6 AuthGuard: la guardia di Jester

L'esigenza di proteggere le diverse view dell'applicazione ha permesso l'utilizzo di una "guardia"[4] per tenere lontano gli utenti non autenticati. Il *guard Angular* viene gestito tramite l'interfaccia dedicata *"CanActivate"*, che consente di verificare i privilegi dell'utente e quindi negare o consentire l'accesso ad un determinato percorso. Di seguito è mostrata l'implementazione di "AuthGuard":

```
export class AuthGuard implements CanActivate {

  /** Costruttore del componente. */
  constructor(private router: Router) {}

  /**
   * Metodo che non consente la navigazione agli utenti non loggati.
   * @param route
   * @param state
   */
  public canActivate(route: ActivatedRouteSnapshot, state:
      RouterStateSnapshot): boolean {
    if (sessionStorage.getItem('currentUser')) {
      return true;
    }
    this.router.navigate(['/login'], { queryParams: { returnUrl:
        state.url }});
    return false;
  }

}
```

Listato 4.21: auth-guard.ts

[4]**Guardia**: servizio con lo scopo di proteggere i percorsi dell'applicazione da utenti non autenticati.

4.2.7 AppRoutingModule: modulo di navigazione

Una best practice di Angular consiste nel creare un modulo dedicato al routing interno del sistema, che per convenzione si chiama *AppRoutingModule*.

```
const routes: Routes = [
  { path: '', component: DashboardComponent, canActivate:
      [AuthGuard] },
  { path: 'login', component: LoginUsersComponent },
  { path: 'clients', component: HomeClientsComponent, canActivate:
      [AuthGuard] },
  { path: 'clients/view/:id', component: ViewClientComponent,
      canActivate: [AuthGuard] },
  { path: 'clients/add', component: AddClientComponent, canActivate:
      [AuthGuard] },
  { path: 'clients/edit/:id', component: EditClientComponent,
      canActivate: [AuthGuard] },
  { path: 'products', component: HomeProductsComponent, canActivate:
      [AuthGuard] },
  { path: 'products/view/:id', component: ViewProductComponent,
      canActivate: [AuthGuard] },
  { path: 'products/add', component: AddProductComponent,
      canActivate: [AuthGuard] },
  { path: 'products/edit/:id', component: EditProductComponent,
      canActivate: [AuthGuard] },
  { path: 'price-lists', component: PriceListsComponent,
      canActivate: [AuthGuard] },
  { path: 'price-lists/add', component: AddPriceListComponent,
      canActivate: [AuthGuard] },
  { path: 'price-lists/view/:id', component: ViewPriceListComponent,
      canActivate: [AuthGuard] },
  { path: 'price-lists/edit/:id', component: EditPriceListComponent,
      canActivate: [AuthGuard] },
  { path: '404', component: NotFoundComponent, canActivate:
      [AuthGuard] },
  { path: '**', redirectTo: '/404', pathMatch: 'full' }
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```

Listato 4.22: app-routing.module.ts

4.3 Realizzazione dei mockup

Vengono di seguito presentate le interfacce grafiche al termine del loro sviluppo. La loro realizzazione si è basata fortemente sui mockup progettati nella sezione 3.4.

Login

Figura 4.1: Interfaccia Login

Dashboard

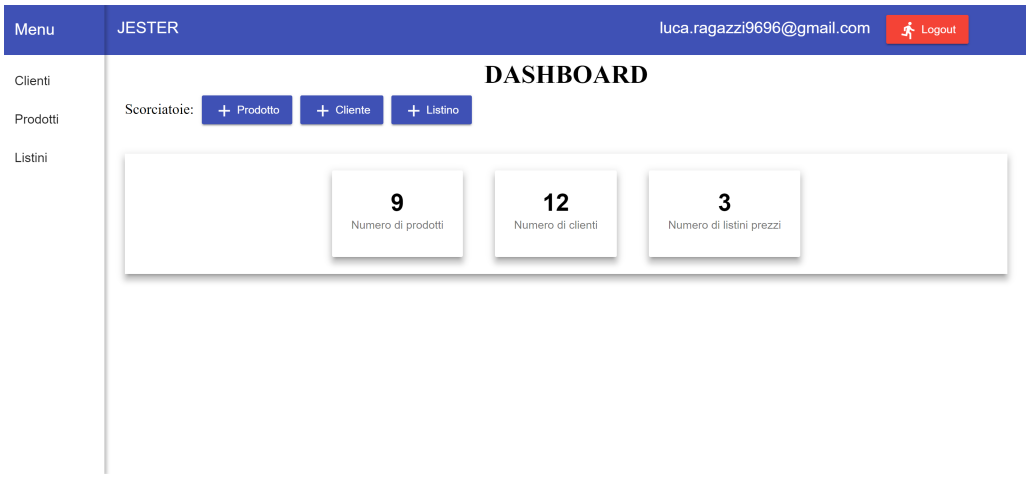


Figura 4.2: Interfaccia Dashboard

HomeClients

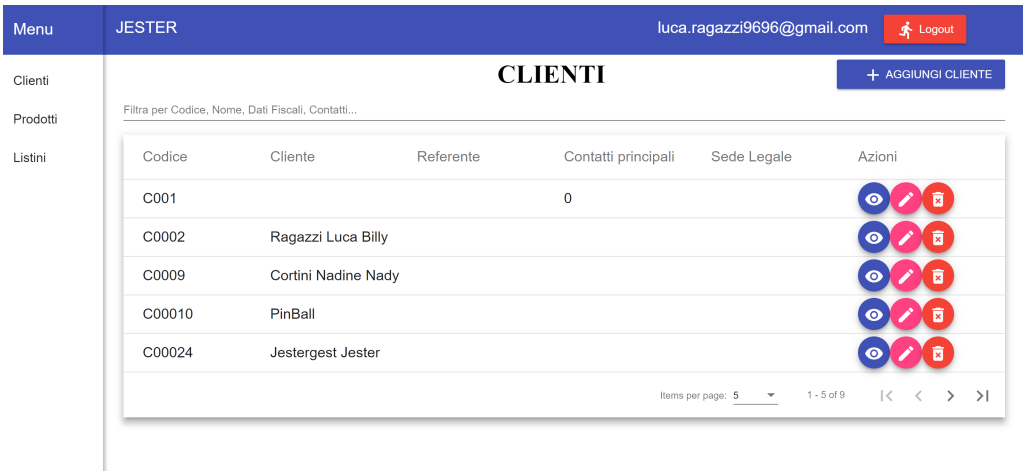


Figura 4.3: Interfaccia HomeClients

DeleteClient

Figura 4.4: Interfaccia DeleteClient

ViewProduct

Figura 4.5: Interfaccia ViewProduct

CreatePriceList

Figura 4.6: Interfaccia CreatePriceList

Capitolo 5

Testing e validazione

In questo capitolo verrà presa in considerazione la fase di validazione del sistema, con lo scopo di soddisfare i requisiti definiti nella fasi di analisi.

Lo sviluppo del testing ha seguito due percorsi differenti:

- Implementazione di *Unit Test*[1];
- Verifica di prove sperimentali effettuate direttamente sul software per soddisfare i tre scenari descritti nella sezione 2.4;

5.1 Unit Testing

Per lo sviluppo dei test si è ricorso all'utilizzo di due framework, già presenti in un'applicazione Angular: **Jasmine** e **Karma**.

5.1.1 Jasmine

Framework che supporta una pratica di sviluppo del software chiamata BDD[2]. Jasmine descrive i test in un formato leggibile anche da utenti meno esperti. È caratterizzata da diverse espressioni:

- **describe(string, function)**: funzione che descrive tutti i test effettuati su un particolare caso di studio;
- **it(string, function)**: funzione che definisce un test specifico;

[1] **Unit testing**: attività di collaudo e validazione di parti specifiche del sistema.

[2] **B**ehaviour **D**riven **D**evelopment: metodologia di sviluppo del software basata sul Test-Driven Development. BDD combina tecniche dell'ingegneria del software e della progettazione orientata agli oggetti per fornire strumenti per lo sviluppo software.

- **expect(actual)**: espressione che descrive il valore aspettato per il corretto funzionamento del test;

Di seguito è mostrato un esempio di test su "AppComponent", effettuato nella fase iniziale dello sviluppo:

```
import { TestBed, async } from '@angular/core/testing';
import { AppComponent } from './app.component';

describe('AppComponent', () => {
  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [
        AppComponent
      ],
    }).compileComponents();
  }));
  it('should create the app', async(() => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.debugElement.componentInstance;
    expect(app).toBeTruthy();
  }));
  it(`should have as title 'app'`, async(() => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.debugElement.componentInstance;
    expect(app.title).toEqual('app');
  }));
  it('should render title in a h1 tag', async(() => {
    const fixture = TestBed.createComponent(AppComponent);
    fixture.detectChanges();
    const compiled = fixture.debugElement.nativeElement;
    expect(compiled.querySelector('h1').textContent).toContain('Welcome
        to Tesi!');
  }));
});
```

Listato 5.1: app.component.spec.ts

Come primo passo, mediante la clausola *"beforeEach()"* si configura il componente che si desidera testare, in questo caso "AppComponent". Tramite la funzione "it(string, function)" sono stati testati rispettivamente la creazione dell'applicazione, il nome del titolo dell'applicazione e il rendering del titolo dentro un tag H1.

5.1.2 Karma

Karma è uno strumento che consente di generare automaticamente i test di Jasmine sia nel browser che da riga di comando, facendo risparmiare tempo al programmatore. Angular CLI gestisce già di default la sua configurazione.

5.1.3 Unit Testing di Jester

Si è deciso di effettuare i test su "FacadeService". In particolare sono stati testati alcuni metodi di CrudService e IndexedDBService.

Testing sulla richiesta GET di CrudService

```
describe('Testing CrudService', inject([HttpTestingController,
    FacadeService], (httpMock: HttpTestingController, service:
    FacadeService) => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [FacadeService],
      imports: [
        HttpClientTestingModule
      ],
    });
  });
  it('should fetch correct data from API', () => {
    let route: string = "/angular-jestergest-new/products.json";
    service.get_server(route).subscribe(data => {
      expect(data.count).toBe(8);
      expect(data.viewVar).toBe("products");
    });
    const req = httpMock.expectOne(route);
    expect(req.request.method).toEqual('GET');
    req.flush({
      count: 8,
      viewVar: "products"
    });
  });
  afterEach(inject([HttpTestingController], (httpMock:
      HttpTestingController) => {
    httpMock.verify();
  }));
}));
```

Listato 5.2: facade.service.spec.ts

In particolare è stata utilizzata la classe "HttpTestingController" con lo scopo di intercettare le richieste HTTP.

Testing su IndexedDBService

Per testare il funzionamento del servizio IndexedDBService si è validato il corretto funzionamento dell'aggiunta (POST) di un nuovo elemento e della lettura (GET) di uno specifico dato.

```
describe('Testing IndexedDBService', inject([FacadeService],
    (service: FacadeService) => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [FacadeService],
    });
  });
  it("should add a new client to clients collection", (done) => {
    let testElement = {
      "name": "Test name",
      "code": "Test code"
    };
    service.getAll('clients', res => {
      expect(res.length).toBe(2);
      done();
    })
    service.add('clients', testElement);
    service.getAll('clients', res => {
      expect(res.length).toBe(3);
      done();
    })
  });
  it("should retrieve the correct element", (done) => {
    service.get('products', 1, res => {
      expect(res.id).toBe(1);
      done();
    })
  });
}));
```

Listato 5.3: facade.service.spec.ts

5.2 Prove sperimentali

Dal momento che Jester è un'applicazione utilizzabile anche offline, il modo migliore per verificare le funzionalità del sistema, descritte nella sezione 2.1.1, è stato mediante il diretto utilizzo dell'applicativo con test sperimentali. In particolare, sono stati soddisfatti tutti i seguenti scenari:

1. Sessione di lavoro completamente online
2. Sessione di lavoro sia online che offline
3. Sessione di lavoro completamente offline

Unico vincolo dell'applicativo risulta essere che nella modalità completamente offline l'utente deve avere la sessione di lavoro già iniziata, ovvero deve risultare già autenticato. In questo modo sarà possibile entrare nell'applicativo e, sfruttando sia l'IndexedDB che il Service Worker, l'utente potrà godere di un'esperienza di lavoro in condizioni offline alla pari di quella online.

Conclusioni e sviluppi futuri

Obiettivo del progetto è stato l'implementazione di un'applicazione gestionale con determinate caratteristiche:

- Reattività di una Single Page Application;
- Funzionamento offline di una Progressive Web Application;

Dal punto di vista tecnologico ho affrontato lo studio di nuovi framework, molto utilizzati nello sviluppo web e non trattati nel corso di questa laurea triennale, mettendomi in gioco utilizzando le conoscenze acquisite durante il corso di Tecnologie Web, integrate dall'esperienza del tirocinio in azienda.

Angular si è rilevato un framework potente e versatile, tra i più completi e utilizzati nel mondo lavorativo odierno per la realizzazione frontend di applicazioni web. Pertanto il suo studio e la sua applicazione mi hanno arricchito professionalmente.

Dal punto di vista ingegneristico ho potuto utilizzare le conoscenze acquisite nei corsi di Programmazione ad Oggetti e di Ingegneria del Software, permettendomi di sviluppare un sistema partendo da un'attenta analisi, seguita dalla progettazione fino allo sviluppo vero e proprio con i relativi test finali.

La scelta di affrontare un tirocinio per tesi mi ha permesso di avvicinarmi al mondo lavorativo, osservando la professionalità dei "colleghi" nel lavoro quotidiano.

Sono riuscito a soddisfare tutti i requisiti del sistema, al quale si potrebbero applicare nuovi sviluppi futuri che riporto di seguito.

Accesso concorrente di più utenti: Jester non gestisce l'accesso simultaneo di più utenti nell'applicativo. Sarebbe interessante sviluppare questo aspetto, permettendo l'effettivo utilizzo dell'applicazione in un contesto aziendale.

Gestione di più dati: Jester gestisce solo prodotti, clienti e listini, al solo scopo di verificare la reattività e l'efficienza di un sistema Single Page Application che funzioni anche in modalità offline. Per il suo futuro reale utilizzo all'interno di un'azienda sarà necessario implementare anche la gestione di ulteriori dati.

Ringraziamenti

Per questo percorso di tesi ci tengo a ringraziare innanzitutto tutti i ragazzi della Librasoft, sempre simpatici e ospitali fin dal primo giorno. In particolare un grazie a Piergiorgio e a Stefano per essere stati degli ottimi tutor e avermi accompagnato dall'inizio del tirocinio curriculare fino al termine del progetto di tesi.

Un sincero ringraziamento va ai miei genitori, a mia sorella, a Sibo e soprattutto ai miei nonni. Senza loro non sarei di certo arrivato fino qua.

Ringrazio inoltre tutti i miei amici, sia quelli con cui sono cresciuto insieme che quelli incontrati in corso d'opera, sia nello studio che nel basket.

Infine un ringraziamento speciale spetta a Nadine, la mia piccola metà che è sempre riuscita a sostenermi e a spronarmi per dare il meglio. Grazie di cuore.

Bibliografia

[1] *5 (Practical) Tips to Help You Secure Your Single Page Application*, https://resources.whitesourcesoftware.com/blog-whitesource/5-practical-tips-to-help-you-secure-your-single-page-application

[2] *Angular*, https://angular.io

[3] *Introduction to components*, https://angular.io/guide/architecture-components

[4] *IIntroduction to services and dependency injection*, https://angular.io/guide/architecture-services

[5] *Service Workers for Offline-First Apps*, https://auth0.com/blog/creating-offline-first-web-apps-with-service-workers

[6] *The service worker life cycle*, https://developers.google.com/web/fundamentals/primers/service-workers

[7] *Middleware*, https://book.cakephp.org/3.0/en/controllers/middleware.html#

[8] *Architettura Client-Server*, http://lnx.poggiodelpapa.com/linguaggi/vb2010_sql/vb2010_03_04_client_server.php

[9] *Architecture overview*, https://angular.io/guide/architecture