Relazione progetto Progettazione di applicazioni web e mobili

Link GitHub: https://github.com/SamueleGalasso/PAWM-project

Node.js

Ho deciso di sviluppare un classico e-commerce, utilizzando come ambiente di sviluppo node.js che è un framework runtime che utilizza come linguaggio di programmazione JavaScript costruito sul motore JS V8 di Google Chrome, grazie a quest'ultimo è possibile scrivere applicazioni web in JavaScript lato server con caratteristiche chiare in termini di velocità e scalabilità. La peculiarità di node.js è che si tratta di una soluzione basata su un modello di I/O server-side asincrono che opera su eventi. Cosa significa tutto ciò? Node richiede al sistema operativo di ricevere notifiche al verificarsi di determinati eventi, rimanendo quindi in "sleep" fino alla notifica stessa: solo in tale momento torna attivo per eseguire le istruzioni previste nella funzione di callback, così chiamata perché da eseguire una volta ricevuta la notifica che il risultato dell'elaborazione del sistema operativo è disponibile. In sintesi mentre Node.js sta elaborando una procedura e aspetta che venga portata a termine può già partire con altre operazioni, da qui il termine asincrono. Questo può portare a molti vantaggi in termini di performance. In questo progetto le operazioni asincrone sono state largamente usate, in particolare quando abbiamo una funzione asincrona la soluzione migliore è quella di utilizzare una Promise cioè un object che rappresenta l'eventuale completamento dell'operazione asincrona al quale "attacchiamo" then().catch(), il primo per gestire la promise e il secondo per gestire eventuali errori generati dalla funzione asincrona.

Express.js

Per quanto sia performante e di uso comune al giorno d'oggi Node.js resta una tecnologia di basso livello, per velocizzare il lavoro è possibile utilizzare particolari framework che mettono a disposizione funzioni che facilitano lo sviluppo e rendono il codice molto più chiaro e leggibile.

In questo progetto come micro-Framework ho deciso di utilizzare Express.js, si tratta di un framework che ci permette comunque di restare ad un livello abbastanza basso non distaccandoci troppo da quello di Node.js, ma porta comunque moltissimi vantaggi come ad esempio generare percorsi (URL) per l'applicazione o di utilizzare templating-engine per il front-end della stessa.

Templating engine: EJS

Per il front-end di questa applicazione ho deciso di utilizzare un templating engine in particolare ejs (Embedded JavaScript Templating), questo perché avevo bisogno di restituire in seguito ad una callback pagine web dinamiche, infatti ejs utilizza plain JavaScript per generare HTML. Questo templating engine ci permette di caricare dati dal server nella view a runtime che servono per effettuare semplici operazioni come cicli (for,while) o condizioni (if,if-else ecc).

Database:

NoSQL,MongoDB,Mongoose

Per quanto riguarda il database ho deciso di utilizzare MongoDB, il quale utilizza NoSql che non è uno specifico linguaggio, ma è il termine universalmente accettato per raggruppare un insieme di tecnologie per la persistenza dei dati che funzionano in modo sostanzialmente diverso dai database relazionali (SQL). MongoDB è un tipo di database orientato ai documenti, cioè memorizza dati in formato JSON-like, si tratta in parole povere di una sorta di albero che può contenere ed annidare molti dati, lavoriamo con schemas e models e grazie a ciò possiamo definire i dati come vogliamo. I documenti sono raggruppati in collections, consente inoltre servizi di alta disponibilità (semplice replicazione di un DB) e garantisce la scalabilità automatica per non influire pesantemente sulle performance.

Mongoose è una libreria molto utile per collegare node.js e MongoDB, rende lo sviluppo più produttivo ed efficiente, infatti lavora sullo stile di un ORM (object relational mapping, come sequelize per sql), la definizione corretta è però ODM (object document mapping). In fase di sviluppo ho utilizzato MongoDB Compass che è una GUI (graphical user interface) che permette di visualizzare e lavorare con i dati presenti nel DB in modo molto più semplice e veloce.

Nel DB ho deciso di creare 4 diverse collections:

-Products utilizzata per memorizzare i prodotti presenti nello store, che hanno un titolo un prezzo una descrizione, l'URL dell'imagine (ho avuto problemi con l' upload di file, quindi purtroppo ho deciso di utilizzare l'URL dell'immagine) e l' userID cioè l'id dell'utente che carica nello shop il prodotto.

-Orders utilizzata per memorizzare gli ordini effettuati dagli utenti, come campi ha un'array di prodotti contenente il prodotto stesso la quantità e l'user che ha effettuato l'ordine.

-Users qui vengono memorizzati tutti gli utenti registrati, ha come campi un'email, una password, dei resetToken utilizzati per il reset della password, e un carrello contenente un'array di items che hanno un productId e la quantità. A differenza delle altre collections definite nel folder models, questa presenta alcune funzioni tipiche di un'utente, ossia aggiungere prodotti al proprio carrello (addToCart), rimuovere un prodotto dal carrello (removeFromCart) e pulire completamente il carrello (clearCart).

-Sessions questa collection serve per tenere traccia degli utenti loggati, infatti dal momento in cui un'utente effettua il login viene creata la sua sessione e nel momento in cui effettua il logout o viene raggiunta l'expiration date la sessione di questo utente viene distrutta e non risulterà più loggato. Ho deciso di memorizzare l'informazione che un'utente è loggato nel back-end in modo che questa informazione non sia manipolabile dal front-end come accade per i cookie. In questi ultimi memorizzo l'hash-id della sessione corrente, in questo modo non perdiamo i dati dell'utente dopo ogni richiesta e non li rendiamo visibili ad altri utenti. Senza la sessione ad ogni request l'utente deve effettuare il login perché ogni richiesta interagisce in modo separato tra loro, la sessione le connette.

MVC Pattern

Ho cercato di seguire e rispettare il pattern MVC (model, view, controller), cioè di distribuire e separare al meglio le responsabilità all'interno del progetto per quanto riguarda dati, viste e request/response handler.

-Model: in questo folder è presente tutto ciò che riguarda i dati e il database, infatti troviamo, order, product e user già sopra citati.

-View: qui sono presenti tutte le viste utilizzate all'interno del progetto.

Ciò che vede l'utente e la parte con cui interagisce per inviare e ricevere dati dal/al back-end

-Controller: si tratta del principale responsabile della gestione delle request e delle response. Quando un'utente interagisce con la view questa genera un'appropriata richiesta, quest'ultima viene gestita dai controllers. In sostanza renderizzano un'opportuna vista con i dati del model come response in seguito ad una request da parte dell'user.

All'interno di questo folder troviamo tre file.js:

-admin.js: controlla e gestisce le richieste dell'utente registrato e autenticato (admin) e permette di svolgere tutte le classiche funzioni di un e-commerce tra cui aggiungere prodotti nello shop, cancellare o editare il proprio annuncio di vendita.

-auth.js: controller utilizzato per gestire l'autenticazione all'interno dell'applicazione.

-error.js: questo controller gestisce gli **errori 404** page not found nel caso in cui non venga trovata una particolare route e **500** per errori legati a problemi server-side.

-shop.js: questo controller gestisce tutte le richieste legate allo shop, quindi il caricamento dei prodotti presenti nel DB, il caricamento del carrello, del checkout ecc. Inoltre con l'aggiunta di **Stripe** è possibile effettuare dei pagamenti in seguito ad un ordine. **Stripe** semplifica e aiuta a rendere più fluide e sicure tutte le fasi del processo di acquisto online. Grazie all'utilizzo di strumenti per la gestione dei flussi di pagamento e anti-frode rende molto sicuro il proprio utilizzo.

Validation

Validation è un pattern utilizzato per verificare se un determinato input di un'utente sia valido o meno, ho deciso di fare questi controlli server-side perché secondo me è un tipo di approccio più sicuro. Come pacchetto ho usato "express-validator/check", sulle routes che in particolare gestiscono le post request controllo se ad esempio quando un'utente deve inserire dati in un form viene rispettata la lunghezza minima di caratteri, se si tratta di una stringa o meno, se al momento della registrazione dell'utente il campo password e confirmPassword matchano ecc., in caso negativo vengono "flashati" dei messaggi d'errore mantenendo comunque i vecchi dati inseriti precedentemente nei vari campi (oldInputs).

Pagination

Con il termine Pagination parliamo di un semplicissimo pattern che ci permette di dividere il contenuto di una pagina in più pagine. Ho aggiunto una serie di link ad ogni view che necessiti di paginazione, che cambiano l'URL aggiungendo dei query parameters (es. …?page=1), questo ci permetterà di scorrere le varie pagine dello shop visualizzando i vari prodotti, di default abbiamo due items per page, valore modificabile solo nel back-end dell'applicazione.

Authentication

Il pattern dell'autenticazione per un'applicazione web è fondamentale, permette all'utente di registrarsi ed effettuare il login all'interno del sito, "sbloccando" alcune funzionalità (routes, URL) proibite a semplici utenti non loggati. Partiamo dal sign-up form, utilizzato per registrarsi, dal momento in cui vengono riempiti i campi e la validazione degli input ha avuto successo viene innanzi tutto cryptata la password, ho scelto di utilizzare "bcryptjs" come pacchetto esterno, questo per aumentare la sicurezza dei dati privati dell'utente che vengono memorizzati nel database. Una volta cryptata la password creo un nuovo utente e lo memorizzo nel database, successivamente se non sono stati lanciati errori/eccezioni viene inviata un'email all'utente appena registrato, informandolo che l'operazione ha avuto successo. Per inviare e-mail ho utilizzato "nodemailer" come pacchetto esterno. Nel login form, una volta superati i controlli per la validazione degli input, cerco nel database un user la cui email combaci con quella inserita dall'utente, se ho avuto successo ora utilizzo bcrypt per comparare le due password, in caso affermativo cioè se le due password combaciano, il login è stato effettuato con successo altrimenti viene settato lo status code **422** e viene "flashato" un messaggio d'errore per migliorare l'esperienza dell'utente all'interno del sito. Quando viene effettuato il login l'utente ha ora a disposizione diverse funzionalità tipiche di un user registrato, ma ogni volta che effettua una nuova request deve di nuovo loggare nel sito, questo perché le request interagiscono tra loro in modo separato. Per risolvere questo tipo di problema nel momento in cui un'utente effettua il login con successo viene creata una sessione e salvata nel database, al suo interno viene memorizzato un campo "isLoggedIn" settato a true (se loggato) grazie al quale possiamo verificare se un'utente è stato autenticato o meno e quindi imporre delle restrizioni per l'accesso a determinate routes (vedi "routes" folder). Nel momento in cui viene effettuato il logout la sessione viene "distrutta" e l'utente risulterà cosi "sloggato" e privato delle funzionalità tipiche di un user autenticato.

Per migliorare l'esperienza dell'utente ho deciso di aggiungere la funzionalità di reset password nel caso in cui venga smarrita o dimenticata. Una volta inserita l'email dell'account di cui si vuole resettare la password, viene creata una stringa di randomBytes e memorizzata in un token che viene a sua volta salvato nel campo resetToken dell'utente salvato nel database, inoltre con resetTokenExpiration a cui assegno Date.now() + 3600000, setto una data di expiration per la validità di questa request (un'ora di tempo prima che scada). Successivamente viene inviata un'email contenente il link di reset che avrà nell'URL come parametro il resetToken, verremo così reindirizzati in una pagina che ci permetterà di inserire una nuova password che verrà nuovamente cryptata e salvata nel DB al posto della vecchia password e gli altri due token di reset verranno settati ad undefined.

CSRF ATTACKS

Per quanto riguarda la sicurezza dell'applicazione ho deciso di aggiungere una protezione contro attacchi di tipo CSRF utilizzando il pacchetto esterno "csurf". Colui che effettua questo tipo di attacchi può indurre in modo non intenzionale la vittima a compiere azioni del tipo: cambiare password, acquistare prodotti, inviare denaro ecc. Vengono utilizzati i session cookies dall'hacker per effettuare questo tipo di attacchi, infatti la vittima deve innanzi tutto essere autenticata. Per proteggerci da questo tipo di attacchi col pacchetto "csurf" genero un csrf-token in cui viene memorizzato un hashed value non indovinabile o decriptabile dall'hacker, quindi dopo ogni post-request controllo la validità di questo token nel front-end (l'ho reso un campo locals per facilitarmi il lavoro di renderizzazione).

Samuele Galasso n°matr. 101260