Tecniche - 06-02 - PROGETTO PC-CONFIGURATOR

Il primo passo consiste nel definire il nome del progetto e creare la relativa cartella principale. Nell'esempio, il progetto si chiama "PC Configurator". All'interno della cartella del progetto vengono poi create tre sottocartelle distinte: una cartella css, una cartella js e una cartella img.

All'interno della cartella js viene creata una sottocartella chiamata model. Le entità esistono anche in JavaScript e questo significa che, in questa fase, verrà inserita un'entità che verrà chiamata Configuration.
In JavaScript le classi sono funzioni e le funzioni sono oggetti. Non è necessario dichiarare preventivamente le proprietà, ma è sufficiente impostarle. Il costruttore può essere uno solo e al suo interno vengono definiti gli elementi firstName, lastName, processor, ram e ssd.
È anche possibile procedere al contrario, cioè non passare alcun valore iniziale, perché all'inizio il PC viene creato vuoto.

Si inizia a scrivere il file definendo il metodo getErrors, che ha il compito di controllare che l'entità sia valida. Il controllo di esistenza si basa sulla valutazione di un valore truthy, ovvero approssimativamente vero: una stringa piena viene interpretata come true, mentre una stringa vuota viene interpretata come false. In JavaScript, infatti, qualsiasi valore può essere utilizzato come un booleano.
Nel file viene definito il metodo getErrors, che restituisce un insieme di errori relativi alla validità dell'entità. All'interno del metodo viene inizializzato un vettore vuoto chiamato errors, che serve a raccogliere i messaggi di errore. Viene poi effettuato un controllo di esistenza sui vari attributi dell'oggetto. In particolare, se il valore firstName non è presente, viene aggiunto un messaggio di errore relativo al nome del cliente. Lo stesso tipo di controllo viene applicato al lastName e al processor, verificando che i rispettivi valori siano valorizzati.

Successivamente viene controllato il valore della ram, verificando che esista e che sia maggiore di zero; in caso contrario viene segnalato un valore non valido. Lo stesso controllo viene effettuato per il valore ssd, che deve anch'esso essere presente e maggiore di zero. Tutti questi controlli si basano sul concetto di valori

truthy e falsy di JavaScript, che permettono di valutare direttamente le proprietà come condizioni booleane.

Nei return non si va a capo perché, in caso contrario, JavaScript interpreta l'istruzione come terminata automaticamente. Questo comportamento è dovuto all'Automatic Semicolon Insertion, che può causare il ritorno di un valore inatteso se il return viene seguito da un'interruzione di riga.

In Java l'uso di this è opzionale, mentre in JavaScript è obbligatorio. In JavaScript, infatti, è necessario utilizzare this per riferirsi alle proprietà dell'oggetto all'interno dei metodi.
JavaScript è un linguaggio duck typed: il tipo di una variabile viene determinato dal suo contenuto e dal comportamento che espone. In altre parole, se un oggetto "starnazza e ha le ali", viene trattato come appartenente a quel tipo, indipendentemente dalla sua definizione formale.

In questo contesto, la flessibilità di JavaScript si manifesta anche nelle funzioni, le quali possono restituire qualsiasi tipo di dato, inclusa la possibilità che una funzione restituisca un'altra funzione. L'entità appena creata verrà collegata ad una pagina HTML e in particolare ad un form. Questo approccio di lavoro si chiama MVC e lo sto applicando al front-end: la classe appena creata rappresenta il nostro model, ovvero la nostra entità. Successivamente, entro nella cartella principale e creo un nuovo file che chiamerò pcconfigurator.html.

Dopo aver creato il file HTML, inserisco i miei elementi, come il logo, e li collego a un file di stile chiamato styles.css, che posizionerò nella cartella css per gestire la formattazione e il layout della pagina.

La differenza principale tra `span` e `div` riguarda il tipo di elemento che rappresentano e il comportamento nel flusso del documento. Il `div` è un elemento di tipo block, quindi occupa tutta la larghezza disponibile e va a capo automaticamente; viene usato per raggruppare blocchi di contenuto più grandi o sezioni della pagina. Lo `span`, invece, è un elemento inline, quindi occupa solo lo spazio necessario al contenuto e non va a capo; viene utilizzato per stilizzare o modificare parti di testo o elementi all'interno di un blocco senza creare un'interruzione di riga.
Adesso devo scrivere il form e associare il relativo file di stile, che chiamerò pcconfigurator.css, per gestire l'aspetto e la disposizione dei suoi elementi.
Ora devo collegare il form al model, stabilendo la connessione tra la view e il

model in modo che i dati inseriti dall'utente vengano gestiti correttamente dalla nostra entità.

Quando il contenuto del form cambia, cambia anche l'oggetto, e viceversa: questa è la logica alla base della programmazione web con MVC. Quindi creo un oggetto `configuration` come model e realizzerò una view composta principalmente dal form. Per mantenerli sincronizzati, è necessario introdurre il controller, che fa da intermediario tra la view e il model, gestendo gli aggiornamenti in entrambe le direzioni.

Il model si occupa della logica, dei dati e dei calcoli: qui vengono gestite le informazioni e le operazioni necessarie. La view invece si concentra sull'interfaccia, riceve gli input dall'utente e mostra i risultati, senza occuparsi dei calcoli o della logica interna. Il controller ha il compito di allineare e sincronizzare i due: riceve gli input dalla view, aggiorna il model, e fa in modo che eventuali cambiamenti nel model si riflettano automaticamente sulla view, mantenendo tutto coerente. In pratica, ogni cambiamento in un componente viene propagato all'altro tramite il controller, garantendo una gestione ordinata dei dati e dell'interfaccia. Nella fase di input, sotto la cartella JS creo una nuova cartella chiamata controller. All'interno del controller creo un nuovo oggetto `new Configuration` e lo sincronizzo tramite una funzione `synch`, che raccoglie i dati dal form e li inserisce nel model, mantenendo aggiornati i valori dell'oggetto in base agli input dell'utente.

Il controller che gestisce la sincronizzazione tra il form e il model. Nella funzione synch, nella fase di input, raccogliamo i valori dai campi del form (pcform.processor.value, pcform.ram.value, ecc.) e li assegnamo alle proprietà corrispondenti del model controller.configuration. Nella fase di output, invece, prendiamo un dato calcolato dal model, come il prezzo (controller.configuration.getPrice()), e lo riportiamo nel form (pcform.price.value).

In pratica, stiamo implementando il ruolo del controller: prende gli input della view, aggiorna il model e poi aggiorna la view con i risultati calcolati dal model ogni volta che un utente modifica un campo del form, bisogna sincronizzare immediatamente il model con i nuovi valori. In pratica, per ogni input del form si associa un evento onchange o oninput che richiama la funzione di sincronizzazione del controller. In questo modo, il model è sempre aggiornato rispetto alla view e, se necessario, la view può essere aggiornata subito con i calcoli o i valori derivati dal model.È necessario importare prima il modello e

successivamente il controller, perché il controller crea una nuova istanza della configurazione.
Nella console del browser è possibile richiamare qualsiasi elemento, permettendo di modificare tutti i siti web che vengono caricati sulla propria macchina.

Nel pattern Model-View-Controller (MVC), la view si occupa esclusivamente della visualizzazione delle informazioni, mentre qualsiasi modifica ai dati è gestita dal controller, che funziona come elemento di collegamento tra modello e vista. Il controller agisce da punto di ingresso del pattern e coordina l'interazione tra i componenti: quando l'utente interagisce con l'applicazione attraverso la view, gli eventi generati vengono gestiti dal controller, il quale determina quali metodi del modello invocare per modificare i dati. Una volta che il modello viene aggiornato, questo notifica la view dei cambiamenti avvenuti attraverso un meccanismo di observer, permettendo alla vista di recuperare i dati aggiornati e aggiornare la propria interfaccia di conseguenza

BACK END
Per preparare il backend con API REST in Spring Boot, genera il progetto tramite Spring Initializr includendo Spring Web per i controller, Spring Data JPA per repository automatici, MySQL Driver per la connettività e DevTools per riavvii automatici, poi configura application.properties con l'URL del database jdbc:mysql://localhost:3306/nome_db, credenziali e spring.jpa.hibernate.ddl-auto=update dopo aver creato il database vuoto in MySQL. Sposta tutti i file statici del frontend HTML, CSS e JS in src/main/resources/static affinché vengano serviti direttamente dal backend, consentendo al controller di gestire le chiamate fetch dalla vista mantenendo la sincronizzazione con il modello. Nelle entità JPA definisci campi con validazioni server-side ignorando completamente i dati in arrivo dal client poiché non esiste sicurezza lato client

Dopo aver definito l'entità Configuration nel package model con annotazioni @Entity, @Id e @GeneratedValue per l'identificativo intero, il passo successivo consiste nella creazione dell'interfaccia repository nel package com.generation.pcconfigurator.model.repository. Qui si implementa ConfigurationRepository estendendo JpaRepository<Configuration, Integer>, dove Configuration rappresenta l'entità gestita e Integer il tipo della chiave primaria, ottenendo automaticamente tutti i metodi CRUD standard come save, findById, findAll e delete senza scrivere codice di implementazione.

Nel mondo reale delle API REST, non si può garantire che ogni operazione restituisca direttamente l'entità Configuration come previsto, poiché le cose possono andare bene o male a causa di errori di validazione, conflitti nel database o eccezioni impreviste. Per questo motivo, i controller Spring Boot utilizzano ResponseEntity come tipo di ritorno standard, permettendo di controllare lo status HTTP, gli header e il body JSON in modo flessibile, senza promettere che il contenuto sia sempre l'entità salvata ma assicurando una response strutturata che il client frontend statico possa interpretare correttamente.

ResponseEntity è parametrizzata come ResponseEntity<Configuration> per indicare che il body della risposta può contenere un oggetto Configuration quando l'operazione va a buon fine, ma questa parametrizzazione non costituisce una garanzia di successo universale poiché le operazioni potrebbero fallire per validazioni errate, errori di persistenza o condizioni impreviste. In questo modo il metodo controller restituisce sempre una ResponseEntity<Configuration>, ma controlla esplicitamente il BindingResult dopo @Valid @RequestBody Configuration config: se config.hasErrors() è true, invia ResponseEntity.badRequest().body(getError(config)) con status 400 e body di errore invece dell'entità, mentre solo se tutto è valido procede con repository.save e ResponseEntity.ok(config) con status 200.

Le API REST devono gestire in modo esplicito gli errori di validazione per assicurare un elevato livello di robustezza del sistema. In questo modo, il controller mantiene una sincronizzazione sicura tra il modello e la vista statica, rifiutando gli input non validi provenienti dal client e fornendo un riscontro preciso senza compromettere l’integrità del database.

Per collegare tutti i file frontend, bisogna metterli in static e completare il file JS con la funzione callAPI, posiziona l'intera struttura delle cartelle HTML, CSS e JS direttamente in src/main/resources/static del progetto Spring Boot, preservando la gerarchia.