Errori frequenti nel progetto d'esame
Applicazioni Web I, a.a. 2022/23

Consigli alla luce dell'esperienza degli esami degli anni passati
v. 3.1 (in rosso variazioni rispetto v. 3.0)

Nota: l'elaborato può essere consegnato in qualsiasi condizione indipendentemente dal soddisfacimento dei punti descritti nel seguito, che non sono una condizione necessaria per arrivare alla sufficienza (ad eccezione della corretta modalità di consegna). In generale, però, soddisfare i punti dovrebbe garantire, in generale, una valutazione buona perché tipicamente si evitano i problemi di maggior gravità.

PROBLEMI DI LOGICA DELL'APPLICAZIONE che in generale portano a perdite di punti

- NON si devono creare ID sul client quando devono essere identificativi (chiavi uniche) nel DB: gli ID DEVONO essere creati dal server, e poi restituiti al client.
- Per gli ID da generare alla creazione di un nuovo elemento unico, usare di preferenza il meccanismo fornito dal DB (es. colonna auto-increment). Altre soluzioni sono in generale sbagliate e portano alla perdita di punti (creazione/calcolo sul client, calcolo del max sulla colonna del DB ecc.). Si ricordi che l'applicazione è utilizzabile da più utenti contemporaneamente, che possono anche non lavorare sugli stessi dati, ma l'unicità dell'ID deve essere garantita. Inoltre, l'ID generato dal DB è, in generale, non direttamente selezionabile o manipolabile dall'utente, quindi il suo valore non dovrebbe influenzare la logica dell'applicazione o essere visualizzato esplicitamente nell'interfaccia dell'applicazione.
- E' inaccettabile fare assunzioni sul client riguardo agli identificativi presenti nel database. Per esempio, se nel progetto è richiesto avere 4 utenti, il codice NON deve mai assumere che i loro identificativi siano 1,2,3,4 e tantomeno fare operazioni matematiche su tali identificativi, per es. per ricavare un indice in un vettore ecc. Gli identificativi devono essere trasferiti come proprietà degli elementi di cui fanno parte (per es., nel caso degli utenti, insieme al nome). In un'applicazione generica gli identificativi potrebbero essere anche non consecutivi e anche non numerici.
- Usare sempre il metodo corretto nello scrivere e chiamare le API HTTP (es. mai POST al posto di GET, GET al posto di POST, PUT al posto di POST, ecc): ogni metodo ha la sua semantica corretta.
- E' necessario verificare sempre, lato server, i ruoli/permessi di coloro che scrivono informazioni nel DB (aggiunte, modifiche, ecc...): non basta l'autenticazione iniziale, ma bisogna verificare che l'utente/utilizzatore autenticato nella sessione abbia il permesso di fare la cosa (per es. con un'aggiunta di condizione "where" su una query oppure una apposita query addizionale su db nel codice lato server).
- Chi non pone attenzione all'autenticazione almeno di base sul server non potrà ottenere il massimo del punteggio, in particolare chi confonde la verifica lato client con quella lato server, che è errore molto grave. Chi afferma: "sul client ho controllato che non si possa fare l'azione X..." non ha compreso la problematica, e non potrà avere il massimo. La verifica della correttezza dei valori e della possibilità di svolgere determinate operazioni solo lato client è concettualmente sbagliata, come chiarito a lezione, ed è uno dei motivi per cui esistono così tanti problemi di sicurezza nelle applicazioni web che ci circondano. Sarà quindi fortemente penalizzata all'esame.
- EVITARE di scrivere regular expressions (regex) per verificare la correttezza formale di

informazioni/dati che sono facilmente gestibili da una libreria: si perde tempo, è molto probabile sbagliarsi, e all'esame spesso si testa un caso non considerato. Esempio: verifica correttezza email o data. Un programmatore in generale non deve passare il suo tempo a scrivere regex per considerare anni bisestili e altre particolarità, deve usare una libreria (ragionevolmente affidabile), far convertire e trasferire le date in qualche formato standard se serve (es. formato stringa ISO).

- NON scrivere un unico componente enorme con la logica di quasi tutta l'applicazione: illeggibile, ingestibile, e quasi sicuramente con errori (es. dipendenze useEffect difficili da gestire ecc.)
- NON inventarsi cose in più non richieste, in particolare se divergono dal testo trascurandone anche solo una parte. Per esempio: se si chiede di aggiungere un nuovo corso specificando (nome, codice, crediti), prendere un corso da una lista predefinita è sbagliato: se lo scelgo da una lista predefinita NON posso aggiungere il nome che voglio, e in più non implemento (cioè mi semplifico indebitamente il progetto) una parte di controlli sul form che definisce il corso, per es. che il numero di crediti sia un numero, e maggiore di 0.
- NON inventarsi formati di dati strani per il trasferimento compatto delle informazioni da client a server e viceversa (es. separando i campi manualmente con virgole o altri simboli speciali, e tra l'altro dovendo poi gestire con pezzi di codice aggiuntivo questi casi). In generale, i campi di testo di un form (titoli, testi, ecc.) devono poter supportare spazi, punti e virgola, virgole e simboli vari. Dove abiti? Ad Ascoli Piceno (nome con spazio). A Mondovi' (con l'accento), e così via. Usare il JSON per serializzare/deserializzare informazioni contenute nelle stringhe Javascript, come viene spiegato nel corso.
- Evitare di usare useEffect se non serve effettivamente. Per esempio: evitare di usarla per reagire al cambiamento di un valore di un campo di un form. Se il form è di tipo controlled, l'handler può gestire la cosa. Molte useEffect nel codice, in particolare per casi facilmente gestibili diversamente, rendono i componenti difficili da capire, gestire, modificare. Invece, usare useEffect in tutti i casi in cui è effettivamente necessaria (caricamento asincrono di dati ecc.). Evitare in generale di usare useEffect con dipendenza su location o history per gestire il cambiamento di pagina quando la paginazione è stata implementata tramite un Router. (E' invece normale usare useParam per mostrare contenuto diverso dipendente da porzioni di URL)
- Se non è esplicitamente richiesto qualcosa di diverso, le operazioni che lavorano sul DB a seguito della chiamata di una API possono includere più query eseguite separatamente (ossia non in una transazione, che è complicata e difficile da implementare in sqlite3, quindi non è richiesta per il corso di Applicazioni Web I). Si assume per semplicità che tra una query e l'altra durante l'esecuzione del codice che corrisponde ad una API del server non avvengano modifiche al DB.

  NB: NON si può assumere, invece, che tra una chiamata ad una API e un'altra chiamata ad una API il DB rimanga necessariamente senza modifiche fatte da altri.
- Di conseguenza, una operazione atomica a livello logico deve essere realizzata all'interno di un'unica API. Il classico caso è la creazione/modifica/cancellazione di una informazione complessa che richiede più operazioni lato server (es. più query). Per esempio, creazione/modifica/cancellazione di un questionario con tutte le sue domande, di un piano di studi con i relativi corsi, utente con tutte le risorse ad esso collegate, ecc. E' gravemente errato effettuare più chiamate dal client verso le API del server per realizzare un'operazione atomica a livello logico.
- Per lo stesso motivo, in generale NON si deve creare un'API (e quindi una chiamata dal client al server) per ogni operazione da fare sul DB. In altri corsi può essere stato indicato qualcosa di simile (in altri contesti), ma per questo corso bisogna riferirsi alle linee guida fornite durante la spiegazione di come progettare delle API direttamente richiamabili dal browser.

- Nel caso sia necessario effettuare delle modifiche ad una struttura complessa che deve rispettare dei vincoli dati, si deve PRIMA controllare che i vincoli siano soddisfatti per la situazione che si verrà a creare, e DOPO effettuare le modifiche (es. query) che vanno ad implementarle, così da evitare situazioni di errore che diventano difficili da annullare (soprattutto in assenza delle transazioni, che è preferibile non usare per semplicità).

PROBLEMI DI INTERFACCIA, se gravi portano a perdite di punti, di sicuro disturbano le prove di funzionalità

- Se l'utente non ha ancora fatto nulla, non devono apparire errori. Per esempio, se un form è ancora da compilare, non deve mostrare già errori (es. campi del form bordati di rosso). Altro esempio: login con username e password rossi perché vuoti, appena arrivati sulla pagina. Solo dopo delle azioni dell'utente, per es. pressione del bottone login (o Invio o submission del form) devono comparire gli eventuali errori (es. è richiesta un'email, manca @, ecc.).
- Per il form di login: se dopo aver inviato le credenziali non ci si può autenticare (per es. perché il server le rifiuta) questo deve essere indicato all'utente, altrimenti l'utente neanche si accorge dell'invio e che c'è stato un problema. Se ci sono più posti in cui fare il login (per es. pagina apposita ma anche campi username e password sempre visibili in alto), TUTTI devono mostrare un messaggio in caso di problemi, non solo quello dell'apposita pagina. Non inserire modi di autenticazione se non richiesti nel testo (es. "login as guest"): usare o dare l'impressione di doversi autenticare, seppur senza credenziali, quando non richiesto, sarà considerato un errore.
- Testare TUTTI i modi di usare i campi dei form, in particolare i campi HTML5: se si scrive direttamente il numero anziché usare le freccette del campo numerico (es. campo numerico "crediti di un corso"), la scrittura diretta del numero deve funzionare, altrimenti l'applicazione perderà punti perché certe funzionalità non si riescono a testare. Attenzione anche che, se non specificato, non è detto che i numeri siano interi.
- NON fare assunzioni non specificate sui dati: un codice identificativo, per esempio, NON è necessariamente numerico, a meno di diverse indicazioni. L'ID interno dell'applicazione può essere numerico, ma all'utilizzatore dell'applicazione NON interessa questo ID interno ed esso non deve essere utilizzato come richiesta nell'interfaccia verso l'utente. Se il codice è visibile (per es. "codice corso"), non assumere che sia solo numerico, se non è esplicitamente indicato nel testo. Il codice di un oggetto/elemento/prodotto generico NON è necessariamente numerico. Può essere comodo implementarlo come numero, ma comunque anche in questo caso è concettualmente sbagliato acquisirlo/gestirlo con campi di un form con per es. type=number.
- NON fare assunzioni inutili, non richieste o sbagliate sui dati: es. almeno 6 caratteri per il nome di un corso (esempi di 4 caratteri: "math", o "greco", "arabo"). In generale NON è compito dello sviluppatore ragionare su questo aspetto, se non è richiesto nella traccia. Testare se è vuoto o no ha senso (eventualmente rimuovendo gli spazi a inizio/fine), ma il numero di caratteri in generale no. Altro esempio: specificare luogo (min 6 caratteri): Roma e Rho non potrebbero essere indicate.
- Evitare di restringere le possibilità di scelta se non indicato dalla traccia. Esempio: se prenoto un appuntamento di cui devo specificare la durata, la durata (es. in minuti) NON deve essere un multiplo di 10, se non esplicitamente richiesto o derivante dalla logica del problema. Se è un problema di interfaccia (perché risulterebbe es. in una combo box troppo lunga), cambiare interfaccia (es. campo numerico o di testo verificato successivamente). Esempio: Il numero di crediti di un corso non deve essere predeterminato (es. 6 8 10 12), ma libero, eventualmente anche con la virgola, a meno di vincoli differenti nel testo. Nel dubbio, chiedere chiarimenti. In generale vincolare questo genere di valori comporta più lavoro, non richiesto e non valutato, e spesso non consente di testare agevolmente l'applicazione.

- Se la scelta possibile di un campo è ristretta (es. solo un prodotto, persona, codice corso, o altro già esistente nell'applicazione) è consigliabile usare un elemento adeguato nell'interfaccia, per es. combo box (menù a tendina) e NON un campo libero, tantomeno richiedendo all'utilizzatore l'ID interno oppure mostrando solo l'ID come identificativo di scelta (come farebbe l'utilizzatore a sapere a cosa corrisponde l'ID?)
- Se è richiesta la selezione di un singolo elemento tra N possibili scelte (es. un utente tra quelli disponibili), è necessario che l'operazione di selezione, internamente, identifichi sempre il singolo elemento in maniera univoca. Internamente non devono essere usati identificativi che non sono garantiti essere unici (es. nome, o nome e cognome, ecc.).
- La selezione di un elemento deve essere facilmente interpretabile dall'utente: per es. non si deve chiedere all'utente di inserire l'identificativo interno del database ("id") per selezionare un utente o altro oggetto. Per evitare ciò, un'alternativa può essere una combo box o casella di selezione con valori pre-riempiti tra quelli possibili (es. nome, cognome, email, tipo, ecc.). Ovviamente, internamente, le singole selezioni saranno associate ad un identificativo univoco.
- Se il numero è, semanticamente, una stringa, il campo NON deve essere numerico (NO type=number). Esempi: numero di carta di credito, numero di telefono, codice prodotto.
- Precisare sempre l'unità di misura di quello che si chiede, per es. Durata: scrivendo "1" cosa si intende? 1h , 1min? In questo caso può andare bene anche usare il campo numerico HTML5 (type=number), ma si deve anche poter scriverci direttamente dentro senza usare le freccette per selezionare il numero. Se devo scrivere 300 non deve essere necessario fare 300 click o attendere che si girino tutti i numeri.
- Campi data/ora: evitare di farli di solo testo senza esempi: come scrivo se non c'è un esempio? (Uso /, - o che altro simbolo tra anno, mese e giorno? e in che ordine, italiano o inglese? Il mese è in numero o lettere?) In generale preferire il type=date o usare librerie per avere un box che si apre che consente la selezione della data. Come minimo è necessario avere un esempio su come scrivere, dentro o di fianco al campo, es. YYYY/MM/DD.
- NON si deve fare il controllo del contenuto della password in fase di login ma solo in fase di creazione di una nuova password (se il testo prevede di crearle). Controllare in fase di login è generalmente scorretto, si danno informazioni su come sono fatte le password ad un eventuale attaccante (es. minimo 6 caratteri, almeno una maiuscola, ecc.), oltre ad essere scomodo in fase di valutazione dell'applicazione perché non consente di usare password qualsiasi per testare login falliti.
- E' comodo che i form facciano partire la sottomissione anche con l'INVIO, non solo con il click sul bottone (si risparmia tempo nel test ed è anche buona norma, es. login). Per fare ciò è sufficiente gestire l'evento onSubmit sul tag form.
- NON usare le funzioni di creazione di finestre del browser, MAI (per es. alert(), prompt(), confirm()). Problemi: non sono integrate con la gestione dell'applicazione di React, la visualizzazione è potenzialmente differente per ogni browser, possono essere disabilitate nel browser, se sono troppe il browser le ferma, ecc.
- NON usare funzioni per ricaricare la pagina del browser da Javascript (es. window.location.reload()). L'applicazione, dopo essere stata caricata la prima volta, non deve richiedere ricaricamento, altrimenti non è una Single Page Application (SPA). Un tale comportamento sarà fortemente penalizzato nella valutazione.
- Campi di un form: evitare di renderli non editabili per costringere ad usare i bottoni, es. + e - in un campo numerico. Non deve essere necessario fare 1000 click per scrivere 1000…
- Il testo nei campi di testo è libero e in generale deve poter supportare spazi, e soprattutto non fallire senza chiarire l'errore se c'è uno spazio (che può essere invisibile e/o inserito per errore).
- Quando un'operazione è stata eseguita, si dovrebbe dare un feedback all'utilizzatore: o cambia

qualcosa nell'interfaccia (aggiunta/rimozione di un elemento nell'interfaccia), o si mostra un messaggio di conferma/esito. In particolare, in caso di errore, si dovrebbe capire cosa non va. (La gestione di tutti i possibili errori non è semplice, ma si può raggiungere un buon compromesso, soprattutto seguendo le regole sopra si minimizzano i potenziali problemi)

- Usare il log degli errori tramite console.log(), sulla console del browser, solamente per lo sviluppo. Il progetto consegnato non dovrebbe usarne per gestire condizioni che si possono verificare durante l'esecuzione dell'applicazione. Eventuali errori dovrebbero sempre essere mostrati all'utente (per es. definendo uno stato e una funzione handleError che lo imposti).
- Provare l'applicazione anche quando il server delle API non è attivo. Spesso durante i test il server non funziona per vari motivi (es. manca un pacchetto npm, il nome del file del DB è scritto in maniera non corretta a causa di maiuscole/minuscole, il server si è piantato).

NAVIGAZIONE DELL'APPLICAZIONE

- Mettere gli eventuali bottoni di navigazione BEN IN VISTA, non nascosti in qualche menu a scomparsa (es. icona utente): diventa una perdita di tempo per chi testa/usa l'applicazione, rischiando che sia considerata una funzionalità mancante nel progetto.
- Se c'è una legenda dei simboli (simboli che in certe applicazioni non sono intuitivi) deve essere sempre visibile o almeno accessibile senza cambiare vista da tutte le schermate dell'applicazione dove i simboli compaiono.
- Deve sempre esserci un bottone per navigare indietro o in una vista/condizione nota, per es. "Annulla/Indietro" (preferibile), o almeno "Home", ben chiaro, non in chissà quale punto o sotto chissà quale scritta cliccabile (es. "MyApp" o loghi vari). Se il bottone non è identificabile, non c'è alternativa a usare il Back del browser, che è sconsigliabile come spiegato nel corso.
- Evitare di inserire ritardi artificiali lato server per mostrare la condizione di loading, perché ciò rallenta il test dell'applicazione, o quantomeno contenerli a 100-200 ms massimo. La presenza della gestione del loading sarà comunque rilevata dall'analisi del codice.

BUONE NORME

- Mostrare da qualche parte nell'interfaccia l'informazione se l'utente è autenticato o meno. Se ha un ruolo (studente/amministratore/altro) mostrare anche quello correntemente selezionato.
- Scrivere sempre l'intestazione delle tabelle e liste: cosa c'è nella prima, nella seconda colonna ecc...? Talvolta è intuitivo, ma molte volte non lo è, in particolare nei tests con dati di prova.
- A livello grafico: evitare effetti particolari (ombre di riquadri ecc...) che rischiano di non essere ben testati e andare in crisi quando le liste si allungano o ci sono tanti elementi. La bella grafica non è richiesta né valutata. I template di default (di bootstrap o altre librerie simili) sono già sufficientemente belli.
- Non esagerare nell'uso della disabilitazione dei warning sulla useEffect (per es. con il commento `//eslint-disable-line`). In alcuni casi può essere necessario e/o comodo, ma non dev'essere la norma. In generale, si dovrebbe evitare che si verifichi il warning evitando o aggiungendo la dipendenza. Per le funzioni pure, per evitare il warning, spesso è sufficiente definire le variabili all'interno della callback passata alla useEffect e non fuori. Nel caso di altre funzioni ciò può essere inevitabile.

CONSEGNA (alcuni problemi portano a perdite di punti o all'impossibilità di valutazione)

- Il tag da usare è "`final`" (scritto minuscolo, senza le virgolette, senza spazi), applicato al commit da valutare, che deve trovarsi nel branch main. Si consiglia di controllare che tutto sia a posto verificando la presenza del tag dall'interfaccia web di GitHub. Se il tag non è presente in forma corretta la consegna non sarà considerata, essendo indistinguibile dal caso in cui non si voglia consegnare. Si ricorda che la data/ora dei commit, per come funziona git, è generata localmente sul PC prima del "push", quindi tale informazione non dimostra nulla riguardo all'avvenuta consegna nei tempi.
- Testare su sistema **CASE SENSITIVE**. MacOS e Windows NON LO SONO. MacOS sembra case sensitive ma NON lo è, il file system ricorda il case dei nomi ma i files si aprono con qualunque case. Attenzione in particolare al file del db che non si legge a causa di maiuscole/minuscole, e che spesso non visualizza un errore sul lato client dell'applicazione se la gestione errori non l'ha previsto. Attenzione a import/require. Se il nome del file del DB è errato, l'applicazione in genere non funziona ed è difficile capire il perché (spesso il messaggio è "internal server error", che non aiuta).
- Ricordare di includere tutti i pacchetti nel `package.json`, sia per il server, sia per il client, altrimenti l'applicazione non è testabile. In altre parole, non installare mai alcun pacchetto a livello globale (tranne nodemon, che comunque non sarà usato in fase di test o sarà già installato a livello globale).
- Essere disordinati nel codice non è "una propria scelta". L'ordine fa parte della valutazione. Non consegnare un file unico (o quasi) con tutti i componenti dentro, ma neanche esagerare all'opposto: raggruppare nello stesso file componenti per funzionalità logiche va più che bene (es. amministratore/utente, o ruolo nell'interfaccia - es. barre di navigazione/menu). Come regola approssimativa, files da 500+ linee di codice sono indice di cattiva organizzazione.
- CLONARE il repository per l'esame e NON modificare la struttura delle cartelle, non rinominare e non spostare files e directories, in particolare "client", "server", e README.md
- Al fine dell'esame, usare password degli utenti tutte uguali e la stessa password (ma con sale differente) per molti/tutti semplificano di molto la verifica manuale dell'applicazione (es. usare valori semplici come "password", o "pwd"). NON vengono tolti punti all'esame per questo, anzi è apprezzato, così come lasciare un utente e password di default già pre-compilati nel form di login (cosa facilmente realizzabile inizializzando lo stato del componente del form, come suggerito a lezione).
- Al fine dell'esame, usare username semplici e corti da scrivere, e regolari, soprattutto email (es. u1@p.it, u2@p.it, u3@p.it), è molto apprezzato. In generale, ogni due/tre minuti in più spesi per testare ognuna delle 100+ consegne al primo appello comporta potenzialmente l'allungamento di un giorno degli esami orali.

README.md: come scriverlo

- Riportare il proprio vero nome cognome e matricola al posto della dicitura generica Student: s123456 LASTNAME FIRSTNAME! Lasciare `Student:` (in inglese) che può essere comodo per processare automaticamente i files README.md
- Per il corso in italiano (Applicazioni Web I), ad eccezione delle prime 2 righe (Exam e Student) il resto può essere scritto sia in italiano sia in inglese, ciò non comporta penalizzazioni né aggiunta di punti. Per comodità, evitare di modificare i titoletti già inseriti nel README.md di esempio (`Application Routes, API Server`, ecc.)
- Verificare ATTENTAMENTE username/password indicate nel file README.md. Se per esempio si dimentica il dominio dell'email, o la password, l'applicazione non è testabile e non sarà valutata. E' inutile e indice di scarsa comprensione della problematica riportare l'hash della password nel README.md

- Controllare che gli screenshot inclusi nel README.md siano collegati correttamente (percorso e nome file con il case giusto). Si verifica facilmente con la funzionalità PREVIEW di VSCode sui file .md
- ".md" è un formato ben definito, sebbene semplice. Non è solo un modo ordinato di scrivere file di testo. Se non si rispetta la sintassi, non funziona. Non è sufficiente includere immagini/screenshots in una cartella qualsiasi. Verificare se è tutto ok tramite il PREVIEW di VSCode. Esistono anche degli editor per scrivere più facilmente il file: cercare "markdown editor" sui motori di ricerca. Alcuni sono anche online, è sufficiente fare copia/incolla della parte testuale risultante.
- Per le tabelle del DB, NON riportare l'SQL di creazione: è di difficile leggibilità. Mettere i nomi delle colonne (possibilmente usare nomi significativi), e tra parentesi le informazioni importanti, es. chiave primaria (se non evidente) o campi unique.
- Per le API: se si includono esempi di richiesta/risposta, mettere esempi CORTI, evitando liste di N oggetti con M campi l'uno. Valutare se ridurre il numero di "a capo" nel JSON per migliorare la leggibilità del README.
- Ricordarsi di specificare il ruolo degli utenti inseriti, amministratore, gestore, utente, creatore, ecc., oppure utente di tipo A, B, C, ecc. (se applicabile, secondo la traccia d'esame).