Errori frequenti nel progetto d'esame
Applicazioni Web I – A-L, a.a. 2020/21

Consigli alla luce dell'esperienza degli anni passati
v. 1.0

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. In generale però riuscire a soddisfare questi punti dovrebbe garantire, in media, una valutazione molto buona perché si evitano i problemi più gravi.

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

- NON creare ID sul client quando devono essere identificativi (chiavi uniche) nel DB: gli ID DEVONO essere creati/calcolati 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 è in generale non direttamente visibile, selezionabile o manipolabile dall'utente, quindi il suo valore dovrebbe impattare sulla logica o interfaccia dell'applicazione.
- Usare sempre il metodo corretto nello scrivere e chiamare le API HTTP (mai POST al posto di GET, e viceversa)
- E' necessario verificare sempre 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 abbia il permesso di fare la cosa (per es. da deserializeUser, o con apposita query su db).
- 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. Chi afferma: "sul client ho controllato che non si possa fare X..." non ha compreso la problematica, e non potrà avere il massimo. La verifica dei valori 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 informazioni/dati che sono facilmente gestibili da una libreria: si perde tempo, è molto probabile sbagliarsi, e all'esame sfortuna vuole che si testa sempre il caso non considerato. Per esempio: verifica correttezza data. Un programmatore 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 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. Se per esempio 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 dato strano 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 codice aggiuntivo questi casi). In generale, i campi di testo di un form 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. C'è il JSON per serializzare/deserializzare informazioni contenute nelle stringhe Javascript e viene ampiamente 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. Ovviamente usare useEffect in tutti i casi in cui è effettivamente necessaria (caricamento asincrono di dati ecc.)

PROBLEMI DI INTERFACCIA, se gravi portano a perdite di punti, di sicuro innervosiscono

- Se l'utente non ha ancora fatto nulla, non deve apparire un form che mostra già errori (es. campi bordati di rosso): per es. login con username e password rossi perché vuoti, appena arrivati sulla pagina. Solo dopo la pressione del bottone login (o Invio) devono venir fuori gli eventuali errori (es. è richiesta un'email, manca @, ecc.).
- testare TUTTI i modi di usare i campi, in particolare quelli HTML5: se scrivo il numero anziché usare le freccette del campo (es. campo "numero crediti di un corso"), deve funzionare, altrimenti l'applicazione rischia di perdere punti perché certe funzionalità non si riescono a testare. Attenzione anche ai numeri: se non specificato, non è detto che siano interi.
- NON fare assunzioni strane o errate sui dati: un codice identificativo, per esempio, NON è necessariamente numerico. L'ID interno può essere numerico, ma all'utilizzatore dell'applicazione NON interessa e non deve essere utilizzato nell'interfaccia. I codici corso non sono numerici (come peraltro al Politecnico). Il codice di un prodotto NON è numerico. E' concettualmente sbagliato implementarlo con campi type=number
- NON fare assunzioni inutili, non richieste o sbagliate sui dati di input: es. almeno 6 caratteri per il nome del corso (4 caratteri non bastano? es. "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): e se è Roma, non va bene?
- 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 (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, non tutto il mondo è come il Politecnico), ma libero, eventualmente anche con la virgola. Nel dubbio, chiedere sugli appositi canali per 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’ignaro utilizzatore l'ID interno oppure mostrando solo l’ID come identificativo di scelta (come fa l’utilizzatore a sapere a cosa corrisponde l'ID?)

- 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 vuol dire? 1h , 1min? In questo caso può andare bene anche usare il campo numerico HTML5, ma si deve anche poter scriverci direttamente dentro senza usare le freccette per selezionare il numero. Se devo scrivere 300 non devo 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 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. Il minimo è 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 (se richiesto) in fase di creazione di una nuova password. Controllare in fase di login è generalmente scorretto, si danno solo informazioni su come sono fatte le password (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.
- I form devono far 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)
- NON usare gli alert() del browser, MAI. Problemi: visualizzazione differente per ogni browser, si possono disabilitare, se sono troppi il browser li ferma, ecc.
- Campi di un form: evitare di renderli non editabili per costringere ad usare i bottoni, es. + e - in un campo numerico. Se devo scrivere 1000 devo fare 1000 click?
- 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 spesso è invisibile e inserito per errore).
- Quando un'operazione si è svolta, 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 errori non è semplice, ma si può raggiungere un buon compromesso, soprattutto seguendo le regole sopra si minimizzano i potenziali problemi)

NAVIGAZIONE DELL'APPLICAZIONE

- Mettere i 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 (che in certe applicazioni non sono intuitivi) deve essere sempre visibile o almeno accessibile senza cambiare vista da tutte le viste 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 "MyApp" o loghi. Se non si identifica il bottone, non c'è alternativa a usare il Back del browser, che è sconsigliabile come spiegato nel corso.

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 all'inizio, colonna in mezzo ecc...? Ogni tanto è intuitivo, ma spesso 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. I template di default (di bootstrap o altre librerie simili) sono già sufficientemente belli.

CONSEGNA (alcuni problemi portano a perdite di punti)

- Testare su sistema **CASE SENSITIVE**. MacOS e Windows non lo sono. MacOS sembra case sensitive ma NON lo è, mantiene 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 produce un errore sul lato client dell'applicazione se non ben gestito. 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).
- Essere disordinati nel codice non è "una propria scelta". L'ordine è 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). In generale, file da 500+ linee sono indice di cattiva organizzazione.
- CLONARE il repository per l'esame e NON modificare la struttura delle cartelle, non rinominare, non spostare, in particolare "client", "server", e README.md
- Al fine dell'esame, usare password degli utenti tutte uguali e la stessa password per molti/tutti semplificano di molto la verifica manuale (es. usare "password", o "pwd"). NON si tolgono 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, 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 minuto/due in più speso per testare ognuna delle 100+ consegne al primo appello comporta potenzialmente lo slittamento 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!
- Verificare ATTENTAMENTE username/password segnalate nel file README. Se si dimentica il dominio dell'email, o la password, l'applicazione non è testabile e non sarà valutata.
- Controllare che gli screenshot inclusi nel README.md siano collegati correttamente (percorso e nome file con il case giusto). Si verifica facilmente con PREVIEW di VSCode.
- "md" è un formato ben definito, sebbene particolarmente semplice. Non è solo un modo carino 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.

- 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.
- Le API: se si includono esempi di richiesta/risposta, farli CORTI, evitando liste di N oggetti con M campi l'uno). Valutare se ridurre il numero di a capo nel JSON.
- Ricordarsi di specificare il ruolo degli utenti inseriti, amministratore, gestore, utente, creatore, ecc. (secondo la traccia d'esame, se applicabile).