Instant Developer Team Works

Indice generale

Perché non basta GitHub? **2**
Il modello relazionale 2
Vantaggi del modello relazionale 4
Dal modello GitHub a Team Works 6

Team Works: concetti base **6**
Working copy 6
Commit 6
Fork 6
Branch 7
Merge 7
Pull request 7
Fetch 7
Reset 7
Revert 7
Conflitto 8
Visualizzazione differenze 8

Organizzazione del lavoro consigliata **8**
La copia master 8
I fork 9
Conferma del lavoro 9
Esecuzione di un merge o fetch 9
Fork relativo a rilasci in produzione 10
Modifiche dei file di risorse 10

Risoluzione dei problemi relativi a Team Works **11**
Codice errato dopo una merge 11
Codice duplicato dopo una merge 12
Errori javascript dopo aver effettuato un’operazione di team working 12
Progetti in stato invalido o danneggiato 12
Ripristino completo dello stato del team working 13

Domande sull’utilizzo di Team Works **13**

Instant Developer Team Works

Questo documento descrive il funzionamento del modulo Team Works di Instant Developer Cloud. Data la natura particolarmente delicata delle operazioni che esso consente, è necessario conoscere il contenuto di questo documento prima di iniziare ad utilizzare le relative funzioni di gestione.

Perché non basta GitHub?

La prima domanda a cui occorre dare risposta è perché Instant Developer Cloud non utilizza un sistema di gestione del lavoro di gruppo standard come GitHub e invece ne include uno proprietario?

Per comprendere la risposta a questa domanda occorre prima analizzare che cosa è un progetto Instant Developer Cloud e come esso viene modificato durante il lavoro dei programmatori.

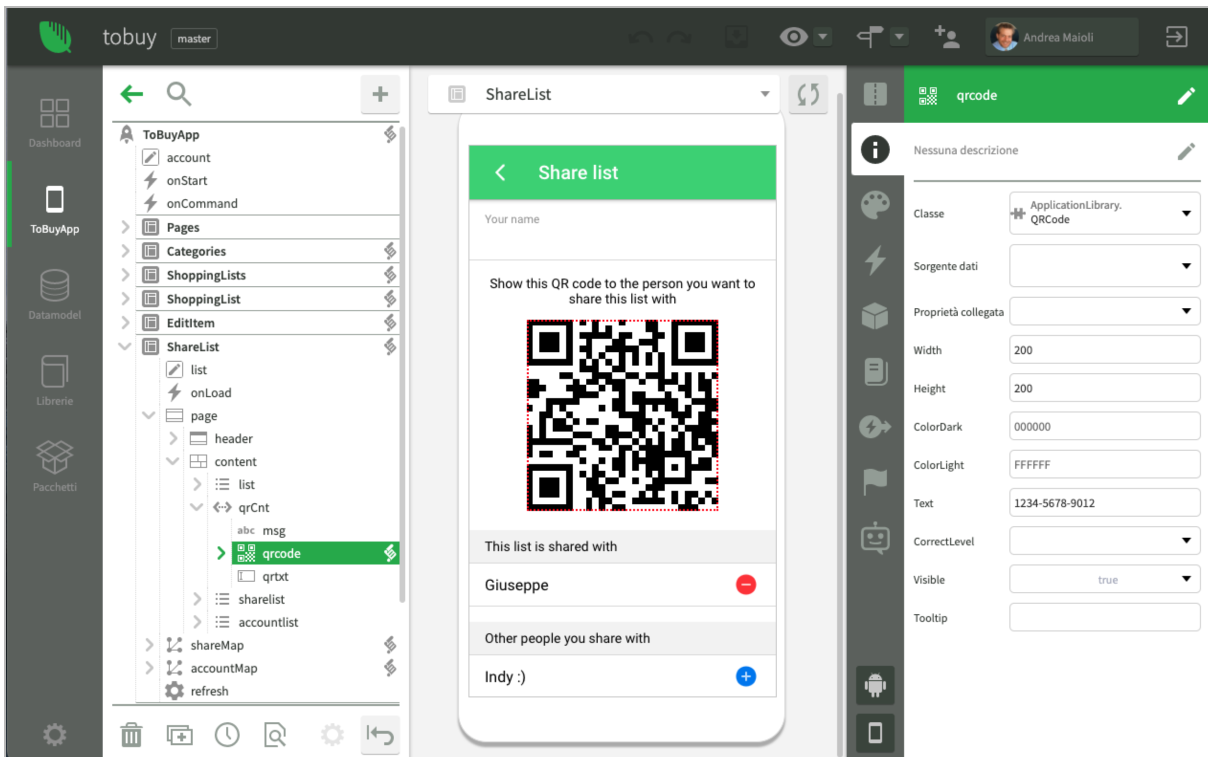
Solitamente un progetto software è un insieme, anche molto corposo, di file di testo che ne contengono il codice nei vari linguaggi necessari a descriverne il funzionamento. In questi casi GitHub è sicuramente un buon sistema di gestione del lavoro di gruppo, inteso come gestione delle modifiche che vengono apportate ad uno o più file di testo.

Il modello relazionale

Il modo con cui Instant Developer Cloud memorizza i dati di un progetto software non si basa sui file di testo. Viene invece utilizzato un grafo memorizzato in un unico -grande- oggetto javascript che contiene come nodi tutti gli oggetti definiti nel progetto, e come archi le relazioni fra di essi.

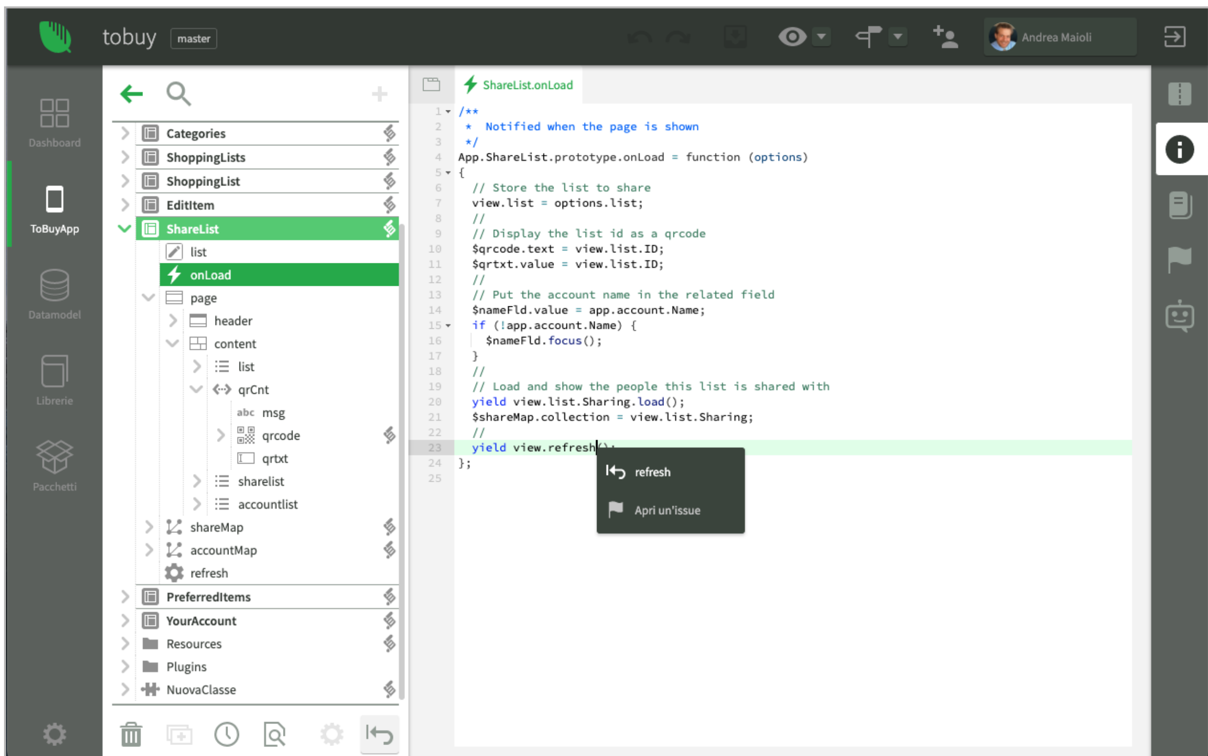
Quali sono le parti di progetto memorizzate nei nodi di questo grafo? In sostanza tutto ciò che è presente, che in qualche modo è stato definito. Facciamo alcuni esempi:

- Le videate dell'applicazione
- Gli elementi grafici che compongono le videate.
- Gli script che gestiscono gli eventi degli elementi grafici.
- Le classi.
- Le proprietà ed i metodi delle classi.
- I database, le tabelle, i campi, le relazioni e gli indici.
- I documenti e i relativi metodi, proprietà ed eventi.
- Le librerie di sistema.
- I pacchetti.
- Le risorse (immagini, video, file…).



L'albero del progetto mostra una parte dei nodi del grafo che descrive il sistema in fase di sviluppo

Oltre a tutti questi elementi, anche tutto il codice presente in ogni script dell'applicazione viene memorizzato come parte del grafo. Quando selezioniamo nell'albero degli oggetti uno script o un metodo, l'editor di codice ricostruisce una versione testuale della parte di grafo dedicata al metodo e lo mostra in un editor di codice javascript.



Anche il codice è parte del grafo del progetto

Quando il codice viene modificato, il grafo sottostante viene ricostruito in tempo reale, in modo tale che tutto quello che viene scritto nell'IDE rimane sempre in forma relazionata.

Nell'immagine precedente possiamo vedere che all'interno del nodo *onLoad*, che rappresenta lo script lanciato all'apertura di una istanza della videata *ShareList*, vengono memorizzati i seguenti tipi di nodi:

- I parametri del metodo, ognuno con le proprie caratteristiche.
- Le righe di codice del metodo
- Per ogni riga, ogni token o parola chiave del linguaggio.

Ogni nodo del grafo ha una serie di caratteristiche che dipendono dal tipo di nodo stesso. Ad esempio, un elemento grafico memorizza tutte le proprietà dell'elemento stesso, che a sua volta dipendono dal tipo di elemento in funzione di come esso è stato definito nelle librerie.

I nodi che rappresentano le righe e i token di codice portano importantissime informazioni di collegamento con gli oggetti presenti nel progetto. In questo modo, infatti è possibile far funzionare gli algoritmi di type inference e di gestione delle referenze. Come esempio di questo possiamo notare nell'immagine precedente che aprendo un menu contestuale a partire dal token *refresh* appare una voce di menu che permette di saltare alla definizione del metodo nella videata. Questa operazione non avviene tramite matching di stringhe, ma proprio come referenza diretta del token al nodo *metodo refresh* nell'albero degli oggetti.

Vantaggi del modello relazionale

La modalità di memorizzazione relazionale permette alcune delle caratteristiche più interessanti e utili di Instant Developer Cloud fra le quali:

1) *Instant refactoring*: modificando il nome o le caratteristiche di un elemento del progetto, le conseguenze di questa modifica vengono apportate automaticamente in tutto il codice presente nel progetto. Se, ad esempio, si cambia il nome al metodo *refresh* in *aggiorna*, anche le righe di codice che lo chiamavano vengono cambiate di conseguenza. Tali modifiche vengono apportate in automatico perchè sono sicure, non richiedono alcuna revisione manuale.

2) *Type inference*: Instant Developer Cloud è in grado di recuperare in tempo reale i tipi riferiti dalla maggior parte delle espressioni javascript, anche se questo linguaggio, per sua natura, non ha alcuna informazione sui tipi a design time. In questo modo è possibile proporre un *intellisense* globale esteso all'intero progetto: dal database, alle librerie di componenti, fino ad ogni elemento di back-end e front-end.

```
app.account.n
;
              Name  TBBE.Account
              notifyDocUpdate  ApplicationLibrary.I
              onInserting  TBBE.Account
              onGetTopics  TBBE.Account
              onResyncClient  TBBE.Account
              onDocUpdate  TBBE.Account
              inserted  ApplicationLibrary.Document
              index  ApplicationLibrary.Document
```

Intellisense globale sull'intero progetto

3) *Sequenzializzatore asincrono*: una delle difficoltà maggiori della scrittura di applicazioni javascript complesse risiede nella gestione delle operazioni asincrone, che, per loro natura, richiedono di descrivere il flusso di un processo complesso tramite callback, come mostrato nell'esempio seguente.

```
fs.readdir(source, function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height + 'x' + height)
            this.resize(width, height).write(dest + 'w' + width + '_' + filename, function(err) {
              if (err) console.log('Error writing file: ' + err)
            })
          }.bind(this))
        }
      })
    })
  }
})
```

L'inferno delle callback

Questo modo di scrivere il codice viene spesso chiamato l'inferno delle callback perché rende estremamente difficile riconoscere il significato del codice, oltre che essere altamente imprevedibile per quanto riguarda la gestione delle eccezioni. Cosa accade infatti in caso di eccezione javascript? Quale sarà la prossima riga di codice ad essere eseguita?

Fin dal 2015 Instant Developer Cloud contiene un framework per la gestione automatica di questo problema. Tale framework converte il codice asincrono in un codice sequenzializzato tramite *yield*. Le righe di codice che coinvolgono operazioni asincrone vengono quindi *sospese* dal costrutto *yield* e poi riprese in automatico al termine dell'operazione asincrona. Anche la gestione delle eccezioni può essere scritta in modo normale, senza preoccuparsi del livello di asincronicità in cui esse avvengono.

Per poter funzionare, l'algoritmo di sequenzializzazione asincrona ha bisogno di informazioni dettagliate sui tipi di oggetti utilizzati nel codice, in quanto il codice stesso deve cambiare di forma se vengono coinvolte o meno operazioni asincrone. Quindi un metodo che inizialmente non è asincrono e poi lo diventa in seguito, causa modifiche automatiche anche in tutti i punti in cui tale metodo è stato utilizzato.

4) *Gestione globale delle referenze*: quando il progetto software diventa complesso, è necessario poter sapere dove un determinato oggetto viene utilizzato. Ad esempio potrei essere interessato a conoscere tutti gli utilizzi di un determinato campo (magari di nome "id") di una tabella del database. Instant Developer Cloud è in grado di estrarre queste informazioni in modo rapido e sicuro navigando il grafo dei nodi del progetto.

Dal modello GitHub a Team Works

È più facile ora spiegare perché non è possibile utilizzare GitHub per la gestione del versioning di progetti Instant Developer Cloud: GitHub è in grado di gestire modifiche a tanti file di testo, un progetto Instant Developer Cloud è invece contenuto in un unico file json.

Quando con GitHub si effettuano modifiche multiple allo stesso file si ottengono facilmente conflitti che devono essere risolti manualmente. Se si utilizzasse GitHub per il versioning dell'unico file json che contiene il progetto Instant Developer Cloud questi conflitti sarebbero all'ordine del giorno e non sarebbero facili da risolvere perché il formato json non è facilmente investigabile da parte di un occhio umano.

È stato quindi necessario implementare un algoritmo di versionamento specificatamente pensato per gestire le variazioni di un grafo relazionale. Il sistema di gestione del lavoro di gruppo basato su questo algoritmo è proprio Instant Developer Team Works.

Come funziona in pratica Team Works? Ad ogni *commit* che avviene in un progetto Instant Developer Cloud, Team Works calcola un file *delta*, che rappresenta l'insieme minimo di trasformazioni che devono essere apportate alla versione precedente del grafo per arrivare a quella attuale. I file *delta* possono essere poi trasportati fra i vari *branch* o *fork* del progetto per essere riprodotti e aggiornare anche altre versioni del grafo che possono avere anche linee temporali diverse.

Nei paragrafi che seguono verranno illustrate le varie modalità di funzionamento di Team Works in modo da comprendere come avvengono le varie operazioni al fine di poter gestire in sicurezza il proprio lavoro in gruppo.

Team Works: concetti base

Working copy

È la copia del progetto su cui si sta lavorando. Può essere la *copia master* se è quella utilizzata come base da cui effettuare i fork, oppure una *copia derivata* se, appunto, è stata ottenuta come *fork* da una versione master.

Commit

Una unità di lavoro confermata, espressa come l'insieme minimo di modifiche necessarie per passare dalla versione confermata attuale alla versione confermata precedente.

Fork

Creazione di una copia di un progetto, solitamente nell'account di un altro programmatore. La versione copiata mantiene il legame con la versione master, potendo inviare i propri commit o ricevere le modifiche presenti nella versione master. Il fork contiene solo il branch master confermato, non contiene quindi la parte di lavoro non ancora confermata della copia master.

Branch

Una versione alternativa sempre contenuta all'interno della propria working copy. Per ogni working copy esiste sempre almeno un branch (quello master). È possibile creare nuovi branch, utili quando si desidera implementare parti complesse che non possono essere confermate fino al termine del lavoro e, al contempo, potrebbe essere necessario proseguire il lavoro anche nella versione che non contiene tali modifiche.

A differenza di GitHub, con Team Works non è necessario sempre creare nuovi branch all'inizio di una unità di lavoro, lo si fa solo in casi specifici, come nell'esempio precedente.

Merge

L'operazione di unione dei commit effettuati su un determinato branch o fork all'interno di un altro branch, solitamente quello master. Se, ad esempio, si è concluso con successo il lavoro effettuato su uno specifico branch è opportuno renderlo disponibile a tutti effettuando il merge del branch all'interno di quello master e poi cancellando il branch specifico dopo il controllo del risultato del merge.

Pull request

L'operazione di invio dei commit effettuati in un fork verso la versione master. Tali commit verranno poi uniti all'intero della versione master con una operazione di merge.

Fetch

L'operazione di recupero di nuovi commit presenti nella versione master a partire da un fork. In questo modo il fork si allinea alla versione master ed è possibile proseguire con una nuova unità di lavoro.

L'operazione di fetch esegue una *rebase* del fork su cui viene eseguita. Cioè, se il fork su cui viene eseguita la fetch non è allineato al master, per prima cose vengono annullate le modifiche presenti nel fork, poi vengono acquisite le modifiche arrivate dal master ed infine vengono nuovamente eseguite le modifiche annullate in precedenza. Se tali modifiche non possono essere eseguite a causa delle variazioni ottenute dal master, si possono ottenere dei conflitti al termine della fase di fetch. Per evitare questa situazione, prima di eseguire una fetch si consiglia di inviare al master le modifiche presenti nel fork tramite una pull-request, accettare la pull-request ed infine allinearsi al master.

Reset

L'operazione di annullamento di tutte le modifiche non confermate nel branch in fase di modifica. In pratica si ritorna all'ultima versione confermata senza poter più recuperare le modifiche annullate, a meno di non ripristinare un backup dell'intero progetto.

Revert

L'operazione di annullamento delle modifiche apportate ad un progetto da parte di un determinato commit. Essa avviene creando un nuovo commit che contiene una versione

invertita del commit precedente. Al posto della *revert* è possibile ripristinare un backup del progetto prima del merge che ha causato problemi.

Conflitto

Durante le operazioni di merge e di fetch possono avvenire conflitti se uno stesso oggetto è stato modificato da entrambi i lati, cioè all'interno dei commit in fase di merge e anche nella versione in cui si sta facendo il merge.

Durante l'operazione sarà necessario risolvere il conflitto, cioè scegliere quale versione mantenere: quella attuale, quella che proviene dai commit in fase di merge, o anche entrambe in alcune occasioni. Dopo aver risolto tutti i conflitti sarà inoltre necessario verificare se occorre correggere manualmente il progetto per completare l'integrazione dei nuovi commit con lo stato precedente. È quindi molto importante lavorare in modo da evitare il più possibile i conflitti, modificando le stesse parti di progetto.

Visualizzazione differenze

Prima di effettuare le operazioni di gestione del progetto (commit, merge, fetch) è necessario valutare l'impatto delle stesse tramite le funzioni di visualizzazione differenze che Team Works include. Solo dopo aver osservato e valutato la bontà del codice che si sta confermando o unendo alla versione attuale sarà opportuno continuare con l'operazione vera e propria.

Organizzazione del lavoro consigliata

La copia master

È la copia di progetto in cui verranno consolidate tutte le modifiche, sia quelle effettuate in locale che quelle in arrivo dai fork. Solitamente la copia master è gestita dal responsabile delle release, una persona in grado di valutare la bontà del codice proprio e di quello dei collaboratori.

È certamente possibile lavorare direttamente sulla copia master, che così non deve essere una versione staccata dalle altre. Per lavorare sulla copia master è consigliabile:

- Quando si devono effettuare modifiche semplici che vengono confermate all'interno della singola giornata di lavoro, è possibile farle direttamente sul branch master.
- Se si intendono effettuare modifiche più complesse, è consigliabile usare un branch specifico. Quando la modifica è conclusa bisognerà effettuare il merge di tale branch sul master.
- Prima di effettuare qualunque operazione di commit è necessario controllare le modifiche tramite la visualizzazione delle differenze.
- Prima di effettuare qualunque operazione di merge di branch o di pull request è necessario controllare le modifiche che verranno apportate tramite il pulsante *Mostra differenze (show incoming)*.

I fork

Sono le copie derivate dal master da sviluppatori che collaborano al progetto insieme con il responsabile delle release. Quando si effettua un fork, verrà copiata solo la versione confermata del branch master della copia master. È quindi possibile effettuare un fork solo dopo aver eseguito almeno un commit sulla copia master.

Per lavorare sui fork è consigliabile seguire queste regole:

- Prima di iniziare una unità di lavoro, effettuare l'operazione di *fetch* per allinearsi al master. Siccome non esiste alcun commit nel fork che non sia stato già comunicato al master in precedenza, l'operazione di fetch non può evidenziare alcun conflitto. In caso di nascita di conflitti, vedere il paragrafo *Risoluzione dei problemi*.
- Eseguire il lavoro e confermare spesso. Si consiglia di non lasciare passare più di una giornata lavorativa senza effettuare un commit.
- Prima di effettuare qualunque operazione di commit è necessario controllare le modifiche tramite la visualizzazione delle differenze.
- Quando le modifiche sono terminate con successo sarà possibile inviarle al master tramite il pulsante *Crea Pull Request*. Il gestore del progetto master verrà avvisato automaticamente via mail.
- Prima di ripartire con una nuova unità di lavoro, effettuare nuovamente l'operazione di fetch. Occorre tenere presente che, se la pull request precedente non è ancora stata accettata, è possibile che vengano evidenziati conflitti non ancora risolti.

Conferma del lavoro

Ogni volta che il lavoro attuale può essere confermato è consigliabile effettuare un commit, l'ideale è quello di confermare il lavoro fatto entro la giornata di lavoro.

Prima di effettuare il commit è necessario verificare le differenze per essere coscienti di quello che si sta marcando come lavoro concluso. In questa fase è consigliabile:

- Eliminare il codice commentato.
- Eliminare i console.log usati per il debug.
- Aggiungere eventuali commenti per rendere chiaro il codice agli altri sviluppatori (o a se stessi dopo qualche tempo).
- Controllare di non aver inserito metodi troppo complessi, eventualmente rendendo il codice più modulare.
- Verificare di non aver violato regole costruttive o di stile.

Esecuzione di un merge o fetch

L'operazione di merge avviene nella copia master al momento dell'accettazione di una pull request, oppure in tutte le copie quando si tratta di unire un branch a quello master. L'operazione di fetch è simile; la sola differenza è che viene utilizzata in un fork per recuperare i nuovi commit.

L'operazione di merge deve avvenire come segue:

- Se il codice che si intende accettare è complesso o "delicato", si consiglia di effettuare un backup della copia master prima di iniziare il merge.

- Si inizia l'operazione di merge tramite il pulsante *Pull Request* o *Merge* dalla dashboard.
- Si verificano le modifiche in entrata tramite il pulsante *Mostra differenze (show incoming)*.
- Se le modifiche sono accettabili, si procede accettando la *Pull Request*, altrimenti si rifiuta la *Pull Request*.
- Se l'operazione di merge genera dei conflitti, si aprirà la pagina di gestione conflitti in cui dovrai decidere se mantenere le proprie modifiche, quelle della controparte oppure entrambe. Dopo aver risolto tutti i conflitti il progetto viene salvato e lo stato del branch confermato (commit).
- A questo punto è necessario controllare la correttezza del progetto effettuando almeno una operazione di compilazione al fine di controllare che:
    - Il codice del progetto sia corretto
    - Non siano presenti metodi duplicati
    - Non siano presenti errori introdotti dall'operazione di merge.

Per quanto riguarda l'operazione di fetch, come già indicato in precedenza, si consiglia di effettuarla dopo aver inviato il proprio lavoro tramite una pull request.

Fork relativo a rilasci in produzione

Al momento dell'installazione in produzione può essere comodo effettuare un fork del progetto in modo da avere una versione identica a quella rilasciata. In questo modo se si devono effettuare delle correzioni sulla versione rilasciata, sarà possibile farle nel fork relativo.

Se tali modifiche devono poi essere riportate nel master, basterà effettuare una pull request verso il master. È invece vietata qualsiasi operazione di fetch, che renderebbe allineato il fork al master anziché alla versione installata.

Modifiche dei file di risorse

I file di risorse caricati nel progetto tramite upload vengono gestiti da teamworks come un blocco unico. Se essi vengono modificati contemporaneamente su più copie del progetto (master o fork), l'ultimo branch di cui si esegue il merge vince sugli altri: solo l'ultima copia di cui è stato fatto il merge entra a far parte della copia master.

Se quindi viene effettuato l'upload di un file di testo, oppure di tipo javascript client, anche se viene modificato tramite l'IDE, esso verrà sempre gestito come risorsa di tipo file e cioè come blocco unico. In questi casi è necessario organizzare le modifiche al file per essere certi che solo un programmatore alla volta le esegua, altrimenti alcune di esse potrebbero andare perse.

Nessuna descrizione

Tipo: Testo

Ambito:

Url: resources/6c2eed0d-a9c7-4092-94c

File: Browse

Dimensione file: 521

Tipo di file: text/plain

Risorse caricata come file

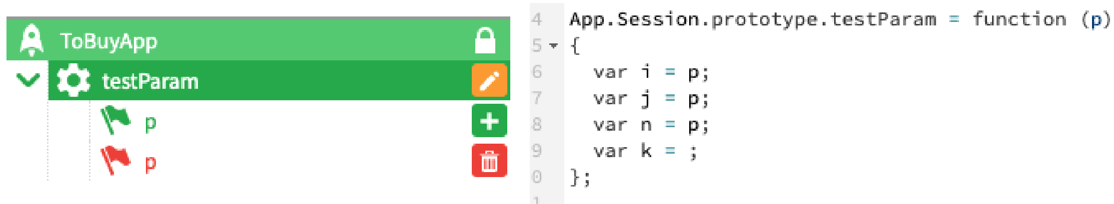
Risoluzione dei problemi relativi a Team Works

Codice errato dopo una merge

Se il codice di un metodo viene modificato sia in un fork che nella copia master, è possibile che al termine dell'operazione di merge il codice risultante non sia valido. Questo avviene anche con GitHub nei casi analoghi.

Come abbiamo visto nei paragrafi precedenti, Instant Developer Cloud ha una funzione di *Instant Refactoring*, che propaga le modifiche effettuate adattando l'intero codice del progetto. È quindi possibile che i metodi vengano modificati anche indirettamente, a seguito di una precedente modifica apparentemente non correlata. È per questo che è molto importante verificare le differenze in fase di commit, in quanto ci si può rendere conto di tutte le modifiche che si stanno apportando al progetto.

Un'altra sorgente di modifiche inattese risiede proprio nella gestione "relazionale" del progetto. Se, ad esempio, viene cancellato un parametro di una funzione, e poi successivamente viene ricreato, esso non mantiene il medesimo valore identificativo interno (guid). Nel progetto in fase di modifica, le referenze al parametro vengono automaticamente convertite al nuovo guid, ma quando viene effettuato il merge di questa modifica su un altro branch, le referenze al vecchio parametro vengono perse.



Cancellando e ricreando un parametro, alcune referenze vengono perse dopo la merge

In questi casi si consiglia di effettuare una correzione manuale del codice dopo la merge, come sarebbe stato necessario anche con GitHub in caso di modifiche multiple alla stessa sezione dello stesso file.

È consigliabile tuttavia risolvere il problema alla radice: se si cancella un parametro per errore, è meglio annullare l'operazione (*ctrl-z*) piuttosto che aggiungerne uno nuovo; in questo caso infatti non si crea alcuna differenza nel progetto.

Codice duplicato dopo una merge

In alcuni casi è possibile ottenere sia codice duplicato all'interno dello stesso metodo sia addirittura metodi o eventi duplicati.

Questo può avvenire per le stesse ragioni viste prima: se, ad esempio, viene aggiunto l'evento *onClick* allo stesso elemento da entrambi i lati, al momento del merge si otterranno due eventi *onClick* per lo stesso elemento. Questa condizione verrà segnalata con un errore al momento della compilazione.

Se si cancella e poi si ricrea un metodo nell'albero, anche se il metodo ha lo stesso nome del precedente è in effetti un oggetto completamente nuovo. Anche in questi casi, l'analisi al momento del commit evidenzia il problema e rende possibile risolverlo prima di trasmetterlo agli altri partecipanti del progetto.

Se infine si modifica lo stesso metodo da entrambi i lati, il sistema che unisce le modifiche alle precedenti può dover mantenere entrambe le righe aggiunte senza sostituirle con le precedenti perché non più esistenti, e in questo caso si può ottenere codice duplicato.

Errori javascript dopo aver effettuato un'operazione di team working

Se il sistema rileva un errore javascript durante un'operazione di team working, la sessione IDE viene automaticamente ricaricata. In tal caso si consiglia di:

- Chiudere e riaprire il progetto dalla console.
- Ritentare l'operazione.
- Se il problema persiste, effettuare un backup del progetto e aprire una segnalazione di errore interno nel sistema di help desk di Instant Developer indicando:
    - Il nome del progetto e il nome del backup.
    - L'errore o l'operazione che si stava tentando di eseguire.
    - L'eventuale situazione di blocco dell'operatività, che rende urgente la segnalazione.

Si segnala che se l'operazione che genera l'errore è una fetch, e il fork non ha codice da inviare al master, è consigliabile effettuare un nuovo fork e ripartire usando quello. In questo modo la situazione che ha causato l'errore viene mantenuta, ma non si rimane bloccati e il lavoro può proseguire.

Progetti in stato invalido o danneggiato

Se per qualunque ragione un progetto risultasse invalido o talmente danneggiato da non poter essere facilmente recuperato, si segnala che il sistema effettua backup notturni

automatici di tutti i progetti modificati il giorno precedente, mantenendo le ultime cinque copie. Oltre ai backup notturni è possibile effettuare un backup manuale tramite la console in qualunque momento.

Se si ripristina un progetto dal backup è necessario sapere che anche l'intero stato di Team Works viene ripristinato. Questo solitamente non ha effetti negativi se il backup ripristinato è recente; potrebbe avere invece effetti imprevedibili se il backup ripristinato è fuori sincronia rispetto allo stato degli altri fork che partecipano al progetto.

Prima di effettuare qualunque operazione di ripristino si consiglia di effettuare un backup del progetto nello stato attuale (invalido o danneggiato) e aprire una segnalazione di errore interno nel sistema di help desk di Instant Developer indicando:

- Il nome del progetto e il nome del backup.
- Lo stato del progetto.
- L'eventuale situazione di blocco dell'operatività, che rende urgente la segnalazione.

Dopo la segnalazione si consiglia di tentare il ripristino di un backup in modo da riprendere il lavoro.

Ripristino completo dello stato del team working

Se si desidera ripristinare lo stato del team working di un progetto, si consiglia di procedere come segue:

1) Consolidare tutte le modifiche dei fork nella copia master.
2) Aggiustare e rendere valida la copia master.
3) Eliminare tutti i fork.
4) Tramite il menu contestuale del progetto nella lista dei progetti della console, inviare il comando *Resetta TW*.
5) Eseguire nuovamente i fork e ripartire.

Domande sull'utilizzo di Team Works

L'esperienza di questi ultimi anni sull'utilizzo di Team Works da parte di decine di gruppi di lavoro ha evidenziato che quando il suo funzionamento è ben compreso il sistema è stabile ed i risultati di utilizzo sono buoni. È quindi essenziale arrivare ad una buona comprensione del comportamento del sistema.

A tal fine vi chiediamo di inviare eventuali domande richieste di chiarimenti relative a Team Works all'indirizzo p.giannelli@instantdeveloper.com o tramite il chatbot presente nell'IDE.