

# **Documentazione progetto di Tecnologie Informatiche per il Web**

*di Alessandro Fiorillo e Filippo Libero*

# **Indice**

Design del database.....	3
Design della navigazione.....	9
Assunzioni.....	13
Scelte implementative.....	15
Algoritmo di scelta delle immagini.....	17

# Design del database

Il database che gestisce l'applicativo è composto da numerose tabelle e viste per tenere conto delle varie relazioni che sussistono tra gli utenti, le campagne, i task e le immagini.

```
CREATE TABLE user (
    first_name varchar(64) NOT NULL,
    last_name varchar(64) NOT NULL,
    username varchar(64) NOT NULL,
    email varchar(64) NOT NULL,
    password varchar(64) NOT NULL,
    role varchar(16) NOT NULL,
    PRIMARY KEY (username),
    UNIQUE KEY username_UNIQUE (username),
    UNIQUE KEY email_UNIQUE (email)
)
```

Le informazioni degli utenti sono contenute nella tabella **user** che possiede i campi per memorizzare il nome, il cognome, lo username, l'indirizzo e-mail, la password di login e il ruolo. Gli utenti sono identificati univocamente attraverso lo **username**, che è chiave primaria della tabella. Tuttavia, siccome essi possono effettuare il login anche con l'indirizzo e-mail, il campo dell'indirizzo e-mail possiede il vincolo di unicità, perché deve identificare univocamente un singolo utente. Il ruolo è rappresentato tramite una stringa di testo che è uguale a “**manager**” per i gestori e “**worker**” per i lavoratori.

```
CREATE TABLE campaign (
    name varchar(64) NOT NULL,
    manager varchar(64) NOT NULL,
    users_for_image_selection int(11) NOT NULL,
    least_positive_ratings int(11) NOT NULL,
    users_for_image_annotation int(11) NOT NULL,
    line_pixels int(11) NOT NULL,
    active tinyint(4) NOT NULL,
    PRIMARY KEY (name),
    KEY manager (manager),
    CONSTRAINT campaign_ibfk_1 FOREIGN KEY (manager) REFERENCES user
    (username) ON DELETE NO ACTION ON UPDATE CASCADE
)
```

Ogni campagna è memorizzata nella tabella **campaign**, i cui campi sono il nome, il manager che la gestisce, i quattro parametri numerici e un valore booleano per indicare se la campagna è stata avviata. La chiave primaria è il **nome**, per cui non possono esistere in tutta l'applicazione due campagne con lo stesso nome, anche se create da gestori diversi.

```

CREATE TABLE worker_campaign (
    worker varchar(64) NOT NULL,
    campaign varchar(64) NOT NULL,
    selection_task tinyint(1) NOT NULL,
    annotation_task tinyint(1) NOT NULL,
    PRIMARY KEY (worker,campaign),
    KEY campaign (campaign),
    CONSTRAINT worker_campaign_ibfk_1 FOREIGN KEY (worker) REFERENCES user (username) ON DELETE NO ACTION ON UPDATE CASCADE,
    CONSTRAINT worker_campaign_ibfk_2 FOREIGN KEY (campaign) REFERENCES campaign (name) ON DELETE CASCADE ON UPDATE CASCADE
)

```

La tabella **worker\_campaign** rappresenta la relazione tra le campagne e i lavoratori, dato che una campagna generalmente ha tanti lavoratori e i lavoratori possono partecipare a più campagne. La chiave primaria della tabella è quindi data dalle coppie **campagna-lavoratore**. L'informazione del task a cui i lavoratori appartengono è data da due valori booleani: uno per il task di selezione e un altro per il task di annotazione. Nella nostra implementazione quindi un lavoratore può partecipare ad entrambi i task della stessa campagna.

```

CREATE TABLE image (
    url varchar(128) NOT NULL,
    campaign varchar(64) NOT NULL,
    PRIMARY KEY (url),
    KEY campaign (campaign),
    CONSTRAINT image_ibfk_1 FOREIGN KEY (campaign) REFERENCES campaign (name) ON DELETE CASCADE ON UPDATE CASCADE
)

```

Ogni campagna è composta da varie immagini di input. Queste immagini sono salvate nella memoria di massa del server in una directory definita all'interno del **web.xml**. All'interno di questo percorso esiste un'altra directory chiamata **campaigns** che contiene una subdirectory per ogni campagna. La tabella **image** del database si occupa solo di memorizzare il percorso assoluto delle immagini e la loro campagna di appartenenza.

```

CREATE TABLE rating (
    worker varchar(64) NOT NULL,
    image varchar(128) NOT NULL,
    rating tinyint(1) NOT NULL,
    PRIMARY KEY (worker,image),
    KEY image (image),
    CONSTRAINT rating_ibfk_1 FOREIGN KEY (worker) REFERENCES user (username) ON DELETE NO ACTION ON UPDATE CASCADE,
    CONSTRAINT rating_ibfk_2 FOREIGN KEY (image) REFERENCES image (url) ON DELETE CASCADE ON UPDATE CASCADE
)

```

La tabella **rating** memorizza i voti forniti dai lavoratori nel task di selezione. La chiave primaria è la coppia **lavoratore-immagine** (che assicura che ogni lavoratore fornisca un solo voto per la stessa immagine) mentre il voto vero e proprio è dato dall'attributo booleano **rating**, in cui 1 corrisponde al voto positivo e 0 al voto negativo.

```
CREATE TABLE annotation (
    url varchar(128) NOT NULL,
    image varchar(128) NOT NULL,
    worker varchar(64) NOT NULL,
    PRIMARY KEY (image,worker),
    KEY image (image),
    KEY worker (worker),
    CONSTRAINT annotation_ibfk_2 FOREIGN KEY (worker) REFERENCES user
    (username) ON DELETE NO ACTION ON UPDATE CASCADE,
    CONSTRAINT annotation_ibfk_3 FOREIGN KEY (image) REFERENCES image
    (url) ON DELETE CASCADE ON UPDATE CASCADE
)
```

Analogamente la tabella **annotation** memorizza le annotazioni delle immagini usando la coppia **lavoratore-immagine** come chiave primaria. L'annotazione vera e propria è contenuta in un file JSON il cui nome segue la struttura *nomeimmagine\_nomelavoratore.json*. Esse sono memorizzate nella directory della campagna a cui appartengono. Trattandosi di file, il database si limita a contenere il loro percorso nel file system.

```
CREATE TABLE pending_rating (
    image varchar(128) NOT NULL,
    worker varchar(64) NOT NULL,
    timestamp bigint(20) NOT NULL,
    PRIMARY KEY (image,worker,timestamp),
    KEY worker (worker),
    CONSTRAINT pending_rating_ibfk_1 FOREIGN KEY (image) REFERENCES image
    (url) ON DELETE CASCADE ON UPDATE CASCADE,
    CONSTRAINT pending_rating_ibfk_2 FOREIGN KEY (worker) REFERENCES user
    (username) ON DELETE CASCADE ON UPDATE CASCADE
)
```

La tabella **pending\_rating** viene usata dall'algoritmo di scelta delle immagini, descritto in dettaglio nella sezione apposita della documentazione. Viene utilizzata per memorizzare quali sono gli utenti che hanno richiesto un'immagine al server ma non hanno ancora eseguito il task di selezione su di essa, perché ci stanno lavorando. Gli attributi necessari sono la coppia **lavoratore-immagine** (usata come chiave primaria) e il **timestamp** in secondi.

```

CREATE TABLE pending_annotation (
    image varchar(128) NOT NULL,
    worker varchar(64) NOT NULL,
    timestamp bigint(20) NOT NULL,
    PRIMARY KEY (image,worker,timestamp),
    KEY worker (worker),
    CONSTRAINT pending_annotation_ibfk_1 FOREIGN KEY (image) REFERENCES
    image (url) ON DELETE CASCADE ON UPDATE CASCADE,
    CONSTRAINT pending_annotation_ibfk_2 FOREIGN KEY (worker) REFERENCES
    user (username) ON DELETE CASCADE ON UPDATE CASCADE
)

```

È l'analogo di pending\_rating per il task di annotazione, chiamato **pending\_annotation**

Leggendo la descrizione di queste tabelle si sarà notato che delle immagini l'unica informazione memorizzata è la loro posizione, ma non ci sono attributi riguardanti per esempio il numero di voti positivi, il numero di annotazioni ecc... Per fare questo abbiamo realizzato varie viste che si occupano di calcolare in automatico questi valori partendo dalle tabelle esistenti, in modo da non dover aggiornare manualmente le statistiche generando potenziali bug difficili da individuare. Ogni vista si occupa di un aspetto diverso dell'applicazione, mentre una vista finale chiamata **image\_statistics** contiene l'unione di tutte le altre viste, ed è quella da utilizzare nel codice Java per l'accesso alle informazioni.

```

CREATE VIEW rating_stats AS
SELECT i.url AS image, ifnull(count(r.rating),0) AS number_of_ratings,
ifnull(sum(r.rating),0) AS positive_ratings, ifnull((count(r.rating) -
sum(r.rating)),0) AS negative_ratings, ifnull((sum(r.rating) >=
c.least_positive_ratings),0) AS approved
FROM ((image i left join rating r on((r.image = i.url))) join campaign c
on((i.campaign = c.name)))
GROUP BY i.url

```

La vista **rating\_stats** si occupa di calcolare le statistiche riguardanti il task di selezione. I valori calcolati sono:

- Il numero di voti totali ricevuti **number\_of\_ratings**
- Il numero di voti positivi **positive\_ratings**
- Il numero di voti negativi **negative\_ratings**
- L'immagine è approvata per il task di annotazione o meno **approved**

```

CREATE VIEW annotation_stats AS
SELECT i.url AS image, ifnull(count(a.image),0) AS number_of_annotations
FROM (image i left join annotation a on((a.image = i.url)))
GROUP BY i.url

```

Con **annotation\_stats** realizziamo l'analogo di rating\_stats per il task di annotazione. L'unico valore calcolato è il numero di annotazioni ricevute **number\_of\_annotations**

```

CREATE VIEW pending_rating_counter AS
SELECT i.url AS image, count(0) AS pending_request_counter
FROM (pending_rating pr join image i on((i.url = pr.image)))
GROUP BY pr.image

```

La vista **pending\_rating\_counter** contiene, per ogni immagine, il numero di occorrenze all'interno della tabella pending\_rating. Questo valore (che abbiamo chiamato **pending\_request\_counter**) è necessario per il calcolo di un altro attributo nella vista image\_statistics che è cruciale nell'implementazione dell'algoritmo di scelta delle immagini.

```

CREATE VIEW pending_annotation_counter AS
SELECT i.url AS image, count(0) AS pending_request_counter
FROM (pending_annotation pa join image i on((i.url = pa.image)))
GROUP BY pa.image

```

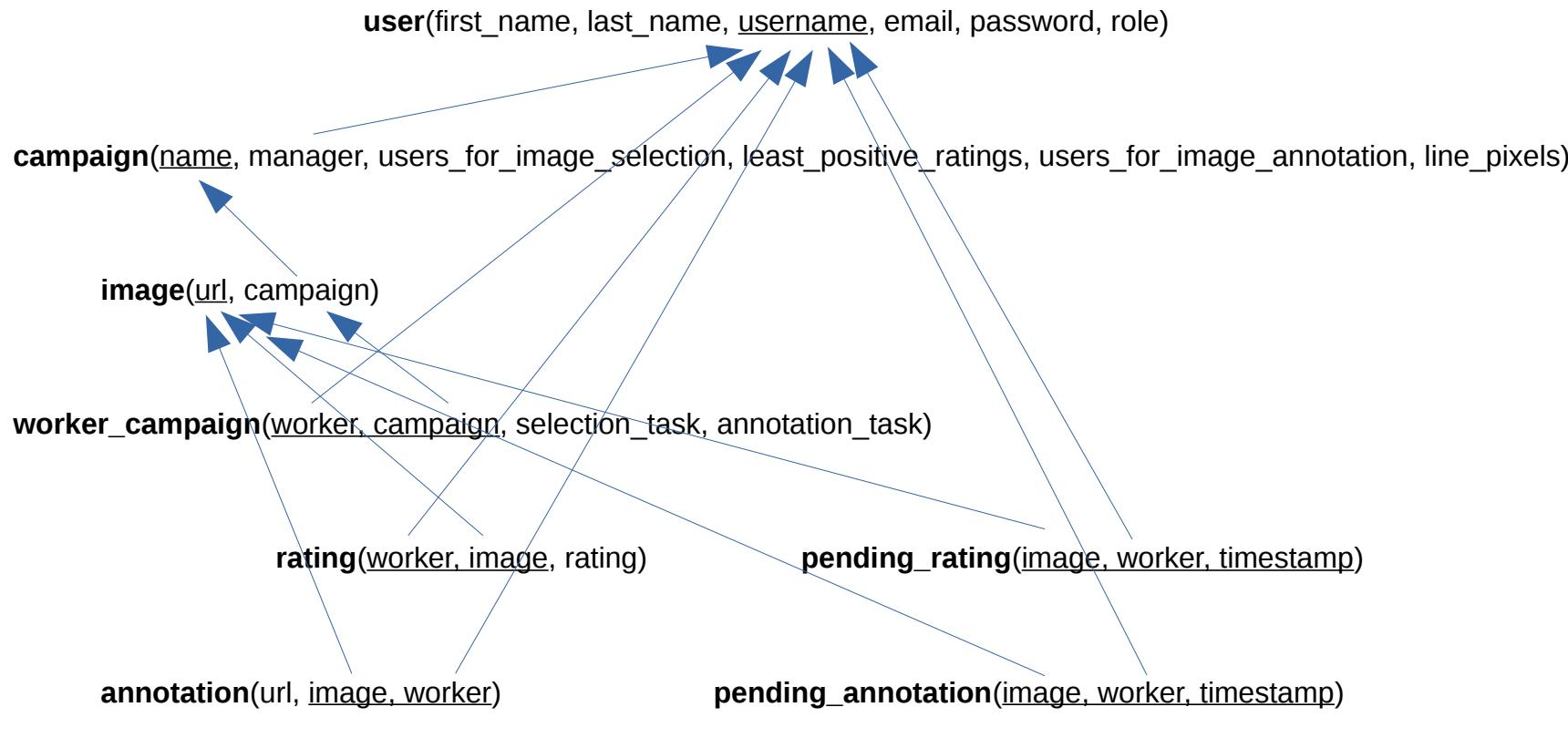
Vista analoga a quella precedente utilizzata per il task di annotazione. Il valore calcolato (cioè il numero di occorrenze di ogni immagine in pending\_annotation) viene chiamato anche qui **pending\_request\_counter**

```

CREATE VIEW image_statistics AS
SELECT rs.image AS image, rs.positive_ratings AS positive_ratings,
rs.negative_ratings AS negative_ratings, rs.approved AS approved,
rs.number_of_ratings AS number_of_ratings, as.number_of_annotations AS
number_of_annotations,(ifnull(prc.pending_request_counter,0) +
ifnull(rs.number_of_ratings,0)) AS rating_request_counter,
(ifnull(pac.pending_request_counter,0) + ifnull(as.number_of_annotations,0))
AS annotation_request_counter
FROM (((image i left join rating_stats rs on((i.url = rs.image))) left join
annotation_stats as on((rs.image = as.image))) left join pending_rating_counter
prc on((rs.image = prc.image))) left join pending_annotation_counter pac
on((rs.image = pac.image)))
GROUP BY rs.image

```

La vista finale, **image\_statistics**, contiene l'unione degli attributi delle viste precedenti. Queste ultime sono solo un appoggio nella costruzione di questa vista, che deve essere l'unica effettivamente usata dall'applicazione per effettuare delle query. Inoltre questa vista è fondamentale nell'implementazione dell'algoritmo di scelta delle immagini perché calcola due valori che sono alla base del suo funzionamento: **rating\_request\_counter** (definito come number\_of\_ratings + pending\_request\_counter della selezione) e **annotation\_request\_counter** (number\_of\_annotations + pending\_request\_counter dell'annotazione). In pratica sono la somma tra il numero di voti (annotazioni) ricevuti e il numero di voti (annotazioni) in attesa di essere ricevuti. Come detto la descrizione approfondita dell'algoritmo è presente nella sezione apposita.



*rating\_stats(image, number\_of\_ratings, positive\_ratings, negative\_ratings, approved)*

*annotation\_stats(image, number\_of\_annotations)*

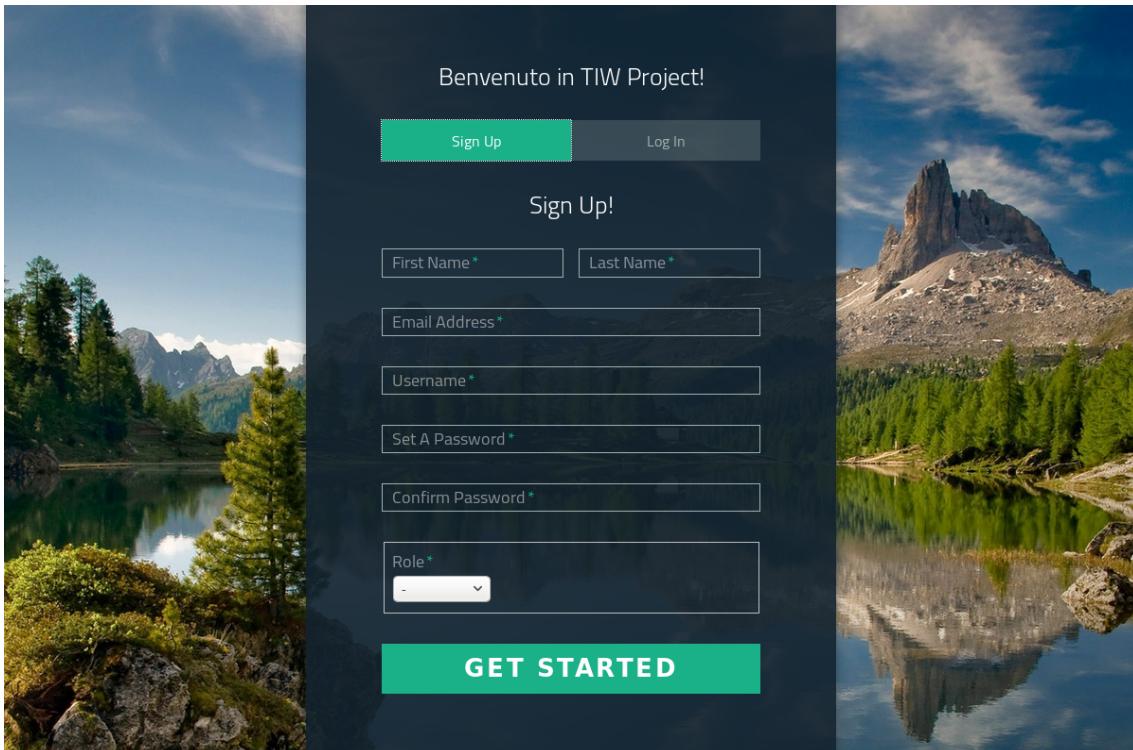
*pending\_rating\_counter(image, pending\_request\_counter)*

*pending\_annotation\_counter(image, pending\_request\_counter)*

*image\_statistics(image, positive\_ratings, negative\_ratings, approved, number\_of\_ratings, number\_of\_annotations, rating\_request\_counter, annotation\_request\_counter)*

# Design della navigazione

La navigazione all'interno dell'applicazione è stata realizzata in modo semplice e intuitivo. Al primo accesso la schermata iniziale mostra una pagina di iscrizione e di login. Negli accessi successivi, se l'utente non ha cancellato i dati della sessione dal proprio browser, viene reindirizzato direttamente alla propria home.



Una volta effettuato l'accesso verrà visualizzata una tabella. Nel caso dei gestori la tabella contiene una lista delle campagne create, con i principali parametri e un pulsante per accedere alle statistiche. Dispone inoltre della possibilità di attivare o cancellare una campagna appena creata. Nel caso dei lavoratori, la tabella contiene le campagne per cui essi sono abilitati, insieme ai pulsanti per avviare il task assegnato e per visualizzare le proprie statistiche. In entrambe le pagine è possibile effettuare il logout tramite un pulsante apposito in alto a destra, mentre i gestori possono creare una nuova campagna selezionando il pulsante **Create Campaign**.

A screenshot of the TIW Project manager's campaign list. The top bar shows "LOGOUT" and the user "Alessandro Fiorillo (Manager)". The table lists three campaigns: "Cancella", "Esempio", and "SuperAlg". Each row includes columns for Name Of Campaign, Users for Image Selection, Least Positive Ratings, Users for Image Annotation, Line Pixels, Statistics, and Active status. For each campaign, there are "VIEW STATS", "ACTIVATE", and "DELETE" buttons. A "CREATE CAMPAIGN" button is at the bottom right.

Name Of Campaign	Users for Image Selection	Least Positive Ratings	Users for Image Annotation	Line Pixels	Statistics	Active
Cancella	5	3	5	6	<a href="#">VIEW STATS</a>	<a href="#">ACTIVATE</a> <a href="#">DELETE</a>
Esempio	3	3	3	3	<a href="#">VIEW STATS</a>	Active
SuperAlg	5	3	5	8	<a href="#">VIEW STATS</a>	Active

Home page di un gestore

*Home page  
di un  
lavoratore*

The screenshot shows a dashboard for a worker. At the top right is a "LOGOUT" button. Below it, the text "Yuri Varrella (Worker)" is displayed. A table lists two campaigns: "Esempio" and "SuperAlg". For each campaign, the manager is listed as "FiorixF1" and the status is "ANNOTATION TASK". To the right of each row is a "VIEW STATS" button. The background of the dashboard features a scenic mountain landscape.

Name Of Campaign	Manager	Enabled Tasks	Statistics
Esempio	FiorixF1	ANNOTATION TASK	<a href="#">VIEW STATS</a>
SuperAlg	FiorixF1	ANNOTATION TASK	<a href="#">VIEW STATS</a>

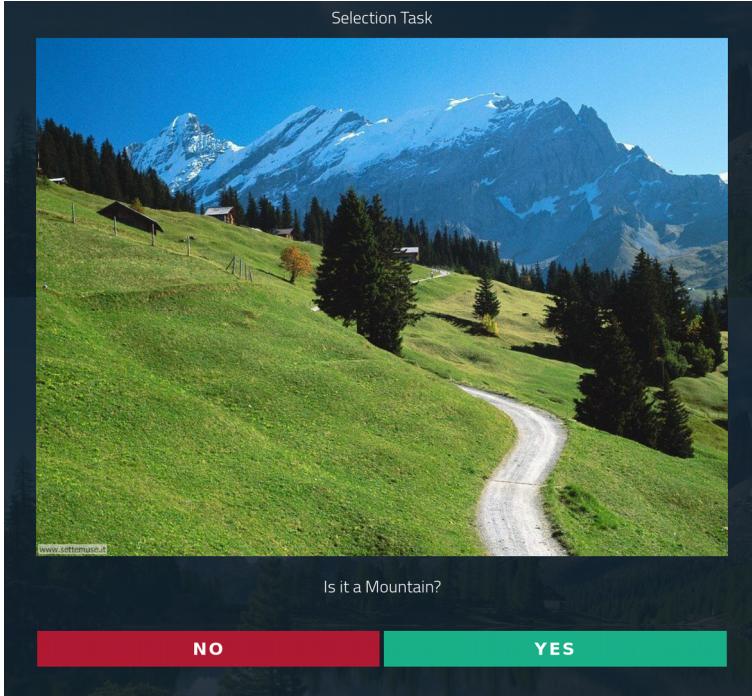
The screenshot shows a form titled "Create New Campaign". It includes fields for "Name of Campaign", "Users For Image Selection", "Least Positive Ratings", "Users For Image Annotation", and "Line Pixels". There is also a section for "Upload Images" with a "Sfoglia..." button and a message "Nessun file selezionato.". Below the form is a table comparing "User For Selection" and "User For Annotation" with checkboxes. At the bottom is a large green "CREATE" button. The background features a scenic view of a lake and mountains.

User For Selection	User For Annotation
91frat	<input type="checkbox"/>
Aresi	<input type="checkbox"/>
berenix	<input type="checkbox"/>
bwoah	<input type="checkbox"/>
CactusFrank	<input type="checkbox"/>
Collx	<input type="checkbox"/>

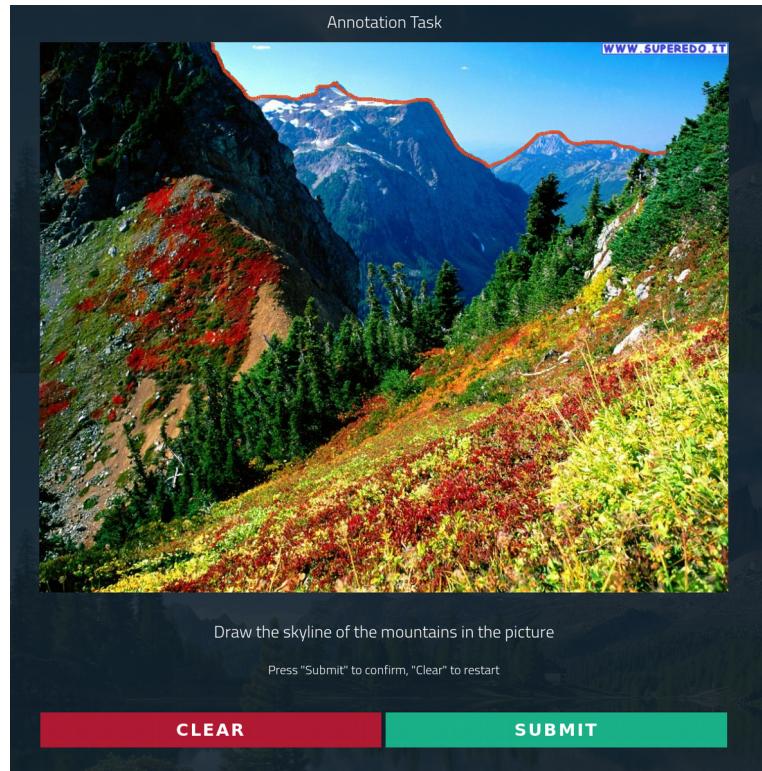
*Pagina di creazione di una campagna*

Le pagine dedicate ai task sono anch'esse semplici e intuitive. Nel task di selezione compaiono l'immagine insieme ai pulsanti **Sì** e **No** per indicare se la foto contiene una montagna. Nel task di annotazione invece si vedono un pulsante per cancellare l'annotazione corrente e un altro per confermare l'annotazione.

Non appena si esegue il task, la pagina viene automaticamente aggiornata con una nuova immagine da elaborare. Per terminare il task è sufficiente premere il pulsante **Home** in alto a sinistra, che torna sulla pagina iniziale, oppure **Logout** in alto a destra per chiudere la sessione.



*Esempio del task di selezione*



*Esempio del task di annotazione*

La pagina delle statistiche mostra una tabella simile a quella pagina iniziale. Nel caso dei gestori ci sono anche tre pulsanti per visualizzare le immagini accettate, rifiutate e annotate con tutte le rispettive annotazioni. Per ogni immagine è presente il numero di voti (positivi e negativi) e il numero di annotazioni. Queste ultime possono essere visualizzate su richiesta premendo il pulsante **Mostra**.

Statistiche campagna "Montanari"

Number of Accepted Images	3
Number of Annotated Images	2
Average Number of Annotations for Image	0.6667

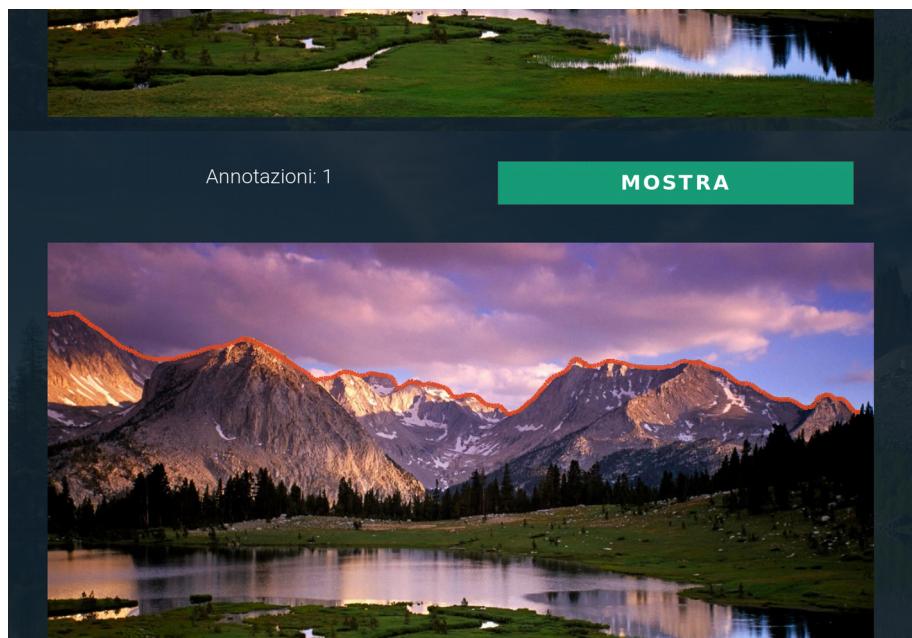
**IMMAGINI ACCETTATE**    **IMMAGINI RIFIUTATE**    **IMMAGINI ANNOTATE**

Immagini accettate



Voti positivi: 3      Voti negativi: 0

*Immagine accettata con 3 voti positivi*



*Immagine accettata insieme alla sua annotazione*

# Assunzioni

Per alcune scelte implementative, come il calcolo delle statistiche, sono state fatte determinate assunzioni, alcune di queste citate velocemente nelle sezioni precedenti.

## Gli utenti possiedono username e email

Gli utenti sono identificati univocamente con il loro username, ma possono effettuare il login tramite quest'ultimo oppure con l'indirizzo email. Ne consegue che anche l'indirizzo di posta deve essere univoco in tutto il database. Inoltre non è possibile per un utente avere il simbolo @ nel proprio username, in quanto potrebbe potenzialmente creare uno username equivalente alla mail di un altro utente. Così facendo uno dei due utenti non sarebbe più in grado di accedere al proprio account, dato che la ricerca dell'utente avviene prima secondo username, poi, se questo non è stato trovato, secondo indirizzo email.

## Le campagne hanno nomi univoci

Ogni campagna è identificata tramite il suo nome, quindi non possono esistere due campagne con lo stesso nome, anche se create da gestori diversi.

## I lavoratori possono svolgere entrambi i task

Un lavoratore può svolgere sia il task di selezione che di annotazione all'interno della stessa campagna.

## Nomi dei parametri della campagna

Ogni immagine deve ottenere N voti nel task di selezione: questo parametro è stato chiamato **users\_for\_image\_selection**. Affinché un'immagine sia poi approvata per il task di annotazione, K di questi voti devono essere positivi: il parametro viene chiamato **least\_positive\_ratings**. Il numero minimo di annotazioni M per ogni immagine viene invece chiamato **users\_for\_image\_annotation**. Infine il parametro PX che indica lo spessore in pixel della linea tracciata si chiama **line\_pixels**.

## Definizione di immagine “approvata” e “non approvata”

Un'assunzione importante è la definizione di immagine approvata e non approvata per il task di annotazione. Abbiamo deciso che un'immagine è “**approvata**” se ha raggiunto i K voti positivi, indipendentemente dal numero totale di voti. Se per esempio una campagna ha i parametri N = 5 e K = 3, ogni immagine passa al task di annotazione non appena arriva a 3 voti positivi, anche se magari i voti totali sono meno di 5. Il fatto che un'immagine sia approvata o meno viene **automaticamente** calcolato dal database, per cui non c'è la necessità di implementare il calcolo manualmente nella query. La vista **rating\_stats** contiene infatti il flag **approved**, per cui le immagini approvate hanno questo attributo settato a 1. Tuttavia se il flag è a 0 non significa necessariamente che l'immagine è stata scartata, perché è il valore iniziale che hanno tutte le immagini quando sono elaborate nel solo task di selezione. Un'immagine è definita come “**non approvata**” se il flag **approved** è a 0 e sono già stati raggiunti gli N voti: in quel caso si può sicuramente dire che l'immagine è stata scartata, perché una volta ricevuti N voti non ne può ottenere altri, quindi non potrà più raggiungere i K voti positivi. Questo calcolo **non** viene svolto automaticamente dal database, per cui va inserito manualmente nella query che cerca le immagini non approvate.

## Struttura del file system

All'interno del web.xml viene definito un path chiamato **update\_location** che contiene la directory in cui verranno scritti e letti tutti i file dell'applicazione. Lì dentro viene creata una directory chiamata **campaigns**, che contiene a sua volta una directory per ogni campagna creata con il suo nome. I dati salvati di ogni campagna sono le immagini (accettati in formato BMP, JPG, GIF e PNG) e le annotazioni (file testuali in JSON).

## Struttura del JSON

Tutte le annotazioni sono salvate in un file testuale in formato JSON. Questi file sono nominati secondo il pattern *nomeimmagine\_nomelavoratore.json*

L'oggetto salvato segue questa struttura:

```
“worker”: nome del lavoratore : String,  
“width”: larghezza del canvas : Number,  
“height”: altezza del canvas : Number,  
“x”: array di coordinate x : Number[],  
“y”: array di coordinate y : Number[],  
“drag”: array di booleani del drag : Boolean[]
```

I valori di **larghezza** e **altezza** del canvas sono necessari per poter ricostruire correttamente l'annotazione su schermi con risoluzione diversa da quella usata da chi ha tracciato l'annotazione. I valori **x** e **y** sono array di numeri contenenti le coordinate dei punti della linea, mentre **drag** è un array di booleani che permette di avere linee separate all'interno della stessa annotazione. Un booleano settato a **false** indica che quel punto è l'inizio di una nuova linea, mentre **true** indica che il punto deve essere unito con quello precedente.

## Definizione delle statistiche

Anche per il calcolo delle statistiche sono state fatte delle assunzioni. Due di queste sono la definizione, per le campagne, di “**numero di immagini approvate**” e “**numero di immagini rifiutate**” già trattate in precedenza. Le altre definizioni sono:

### Numero di immagini annotate (campagne)

Tutte le immagini che hanno almeno un'annotazione

### Numero medio di annotazioni (campagne)

La media di annotazioni di tutte le immagini che sono state accettate. Comprende quindi anche immagini accettate che hanno ancora zero annotazioni.

### Numero di immagini accettate (lavoratore)

Le immagini in cui il lavoratore ha premuto “sì” nel task di selezione.

### Numero di immagini rifiutate (lavoratore)

Le immagini in cui il lavoratore ha premuto “no” nel task di selezione.

### Numero di immagini da selezionare (lavoratore)

Immagini che il lavoratore non ha votato e che non hanno raggiunto il numero minimo di voti né totali né positivi. Ciò garantisce che un utente non voti inutilmente un'immagine che è già stata approvata (o definitivamente rifiutata) per il task di annotazione.

### Numero di immagini annotate (lavoratore)

Le immagini in cui il lavoratore ha inviato un'annotazione.

### Numero di immagini da annotare (lavoratore)

Immagini che il lavoratore non ha annotato e che non hanno ancora raggiunto il numero minimo di annotazioni. Anche questo fa in modo che un lavoratore non annoti un'immagine che ha già raggiunto il numero sufficiente di annotazioni.

# Scelte implementative

Il codice Java è suddiviso in tre package:

- **Beans**: contiene classi che rappresentano oggetti con attributi e i vari getter e setter.
- **Servlet**: contiene tutte le servlet che vengono chiamate nell'applicazione.
- **Start**: contiene una classe chiamata **DatabaseManager** che gestisce internamente l'interfacciamento col database. Fornisce due metodi statici **executeQuery** e **executeUpdate** che prendono come parametri la stringa SQL e gli attributi da inserire nella query. Il metodo **executeQuery** ritorna un **List<Map<String, Object>>** dove String è il nome dell'attributo, Object è il valore in esso contenuto, Map rappresenta una tupla e List contiene al lista di tuple. Invece **executeUpdate** ritorna una stringa che è vuota se l'operazione ha avuto successo, altrimenti contiene un messaggio di errore.

Tutte le servlet in cui l'accesso può avvenire solo dopo il login, contengono nelle prime righe il codice che estrae il nome utente dalla sessione e verifica se quell'utente ha diritto ad accedere alla servlet. Se non ce l'ha, risponde subito con un errore di tipo **403 – Forbidden**. La maggior parte delle servlet estrae e manipola dati dal database, che poi inserisce nella richiesta come attributi e li passa tramite una forward ad una pagina JSP che userà quei dati per generare dinamicamente il contenuto della risposta. Seguendo questa struttura, tutte le richieste fatte dal browser sono dirette ad una servlet, con l'eccezione della pagina iniziale: essa corrisponde al file **index.jsp**, che controlla se l'utente ha una sessione valida. Se ce l'ha fa un redirect alla sua home, altrimenti mostra la pagina di benvenuto per potersi iscrivere o loggare.

Per i gestori, la lista delle campagne create e la creazione di una nuova campagna coesistono in un'unica pagina grazie a due container che vengono mostrati e nascosti in modo alterno dal codice JavaScript. Inoltre per la creazione delle campagne esistono numerosi controlli: alcuni vengono effettuati già al lato client grazie ai tag offerti dall'HTML 5, mentre il controllo completo avviene a lato server. Una campagna viene accettata se:

- Il suo nome non è già stato utilizzato per un'altra campagna
- I quattro parametri numerici sono effettivamente dei numeri interi
- I parametri numerici sono positivi
- I lavoratori scelti per il task di selezione sono almeno N
- I lavoratori scelti per il task di annotazione sono almeno M
- I voti positivi richiesti sono minori o uguali al numero di voti totali richiesti ( $K \leq N$ )
- Lo spessore della linea in pixel è compreso tra 1 e 10
- Sono stati caricati dei file di tipo BMP, JPG, GIF o PNG

Per i lavoratori, i due task sono stati realizzati in modo simile tra loro: la servlet **SelectionTask (AnnotationTask)** contiene la logica di scelta della prossima immagine, che consegna alla pagina **selection\_task.jsp (annotation\_task.jsp)**. Quest'ultima mostra l'immagine e i pulsanti per svolgere il task. Quando il lavoratore invia il risultato del suo lavoro al server, esso viene valutato e salvato dalla servlet **Rating (Annotation)** che poi fa una forward a **SelectionTask (AnnotationTask)** per scegliere nuovamente la prossima immagine. Il lavoratore quindi, non appena invia il voto (l'annotazione), riceve subito la prossima immagine da elaborare e può terminare lo svolgimento del task con un link per tornare alla pagina principale. L'effettiva implementazione dei task è molto semplice per quanto riguarda la selezione, dato che è servito solo inserire due pulsanti che inviano al server la stringa “**yes**” e la stringa “**no**”. Per il task di annotazione invece viene usato un canvas come contenitore dell'immagine e su di esso sono stati definiti degli handler per gli eventi di **mousedown**, **mouseleave**, **mouseup** e **mouseleave**. Al momento del submit

viene generata la stringa JSON che verrà fornita al server. Quest'ultimo verifica la correttezza sintattica del JSON attraverso un'espressione regolare.

La stesura delle statistiche è semplice per quanto riguarda i lavoratori, dato che si tratta solo di riportare il risultato di alcune query. Per i gestori invece bisognava fornire anche la lista delle immagini accettate, rifiutate e delle annotazioni svolte. Per fare ciò, la servlet **ManagerStatistics** si occupa di creare degli array di immagini accettate, rifiutate e di annotazioni svolte. La corrispettiva pagina **manager\_statistics.jsp** si occupa, per ogni immagine, di costruire un tag <img>. La parte più complessa sta nella ricostruzione delle annotazioni, dato che per ognuna di esse viene creato un canvas che deve essere riempito tramite JavaScript (quindi dal client). La soluzione usata, un po' arcana ma funzionante, è stata quella di scrivere tramite JSP un oggetto JavaScript che associa ad ogni immagine una lista di annotazioni. A quel punto è il client ad occuparsi di riempire tutti i canvas (per motivi di sincronizzazione i canvas vengono effettivamente riempiti solo nel momento in cui l'utente chiede di visualizzarli premendo l'apposito pulsante).

La prossima sezione è interamente dedicata alla descrizione dell'algoritmo di scelta delle immagini, dato che la sua progettazione e realizzazione è stata la parte più impegnativa del progetto.

# Algoritmo di scelta delle immagini

Una delle richieste più importanti e impegnative del progetto consiste nel trovare un algoritmo di scelta della prossima immagine da mostrare tale per cui ogni immagine ottenga l'esatto numero minimo richiesto di voti e di annotazioni, rappresentati dai parametri N ed M della campagna.

La soluzione trovata è piuttosto articolata, tanto che per testare la correttezza dell'algoritmo (non solo come implementazione, ma anche come validità logica), è stato realizzato uno script in Python che esegue l'algoritmo in un ambiente simulato in cui più utenti lavorano contemporaneamente sulla campagna, richiedendo immagini e inviando voti e annotazioni al server. In questo script il server e le immagini sono rappresentati tramite delle classi omonime, in cui il **server** realizza le funzioni di consegna delle immagini e ricezione di voti e annotazioni, mentre le **immagini** sono rappresentate con un numero identificativo e delle variabili che corrispondono agli attributi presenti nel database. Ogni **utente** è rappresentato da un thread che in maniera asincrona chiede una nuova immagine al server oppure consegna un voto o un'annotazione. Inoltre è stata inserita la possibilità che un utente richieda un'immagine ma poi non esegua il task, simulando l'evento di un utente reale che dopo aver ottenuto l'immagine chiude il browser oppure aggiorna la pagina. Lo script viene eseguito per un certo periodo di tempo al termine del quale stampa a video i risultati, che consistono nel numero di voti e annotazioni ricevuti da ogni immagine e alcune statistiche rilevanti. Tra queste statistiche figurano il numero e la percentuale di immagini che hanno ricevuto esattamente N voti (M annotazioni), il numero massimo di voti (annotazioni) in più che un'immagine ha ricevuto e la media di voti (annotazioni) in più ricevute da ogni immagine. Molti parametri come il numero di immagini, il numero di utenti, il numero minimo di voti (annotazioni) e persino la probabilità che un utente non esegua il task sono configurabili. Tramite queste statistiche e diverse configurazioni dei parametri abbiamo valutato l'efficacia degli algoritmi provati. L'algoritmo ideale infatti dovrebbe fare in modo che la percentuale di immagini con esattamente N voti (M annotazioni) sia del 100% e di conseguenza che la media di voti (annotazioni) extra ricevuti sia zero. L'algoritmo che alla fine è stato implementato nell'applicazione web riesce a raggiungere questo obiettivo. Trattandosi di un algoritmo sofisticato, lo presenteremo passo passo, partendo da un algoritmo più semplice e banale. Da quello indicheremo quali sono i suoi difetti e come li abbiamo risolti, arrivando via via all'algoritmo definitivo. Come spiegato in precedenza, lo script Python realizzato si basa simulando i task di selezione e annotazione eseguiti da più utenti. Tra i due task però c'è una differenza da tenere in considerazione: nel caso del task di selezione le immagini della campagna sono disponibili già all'avvio di quest'ultima, mentre nel task di annotazione le immagini inizialmente non sono disponibili, perché diventano annotabili solo dopo aver superato il primo task. Per questo motivo esistono due versioni dello script: una prima versione in cui le immagini sono tutte disponibili fin da subito (quindi rappresenta più fedelmente il task di selezione) e una seconda in cui le immagini diventano disponibili a run-time (rappresentando quindi con maggior precisione il task di annotazione). D'ora in poi, per semplicità di scrittura, ci riferiremo al solo task di annotazione, ma le considerazioni fatte sono equivalenti per il task di selezione, a meno che non verrà detto esplicitamente.

Iniziamo perciò il percorso verso l'algoritmo definitivo partendo da una situazione iniziale molto semplice. Se si richiede che un'immagine deve ottenere almeno M annotazioni, la primissima soluzione che viene in mente è, quando un utente richiede un'immagine, consegnare quella con il minor numero di annotazioni all'attivo tra quelle che non hanno

ancora ricevuto la quantità necessaria. Si tratta di un algoritmo facile sia concettualmente che da implementare. Facendolo eseguire più volte al simulatore Python, si ottiene che su 100 immagini in una campagna con 100 utenti e un target di 25 annotazioni, solo il **12 %** delle immagini arriva ad ottenere il numero esatto di M annotazioni, mentre le restanti arrivano ad avere anche una media di **40** annotazioni più del necessario, con dei picchi massimi di **100** annotazioni. Si comporta in maniera diversa ma comunque insoddisfacente nel task di selezione: rimangono il 12 % circa le immagini con esattamente N voti, mentre le restanti hanno mediamente 10 voti in più, con picchi di 40. Il motivo per cui ciò accade è molto semplice. Supponiamo di avere una campagna in cui il parametro M vale 10 e rimane un'immagine con 9 annotazioni. Quando un utente fa una richiesta al server, quest'ultimo consegnerà per forza l'immagine con 9 annotazioni, essendo l'unica rimasta. L'utente quindi riceve l'immagine e la annota, ma finché non ha concluso e inviato l'annotazione, qualunque altro utente che faccia una richiesta al server continuerà ad ottenere quella stessa immagine. Quando l'utente iniziale avrà consegnato l'annotazione, l'immagine avrà sì ottenuto le 10 annotazioni necessarie, ma successivamente arriveranno anche le annotazioni di tutti gli utenti che si sono susseguiti, consegnando delle annotazioni sprecate. Il prossimo passo quindi è fare in modo che il server si ricordi di aver consegnato un'immagine ad un certo utente e usare questa informazione per gestire meglio le richieste successive.

Partendo da questo problema siamo giunti ad una versione migliorata dell'algoritmo precedente. Per ogni immagine non memorizziamo solo il numero di annotazioni ricevute (che possiamo chiamare `annotation_counter`), ma anche il numero di richieste, cioè il numero di volte in cui l'immagine è stata inviata in risposta ad un utente (e lo chiameremo `request_counter`). Questo valore viene incrementato ogni volta che inviamo un'immagine come risposta e lo usiamo nella politica di scelta. L'algoritmo basilare del "prendi l'immagine con il minor `annotation_counter` tra quelle che hanno `annotation_counter < M`" diventa "prendi l'immagine con il minor `request_counter` tra quelle che hanno `annotation_counter < M`". Eseguendo questo algoritmo sul simulatore si nota un netto miglioramento: il **35 %** delle immagini riesce ad avere esattamente M annotazioni, mentre nelle restanti le annotazioni di troppo sono mediamente solo **2**, con un picco isolato di **10** annotazioni in più. Il motivo per cui alcune immagini continuano ad avere delle annotazioni in più è simile a quello dell'algoritmo precedente. Ponendo una campagna con  $M = 10$  e un'immagine che conta 7 annotazioni e 10 richieste, quest'immagine verrà comunque consegnata a chi fa una richiesta, dato che il discriminante tra un'immagine che può essere consegnata e una definitivamente archiviata rimane il numero di annotazioni. Tuttavia usando il numero di richieste anziché il numero di annotazioni per scegliere tra i potenziali candidati aumenta visibilmente le prestazioni dell'algoritmo.

Dopo aver pensato questi due algoritmi, è stato difficile trovare delle alternative più efficienti, ma grazie alla presenza del simulatore abbiamo potuto fare degli esperimenti che hanno fornito dei risultati interessanti. Cosa succederebbe se per la scelta delle immagini non ci basassimo anche sul numero di annotazioni ricevute, ma esclusivamente sul numero di richieste? Abbiamo sperimentato questa idea sul simulatore, aggiungendo però nei parametri la condizione per cui è garantito che quando un utente fa una richiesta eseguirà il suo task. Così facendo il risultato è stato sorprendente: il **100 %** delle immagini ha ricevuto esattamente M annotazioni, proprio come richiesto dalle specifiche. Il problema principale però è proprio l'assunzione che ogni utente esegua il suo task quando riceve l'immagine. Nella realtà non è così, perché nulla vieta all'utente di ottenere un'immagine e poi chiudere il browser, oppure aggiornare la pagina ricevendo così un'immagine diversa. La parte più complessa e insidiosa (ma affascinante) dell'algoritmo

implementato è proprio la gestione di questi casi: come capire che un utente non ha eseguito il task e di conseguenza rendere disponibile un'immagine per gli altri utenti? La soluzione proposta prevede di utilizzare una tabella nel database in cui memorizzare tutte le richieste per le quali non è ancora stato riscontrato il risultato del task: questa tabella (chiamata pending\_annotation per l'annotazione e pending\_rating per la selezione) contiene in ogni tupla l'utente che ha fatto la richiesta, l'immagine che gli è stata consegnata e il timestamp. Periodicamente si controlla questa tabella e si cancellano le richieste per le quali è passato un certo periodo di tempo senza riscontro (decrementando anche il request\_counter dell'immagine corrispondente). Se invece l'utente esegue il task e invia l'annotazione, cancelliamo la richiesta dalla tabella, ma manteniamo inalterato il contatore di richieste (che può quindi essere visto meglio come la somma di annotazioni ricevute e annotazioni pendenti). Bisogna definire meglio cosa si intende per "periodicamente" e "un certo periodo di tempo". Il controllo della tabella deve verificarsi piuttosto frequentemente, in modo da lavorare con dati aggiornati: la strategia migliore infatti è quella di effettuare il controllo ogni volta che un utente richiede un'immagine come prima operazione, in modo che i dati utilizzati per la scelta della prossima immagine siano sempre aggiornati. Anche la scelta del timeout è un aspetto fondamentale, e anzi è cruciale per il corretto funzionamento dell'algoritmo. Un periodo di timeout troppo lungo lascerebbe inutilmente attive delle richieste per cui non c'è riscontro, mentre un periodo troppo corto porta a risultati disastrosi. Mettiamo caso per esempio che il tempo di timeout sia di 10 minuti. Un utente richiede un'immagine, ma subito dopo per qualche motivo abbandona la postazione (ma lascia aperto il browser). Dopo 15 minuti ritorna e invia l'annotazione. A lato server accade che la richiesta di quest'utente è stata cancellata dal database perché è scaduto il tempo limite, ma l'annotazione è arrivata comunque. Così facendo il contatore request\_counter è stato decrementato, ma annotation\_counter è stato incrementato: il risultato è che il numero di annotazioni ricevute è paradossalmente superiore al numero di richieste! Considerando che il numero di richieste è il parametro usato come discriminante per la scelta della prossima immagine, questo porta ad avere tranquillamente immagini con più di M annotazioni. L'unica soluzione è settare come tempo di timeout un valore che sia almeno la durata della validità di una sessione web.<sup>1</sup> Un valore più basso del timeout renderebbe l'algoritmo vulnerabile in queste situazioni. Il tempo di validità di una sessione web è comunque eccessivamente lungo (20 o 30 minuti) considerando che il task di annotazione richiede meno di 5 minuti per essere eseguito e il task di selezione qualche secondo. Si possono allora fare delle considerazioni, ovvero un task non viene eseguito se l'utente compie una delle seguenti azioni:

- Chiude il browser
- Fa il logout
- Aggiorna la pagina

Per la prima situazione non possiamo fare nulla, ma negli altri due scenari possiamo eliminare le richieste che non avranno riscontro già in anticipo senza dover aspettare il tempo di timeout. Se un utente chiede di fare il **logout** dal sito, si può già in quel momento cancellare tutte le sue richieste per cui non ha svolto il task, perché se esce dal sito è sicuro che non potrà svolgerli. Nel caso dell'aggiornamento della pagina, basta cancellare tutte le richieste dell'utente ogni volta che esso ne fa una.

Aggiungendo tutte queste caratteristiche all'algoritmo di scelta e implementandole nel simulatore, si ottengono i risultati sperati: il **100 %** delle immagini riesce ad avere esattamente M annotazioni, anche in presenza di utenti che non svolgono il task. A questo

---

<sup>1</sup> In realtà nell'applicazione web il valore di request\_counter è calcolato automaticamente dalla vista `image_statistics` come `# annotazioni ricevute + # annotazioni pendenti`, per cui non bisogna incrementarlo e decrementarlo manualmente, come accade invece nello script Python.

punto abbiamo tutti i pezzi necessari per descrivere l'algoritmo di scelta delle immagini nel suo complesso:

### L'utente X vuole eseguire il task

→ servlet AnnotationTask e SelectionTask

V

Cancellare tutte le richieste scadute, ovvero le righe di pending\_annotation in cui now – timestamp > session-timeout

Ogni richiesta cancellata decrementa request\_counter

V

Cancellare tutte le richieste precedenti dell'utente, ovvero le righe di pending\_annotation in cui worker = X

V

Cercare tutte le immagini con request\_counter < M

V

Tra queste prendere le immagini non annotate da X

V

Tra queste prendere le immagini col minor valore di request\_counter

V

Se ci sono più immagini candidate, sceglierne una a piacere

V

Aggiungere la richiesta attuale in pending\_annotation, questo incrementa request\_counter

V

### Inviare l'immagine come risposta

### L'utente X invia un'annotazione

→ servlet Annotation e Rating

V

Memorizzare l'annotazione e cancellare da pending\_annotation la richiesta corrispondente

V

request\_counter resta invariato

### L'utente X richiede il logout

→ servlet Logout

V

Cancellare da pending\_annotation tutte le richieste dove worker = X

Ogni richiesta cancellata decrementa request\_counter

V

### Effettuare il logout

Gli script Python che simulano l'algoritmo sono in allegato a questa documentazione con i nomi **selezione.py** e **annotazione.py**. Per eseguirli basta caricarli nell'interprete Python, senza la necessità di parametri aggiuntivi.