FRAMEWORK FRONT-END

Prima release: 1.0 del 09/05/2024
Versione 1.1 del 30/05/2024

INFO

Questa guida descrive le caratteristiche peculiari del framework front-end.
Si concentra in particolar modo sulle customizzazioni che lo rendono sostanzialmente diverso da una comune applicazione Ember e fornisce delle direttive per la standardizzazione del codice.

Prerequisiti fondamentali sono:

- conoscenza approfondita di Ember.js
- conoscenza approfondita dei plugin aggiuntivi
- conoscenza dell'architettura e dei contenuti del tema grafico utilizzato (Architect)

È inoltre raccomandabile una buona conoscenza di Cordova per la generazione delle app Android e iOS.

EMBER.JS

Le guide base di Ember.js sono disponibili qui (per la versione italiana delle guide potete usare con buoni risultati il convertitore Google. L'unica attenzione riguarda la traduzione dei termini 'model' e 'template' che vengono entrambi tradotti in 'modello').

Dopo aver letto la sezione *Getting Started* e seguito scrupolosamente i due *Tutorial* proposti, è essenziale studiare almeno le seguenti voci del menù laterale sinistro: *Components*, *Routing*, *Services*, *EmberData*, *In-Depth Topics*, *Application Concerns*. *Sebbene non strettamente necessario, i più operosi vorranno poi seguire le restanti guide della pagina, approfondire la conoscenza con la documentazione Api e gli approfondimenti sulla CLI.*

PLUGIN AGGIUNTIVI

Il framework si basa fortemente su alcuni plugin Ember aggiuntivi, la cui conoscenza è indispensabile: render-modifiers, truth-helpers, event-helpers, gli importantissimi ember-concurrency e ember-file-upload, l'essenziale tracked-built-ins, l'utility ember-uuid, oltre ai due plugin su cui si basano le funzioni core del framework, ember-simple-auth e ember-permissions. Senza la conoscenza dei suddetti, è pressoché inutile proseguire nella lettura di questa guida.

ARCHITECT UI

Sebbene Ember non abbia una vera e propria sezione dedicata al layout, il framework ci mette a disposizione un tema personalizzabile e un intero menù di Setup per la customizzazione.
La conoscenza del tema è indispensabile per diversi fattori:

- evitare di scrivere codice di stile ridondante rendendo la build (e di fatto il sito stesso) più pesante;
- mantenere uno stile omogeneo in tutte le pagine del sito;
- evitare errori che rendano il layout mal funzionante su mobile/app o desktop

Dando per scontata la conoscenza di scss e di Bootstrap 4, sui quali si basa il tema Architect, è importante notare che il tema, originariamente basato su handlebar e web-pack, è stato fortemente rielaborato (nell'architettura, non nella grafica) per adattarsi ai meccanismi di build di Ember e alla sua natura SPA (single-page-application).

Inoltre la sezione principale del tema (l'head e una parte del body), comune sia alle pagine dedicate agli utenti non loggati che a quelle dedicate agli utenti loggati, è stata inglobata nel framework, anche sotto forma di componenti dedicati che interagiscono con il database per leggere le configurazioni di setup.

Per facilitare l'accesso al codice del tema, nel repository del framework è disponibile una cartella ad esso dedicata. All'interno sono disponibili due sotto-cartelle: ***architectui-html-pro*** e ***src/DemoPages***. La prima contiene le singole pagine html del tema, utili per analizzare via browser le features da esso offerte. La seconda, organizzata in *webpack-style*, contiene i sorgenti delle pagine stesse ed è da qui che dovremo prelevare il codice da inserire nei nostri template e componenti.

È assolutamente sbagliato prelevare il codice html dall'*inspector* del browser: essendo il risultato del rendering eseguito da jQuery e da Bootstrap, l'html visualizzato nel browser non corrisponde a quello originale del tema e comporta malfunzionamenti o impaginazioni grafiche inaspettate. È altrettanto inopportuno importare codice dai tutorial di Bootstrap reperibili in rete, poiché i fogli di stile del tema potrebbero renderizzare in modo inaspettato il codice non contemplato dal tema stesso.

Una sentinella dell'utilizzo improprio delle potenzialità del tema può essere il ricorso massivo ai fogli di stile. Nella maggior parte dei casi non avremo assolutamente bisogno di scrivere fogli di stile aggiuntivi perché il tema fornisce nativamente gli elementi e le classi necessarie alla maggior parte delle impaginazioni. Salvo poche righe di utilità che potrebbero tornarci utili, se i nostri fogli di stile custom iniziano a diventare sostanziosi, dobbiamo chiederci se stiamo sfruttando correttamente le potenzialità del tema o se stiamo erroneamente introducendo uno stile che non rispecchia quello del tema.
Va inoltre tenuto conto che la fase di build del framework (anche e soprattutto in fase di CI/CD) è fortemente appesantita dal processore scss (non presente nativamente in Ember) che deve elaborare i numerosi *scss* del tema; evitiamo di aggiungere altri fogli di stile quando non strettamente necessari!

Va inoltre dato uno sguardo (possibilmente in sola lettura!) al file di customizzazione **styles/custom/custom.scss** che contiene classi di utilità di uso comune nel framework (ad esempio la classe *mae-table* per le tabelle).

Infine, per quanto dovrebbe essere sottinteso per chi conosce bene Bootstrap, vanno sempre ricordati i numerosi file *_variabbles.scss* e *_layout-variables.scss* che definiscono nel profondo lo stile del tema e dai quali dovremmo importare le variabili per un corretto sviluppo dei nostri fogli di stile (pensiamo a cosa succederebbe se di punto in bianco un cliente decidesse di modificare totalmente i colori del tema…!)

LIBRERIE ESTERNE… PERCHÉ?

Lo scopo del framework è quello di fornire una piattaforma completa per lo sviluppo. Questo vuol dire che le librerie di terze parti più utili sono già state incluse, vuoi perché ritenute dei *must-have* nello sviluppo Javascript, vuoi perché necessarie al tema Architect. Queste ultime sono già particolarmente numerose e contribuiscono significativamente ai tempi di build e alla dimensione finale del javascript e del css post-build. Tenendo conto che il framework consente di generare applicazioni Android e iOS, comprendiamo bene quanto il peso degli .apk e degli .ipa sia un deterrente al loro download o alla loro installazione!

Laddove ce ne fosse un reale e insormontabile bisogno, comunque, useremo **npm** per la loro installazione e le importeremo nel file ember-cli-build.js come imparato dalle guide di Ember!
È vivamente consigliabile l'analisi del file **ember-cli-build.js** per conoscere l'elenco delle librerie disponibili nel framework.

NO JQUERY!

Sebbene disponibile nel framework, l'utilizzo della libreria jQuery è fortemente sconsigliato, perchè, seppure Ember non abbia un concetto di DOM virtuale marcato come in altri framework (React, Vue ecc), il binding bidirezionale tra interfaccia e Javascript (che è ciò che caratterizza Ember) comporta l'inadeguatezza di jQuery, che nasceva appunto per la manipolazione del DOM!
L'unica eccezione riguarda l'utilizzo di eventuali plugin su di essa basati, ma in tal caso è conveniente realizzare componenti wrapper che consentano di utilizzare il plugin come un normale componente Ember.

LEGAL

Il framework è stato sviluppato nel rispetto delle normative GDPR.
Nel menù **Contenuti**/"**Documenti legali**" è stata dunque definita un'apposita area dedicata alla compilazione dei documenti legali di "*Informativa sulla Privacy*", "*Termini e Condizioni*" e "*Accettazione dei Cookie*", il cui accesso è riservato ai soli utenti a cui abbiamo assegnato il ruolo "*Legal*".

Faccio notare che il titolo "Accettazione dei Cookie" è esemplificativo e sarebbe più opportuno parlare di LocalStorage. ***I cookie non devono essere utilizzati*** *perché mal compatibili con lo sviluppo delle App.*

Il framework si occupa di chiedere agli utenti l'accettazione dei suddetti documenti legali in occasione del primo accesso e richiede l'accettazione ogni volta che apportiamo modifiche salienti ai suddetti contenuti.

Anche l'utilizzo dei **LocalStorage** dovrebbe essere evitato o ridotto al minimo e in ogni caso, per ciascun nuovo parametro che memorizziamo in essi, l'informativa sulla privacy deve essere aggiornata al fine di rendere trasparenti le motivazioni per le quali imponiamo all'utente un salvataggio dati permanente.

Allo stato attuale utilizziamo i localStorage per memorizzare 4 parametri:

- poc-allow-cookie:
  indica se l'utente ha espresso il consenso all'utilizzo dei cookie e, in caso affermativo, indica se ha accettato tutti i cookie o solo quelli tecnici. Su questo parametro si basa l'apposito componente che, in ambiente Web e Android mostra all'utente una modale di autorizzazione all'uso dei cookie (in ambiente iOS il popup non viene mostrato, come previsto dal regolamento Apple).
- poc-allow-gdpr:
  memorizza il timestamp in cui l'utente ha accettato i "termini e condizioni" e la "privacy-policy".
- poc-fingerprint:
  si tratta di un guid che identifica univocamente il browser da cui è connesso l'utente ed è utilizzato per il multi-sessione
- poc-user-lang:
  memorizza il codice di lingua dell'utente corrente, utilizzato per il multi-lingua.

Ci sarebbe un quinto parametro che il framework crea al login e distrugge al logout. Ne parleremo più avanti ed è il *luogo* più adatto in cui memorizzare i dati di *sessione*.

I PREREQUISITI

Nel readme.md del repository del framework sono elencati tutti i passaggi di installazione, indispensabili per il corretto avvio del framework stesso. In particolar modo abbiamo bisogno di:

- **nvm**: la versione minima richiesta è la 0.39.3.
- **npm**: la versione minima richiesta è la 9.5.1 (e va installata ovviamente tramite nvm)
- **Node.js**: la versione minima richiesta è la 18.16.0 (e va installata tramite npm)
- **Git**: più che sottinteso!
- Ember-cli: fondamentale per il funzionamento di Ember.

Gli utenti Mac potrebbero essere abituati a lavorare con Yarn piuttosto che con npm ed ovviamente il framework è predisposto anche a questo. Anzi, molte delle dipendenze sono state installate con Yarn perchè npm non riusciva nell'intento! Per installare Yarn useremo npm (il che potrebbe sembrare ironico, dato che è un'alternativa a npm): npm install -g yarn.

CREARE UN NUOVO PROGETTO

Per la creazione di un nuovo progetto andiamo nel repository del Framework disponibile qui.
In alto a destra premere il pulsante FORK; si aprirà un'interfaccia dedicata.

Nella sezione "Project Name" definire il nome del nuovo Progetto, seguendo la sintassi qui riportata:

```
DEST-NOME-ENVIRONMENT
```

(esempio: EXT-OrarioSanteMesse-FE), in cui:

- DEST definisce il tipo di progetto e può assumere i seguenti valori:
  - EXT: per progetti destinati ai Clienti
  - INT: per progetti interni di Maestrale
  - TS: per progetti destinati a TeamSystem
- NOME nome sintetico del progetto, senza spazi
- ENVIRONMENT indica se il repository è di tipo backend o frontend. Può assumere i seguenti valori:
  - BE: per progetti backend
  - FE: per progetti FE (ovviamente useremo questo valore).

Nella sezione "Project Url" selezionare la voce "mit.custom.ember" e confermare l'operazione premendo il pulsante "Fork project". Una volta creato il Fork, clonarlo nel proprio PC.

I SETTAGGI DI BASE

Una volta clonato il nuovo progetto nel computer, è necessario personalizzare le configurazioni e testare il funzionamento dell'applicativo, come indicato di seguito.

INSTALLAZIONE DELLE DIPENDENZE

Apriamo il terminale nella cartella WEB del progetto e lanciamo il comando ***npm install*** per installare le dipendenze *Node* necessarie.

PUNTAMENTI

Nella root principale del progetto (cartella WEB) aprire il file config/environment.js.

Al suo interno personalizzare le variabili di puntamento agli ambienti di back-end:

- ***apiHostDev***: punta all'ambiente di sviluppo,
- ***apiHostTest***: punta all'ambiente di UAT (*User Acceptance Test*), ovvero all'ambiente che il Cliente consulta per validare le implementazioni,
- ***apiHostProd***: punta all'ambiente di produzione.

I valori da inserire dovranno ovviamente essere forniti dai Sistemisti e/o dagli sviluppatori BE.
Modificare inoltre le variabili:

- **feHostDev**: indirizzo pubblico dell'applicativo FE in ambiente di sviluppo,
- **feHostTest**: indirizzo pubblico dell'applicativo FE in ambiente di sviluppo,
- **feHostProd**: indirizzo pubblico dell'applicativo FE in ambiente di sviluppo

NOTA: per rendere effettive le modifiche al file environment.js, è necessario arrestare (*Control* + C) e quindi riavviare (*ember serve*) la Ember-CLI.

CSP (Content Security Policy)

Aprire il file **config/content-security-policy.js** ed aggiornare i puntamenti agli ambienti.

CONFIGURAZIONE OFFLINE

Nel caso in cui l'applicativo non riuscisse a dialogare con il back-end, verranno utilizzate le configurazioni presenti nella cartella WEB/app/_customs all'interno dei file **layoutStile.js** e **offLineConfig.js**. Personalizzare i valori al loro interno in base alle necessità.

MENÙ LATERALE

Sempre nella cartella WEB/app/_customs è presente il file **sidebarContents.js** che contiene le impostazioni del menù laterale. Il file prevede tutti i commenti necessari a comprenderne il funzionamento. Ciò che possiamo anticipare è che per ciascuna voce del menù possiamo:

- specificare se la visualizzazione è disponibile su App e/o su Web;
- specificare in quali environment è visualizzabile (development, test, production);
- specificare quali eventuali permessi dell'utente (*claims*) sono necessari per la sua visualizzazione.

COLORI CUSTOM

Sebbene il tema preveda la possibilità di impostare diverse combinazioni di colore predefinite, è comunque possibile aggiungere due varianti personalizzate.
A tal scopo aprire il file **WEB/app/styles/custom/custom.scss** e personalizzare le seguenti variabili:

- **$custom-bg-color1** : colore di sfondo, applicabile sia ai bottoni che al tema (header e sidebar) del sito
- **$custom-text-color1** : colore del testo dei bottoni
- **$custom-bg-color2** : secondo colore di sfondo, applicabile ai bottoni (e non al tema)
- **$custom-text-color2** : secondo colore del testo dei bottoni

PORTA

Di default l'applicativo è raggiungibile alla porta 4200 del localhost. E' possibile modificare tale valore nel file **.ember-cli** della root principale. L'utilizzo locale di porte diverse consente l'esecuzione contemporanea di più applicativi basati sul framework.

PERSONALIZZAZIONE DEI LOGHI

Aprire la cartella **WEB/public/assets/utils/images/** e personalizzare le immagini **logo.png** e **logo-inverse.png** (senza modificarne le dimensioni).

TITOLO DEL SITO

Aprire il file **index.html** della root principale e modificare il titolo del sito nell'attributo <title>.

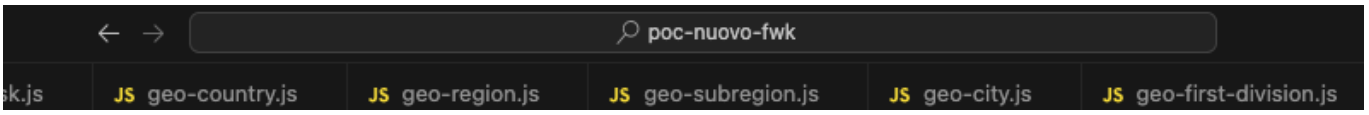
FAVICON

Nella cartella WEB/dist modificare l'immagine **favicon.ico** senza modificare né la dimensione né l'estensione del file.

Terminati i settaggi iniziali possiamo testare l'applicativo lanciando nel terminale il comando *ember serve* ed aprendo il browser all'indirizzo http://localhost:4200/ (o altra porta scelta).

EDIT DEL CODICE

Per lavorare al codice useremo Visual Studio Code. Il modo corretto di importare il progetto è quello di selezionare l'intera cartella del repository attraverso il menù File/"Apri cartella…". In questo modo il nome del repository apparirà nell'header di Visual Studio Code (utile quando abbiamo più istanze di VSC aperte contemporaneamente).



È essenziale impostare le seguenti configurazioni:

1. installare tutti i **plugin raccomandati** di *Visual Studio Code* qui elencati:
    - Ember JS Snippets
    - Ember JS (ES6) and Handlebars code snippets
    - Ember.js Extension Pack
    - VS Code ESLint extension
    - Glimmer Templates Syntax for VS Code
    - vscode-glimmer
    - jsdoc
    - Auto rename tag
    - Rainbow Brackets
    - Region marker

2. nelle *Preferenze* di Visual Studio Code impostare '***Tab Size***' a **due spazi**
3. Nelle *Impostazioni* di Visual Studio Code, alla voce "**Search: Exclude**", aggiungere i seguenti criteri:
    - **/ARCHITECT (exclude Theme folder from search)
    - **/APP (exclude APP folder from search)
    - **/WEB/dist (excludes distribution folder from the search)
    - **/node_modules
    - **/bower_components

    in questo modo le eventuali ricerche in tutta la soluzione non forniranno risultati indesiderati o ridondanti.
4. Se avete impostato il salvataggio automatico delle modifiche ai file su Visual Studio Code è essenziale disattivarlo per non sovraccaricare la CPU con le continue build accodate della Ember-cli.

EMBER-CLI

Il modo migliore per gestire la Ember-cli è quello di aprire due *prompt dei comandi* di Windows (ovvero due *Terminali* su mac) nella cartella WEB del framework ed usare il primo per lanciare il comando *ember serve* ed il secondo per lanciare tutti i comandi di generazione degli elementi di ember e per gli unit-test.

Se avete la necessità di eliminare elementi precedentemente creati con *ember generate*, è essenziale utilizzare ***ember destroy***, così che la *ember-cli* provveda ad eliminare i relativi unit-test che altrimenti, restando *orfani*, produrrebbero falsi errori di build.

GLI ESEMPI IN QUESTA GUIDA

Per non appesantire inutilmente la guida, gli elementi di importazione negli esempi Javascript sono stati omessi.

IMPOSTAZIONI

Il framework presenta un vasto menù di Setup. Molte delle voci sono auto esplicative e non verranno approfondite. In questa sezione chiariremo i parametri di configurazione che non sono interamente gestibili da interfaccia.

VISIBILITA' DELLE PAGINE PER UTENTI NON LOGGATI

In questa sezione vedremo una prima differenza tra una comune applicazione Ember (come quelle dei tutorial) ed un'applicazione realizzata con il framework.

Di default gli utenti non loggati al framework visualizzano la pagina di login. Se abbiamo definito una rotta custom e proviamo a navigare verso essa, senza essere loggati, non riusciremo a visualizzarla (o più esattamente, dopo il cambio pagina, il framework comanderà di nuovo un redirect verso la pagina di login.

Questo comportamento è pilotato dalla voce "***Forzare il redirect al login?***" del menù Impostazioni/Setup/Sicurezza che di default è impostata a "Sì". Impostandola a "No" il comportamento sarà quello tipico di Ember (potremo cioè visualizzare qualunque pagina, a meno delle eventuali restrizioni introdotte con il plugin ember-simple-auth) ma questa opzione è applicabile ai soli siti *vetrina*, perché non fornisce un livello di sicurezza adeguato alle applicazioni più *complesse*.

Quando il valore è impostato a "Sì" abbiamo comunque la possibilità di fornire accesso agli *unlogged* a specifiche pagine. A tal scopo dobbiamo fornire al framework un "***elenco delle pagine ammesse***" editando il file **routers/application.js** (che come noto è uno dei file più importanti di un'applicazione Ember) ed aggiungendo all'array ***unloggedEnabledPages*** le rotte che vogliamo rendere accessibili (senza eliminare quelle già presenti, necessarie al funzionamento del framework).
NOTA: è previsto lo spostamento della suddetta variabile all'interno del file config/environment.js in una prossima release del framework.

Pur avendo inserito le nostre rotte, noteremo che al tentativo di accedervi via url avverrà di nuovo il redirect verso la pagina di login. In effetti il framework prevede un'unica pagina di accesso per gli utenti non loggati, che di default è la pagina login.
Le altre *pagine ammesse* sono visitabili solo attraverso i link inseriti con il componente *LinkTo* nella pagina di login.
In altre parole le *pagine ammesse* non sono raggiungibili direttamente via url, ma sono accessibili dalla pagina di login cliccando i link di Ember. E in ciascuna *pagina ammessa* possiamo inserire altri *LinkTo*.

Il framework offre pure la possibilità di modificare la pagina di accesso di default per gli utenti non loggati. Se vogliamo che questi atterrino su una pagina diversa da login, torniamo nel file **routers/application.js** e individuiamo il seguente codice:

routers/application.js

```
{
   // In ambiente Cordova (cioè se il codice gira in un'App) ed utente non loggato, redi
a 'login' o a 'slider' in base alle configurazioni
   this.session.requireAuthentication(transition, 'welcome-slider');
} else {
   this.session.requireAuthentication(transition, 'login');
}
```

modifichiamo la seconda *session.requireAuthentication* sostituendo “login” con la rotta di nostro interesse.

Sebbene per l’ambiente Web il capitolo sia completo, per l’ambiente Cordova dobbiamo fare un ulteriore approfondimento.
Se apriamo il menù Impostazioni/Setup/Sliders notiamo l’opzione “***Mostrare lo slider di benvenuto?***” diversificata per App e per Web. Se l’opzione è impostata su “No”, il comportamento dell’App è identico a quello Web, ovvero gli utenti non loggati verranno rediretti alla pagina principale (*login*, se non l’ abbiamo sostituita). Se impostiamo l’opzione a “Sì” il redirect sull’App avverrà verso la rotta *welcome-slider*, che mostra uno slider di benvenuto e le cui foto possono essere gestite da interfaccia sempre dal menù Impostazioni/Setup/Sliders.

REGOLE GENERALI

In questa sezione sono fornite alcune regole generali atte a standardizzare la scrittura del codice.

COMPONENTI

- Il codice HBS del componente deve essere sempre contenuto in un avente una classe che si chiama come il componente:

my-component.hbs

```
<div class="my-component" ...attributes>
   <!-- contenuto HBS del componente -->
</div>
```

- se dobbiamo creare un foglio di stile dedicato, questo si chiamerà come il componente, avrà estensione *scss* e lo inseriremo in styles/components.
  Linkiamo poi il suddetto file all'interno del file **styles/app.scss**, alla fine della sezione dedicata ai componenti custom.
  Il foglio di stile va creato solo se strettamente necessario, ma è preferibile utilizzare le classi di stile del tema per non appesantire la build.
  Il foglio di stile inizierà con la definizione della classe principale del componente e tutti gli altri stili saranno in essa contenuti, al fine di evitare che lo stile definito in questo file possa interferire con gli altri elementi del sito:

styles/components/my-component.scss

```
.my-component {
   /* tutti gli stili vanno definiti qui! */
}
```

- se dobbiamo assegnare un ID a un componente, è necessario ricordare che l'applicazione Ember è una SPA, quindi l'id deve essere unico fra tutti i componenti e template. Una buona norma per garantirne l'unicità è definire un nome complesso la cui radice sia il nome del componente (o del template) in cui è inserito. Nel caso d'esempio, se volessimo definire l'id per un tag *input* per la digitazione di un nome, potremmo usare "*my-component-input-name*".
- è buona norma gestire il ciclo di vita del componente. Ipotizziamo di avere un componente al cui interno abbiamo definito un *listener*. Al momento della distruzione del componente vorremmo che il *listener* venga eliminato, al fine di liberare la memoria occupata. Useremo quindi il metodo **willDestroy** (richiamato da Ember subito prima che il componente venga distrutto) al cui interno inseriremo tutte le necessarie operazioni di *pulizia*.

Metodo willDestroy di un componente

```
willDestroy() {
   super.willDestroy(...arguments);
   // qui inseriamo il codice di pulizia!
}
```

- Una buona abitudine per migliorare la UX è quella di inserire dei loader che segnalino la fase di inizializzazione (mitigando la percezione di attesa e informando l'utente sul processo in corso) e degli avvisi che indichino eventuali errori verificatisi all'avvio del componente (e lo stesso discorso può essere esteso ai controller).
  È importante inoltre mantenere uno stile omogeneo della UX per tutti i componenti.
  A tal scopo nella quasi totalità dei componenti troverete un codice ricorrente che riportiamo qui per spiegarne il funzionamento e che potrete usare come base di partenza per i vostri sviluppi.

Estratto di codice HBS di partenza per i vostri componenti
ricordare di inserire il codice dentro a un div la cui classe si chiami come il componente!

```
{{!-- WAITING --}}
{{#if (eq this.serviceAvailable 'waiting')}}
 <Standard::LoadingSpinner @icon="true" @msg="Attendere prego..." @style="3"
class="alert-info" />
{{/if}}

{{!-- UNAVAILABLE --}}
{{#if (eq this.serviceAvailable 'unavailable')}}
 <Standard::LoadingSpinner @icon="" @style="3" class="alert-danger">
   {{this.error}} <a href="#" class="ml-2" {{on 'click' this.start}}>Riprova.</a
 </Standard::LoadingSpinner>
{{/if}}

{{!-- AVAILABLE --}}
{{#if (eq this.serviceAvailable 'available')}}
   {{!-- il vostro codice qui --}}
{{/if}}
```

Estratto del codice Javascript di partenza per i vostri componenti

```
@tracked serviceAvailable = 'waiting';

constructor(...attributes) {
 super(...attributes);
 this.start();
}

@action
async start() {
 try {
   this.serviceAvailable = 'waiting';
   // il codice di inizializzazione qui
   // ...
   this.serviceAvailable = 'available';
 } catch (e) {
   console.error(e);
   this.serviceAvailable = 'unavailable';
 }
}
```

Molto banalmente, definiamo una variabile @tracked chiamata *serviceAvailable* e con valore "waiting". L'HBS mostrerà un *Loading Spinner* di attesa. Il costruttore richiama il metodo *start* che si occupa dell'inizializzazione. All'interno di questo metodo inseriremo il codice di avvio del componente (ad esempio l'estrazione dal DB di un elenco di record). Notiamo che il metodo è definito come *async*, per consentire l'utilizzo di *await* al suo interno ed è decorato con @action per essere richiamabile dall'HBS.

Un try-catch al suo interno si occupa di valorizzare a “unavailable” la variabile *serviceAvailable* in caso di errori, comportando la comparsa del *Loading Spinner* di errore. All’interno di questo è presente un richiamo al metodo *start*.
Se l’inizializzazione avviene senza errori, alla fine del metodo *start* valorizziamo la variabile tracked ad “available”, consentendo così di visualizzare l’interfaccia vera e propria del componente.

CONTROLLER

Se non strettamente necessario, evitiamo di scrivere controller per definire il Javascript da associare ai template. È preferibile creare un componente in modo da rendere il codice riutilizzabile (affronteremo un caso di eccezione a questa regola quando parleremo dei *query parameters*).

TEMPLATE

Il codice dei nostri template è strettamente vincolato alla struttura del tema utilizzato, Architect UI (basato su bootstrap 4). È importante conoscere a fondo la struttura del tema ed utilizzare il più possibile gli elementi grafici in esso presenti.

La sezione principale del tema, comune sia per le pagine dedicate agli utenti non loggati che per quelle dedicate agli utenti loggati, è inglobata nel framework e non modificabile. Stiamo cioè parlando della sezione head e di una parte della sezione body.
Pertanto, nella creazione dei template dovremo distinguere due tipologie di pagine:

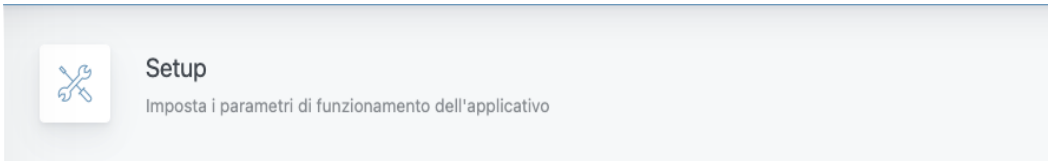
- **Pagine per utenti non loggati**
  Usano una struttura html dedicata, visibile ad esempio nella pagina di login (templates/login.hbs) o di registrazione (templates/registration.hbs). È indispensabile prendere spunto dal codice in esse presenti ed apportare eventuali modifiche con attenzione, per non alterare la visibilità e le funzionalità sia in ambiente web che, soprattutto, in ambiente mobile.
  Ricordare sempre che la visibilità delle pagine è soggetta alle impostazioni configurate nel menù di Setup e può essere diversa tra Web o App.
- **Pagine per utenti loggati**
  In questo caso abbiamo molta più flessibilità ed il framework ci offre componenti che velocizzano la creazione standardizzata dei template.
  Salvo casi molto particolari, ogni template inizierà con questo componente:

componente app-page-title

```
<ArchitectUi::AppMain::AppPageTitle
   @title="Setup"
   @titleKey="template.setup.title"
   @description="Imposta i parametri di funzionamento dell'applicativo"
   @descriptionKey="template.setup.subTitle"
   @icon="pe-7s-tools"
/>
```

(la descrizione dei parametri è disponibile nel file Javascript del componente stesso; sorvoliamo per ora i parametri di traduzione di cui parleremo più avanti).

Produce questo risultato:



Se vogliamo aggiungere contenuti all'interno dell'intestazione, ad esempio la *page-title-actions* del tema, basterà eliminare la "*self-closing tag*" (vale a dire la terminazione /> da sostituire con >), aggiungere il codice di nostro interesse e terminare con il tag di chiusura.

- se dobbiamo creare un foglio di stile dedicato, questo si chiamerà come il template, avrà estensione *scss* e va inserito in styles/templates.
  Linkiamo poi il suddetto file all'interno del file styles/app.scss, alla fine della sezione dedicata ai template custom.
  Il foglio di stile va creato solo se strettamente necessario, ma è preferibile utilizzare le classi di stile del tema per non appesantire la build.
- La regola generale da tenere sempre a mente è "**mobile first**". Nella progettazione dei nostri template useremo Chrome per visualizzarne il risultato prima in modalità *mobile*, poi in modalità desktop; questo è essenziale per garantire la fruibilità del sito da dispositivi mobili e per evitare l'onere di ristrutturare i template in futuro, come nell'eventualità in cui Cliente richieda la versione App dell'applicativo.
- Quando usiamo le griglie di bootstrap dobbiamo sempre definire, per gli elementi che la compongono, almeno le dimensioni su *mobile* e su desktop, in questo ordine. Ad esempio "col-12 col-md-6" (e non "col-md-6 col-12"). Se le dimensioni fossero uguali per app e desktop, ci limiteremo a specificare quelle del *mobile* (es "col-12").
- Il testo dei pulsanti è sempre maiuscolo e le relative classi di stile vanno inserite nell'ordine corretto (nel codice del tema Architect originale, purtroppo, non è stata prestata attenzione a questo aspetto). Ad esempio scriveremo:

```
<button class="btn btn-sm btn-secondary btn-hover-shine" type="button">
   <i class="fa fa-search mr-2"></i> CLICK
</button>
```

Abbiamo cioè inserito prima la classe bootstrap dei pulsanti (btn), poi l'eventuale classe dimensionale (btn-sm), poi le classi di stile (btn-secondary e btn-hover-shine).
Notare inoltre che l'icona presenta la classe "mr-2" per separarla correttamente dal testo.
Elimineremo tale classe di margine qualora il pulsante abbia anche la classe "btn-icon-only".

MODEL

Per convenzione, le eventuali relazioni nel controller vanno aggiunte prima degli attributi.
Per ciascun attributo è conveniente aggiungere un commento riassumente il significato ed eventuali enumerazioni (valori ammessi), informazioni che in futuro semplificheranno il lavoro di manutenzione.

ROUTERS

Quando viene generata una rotta dedicata agli utenti loggati, questa va immediatamente inserita nel menù Impostazioni/Setup/Rotte, specificando anche un titolo e una descrizione. Questo passaggio è essenziale per poter poi definirne i permessi di accesso (come vedremo più avanti).

Se una pagina è dedicata ai soli utenti loggati, o ai soli utenti non loggati, nel corrispondente gestore di rotta va sempre inserito il codice che filtra gli accessi.

accesso per soli utenti loggati

```
@service session;

async beforeModel(transition) {
   await this.session.setup();
   this.session.requireAuthentication(transition, 'login'); // solo utenti loggati
}
```

accesso per soli utenti non loggati

```
@service session;

async beforeModel() {
   await this.session.setup();
   this.session.prohibitAuthentication(''); // solo utenti non loggati
}
```

ROUTER.JS

Quando generiamo una rotta con il comando *ember g route …*, questa viene automaticamente inserita alla fine del file router.js. Per fare ordine e rendere il file più leggibile, andremo a spostare la riga creata dalla ember-cli alla fine della sezione “NOT LOGGED USERS”, se la rotta è dedicata agli utenti non loggati, oppure alla fine della sezione “NOT LOGGED AND LOGGED USERS” se la rotta è disponibile per tutti gli utenti oppure alla fine della sezione ”LOGGED USERS” se la rotta è dedicata ai soli utenti loggati.
Fare attenzione a non eliminare le rotte già presenti per non compromettere il funzionamento del framework.

REGOLE COMUNI

- Sia nei file Javascript che negli HBS, è buona norma suddividere il codice per aree funzionali mediante il plugin ***region marker*** di Visual Studio Code, soprattutto in file particolarmente lunghi e/o articolati.
- Quando una parte del codice viene lasciata temporaneamente incompleta, aggiungere sempre il commento **TODO**, sia nei file Javascript che negli HBS. Prima del rilascio è buona norma cercare tale commento nell’intera soluzione per scovare eventuali dimenticanze.
- È essenziale **commentare** sempre il codice scritto, per consentire ad altri sviluppatori di comprendere più facilmente le logiche su cui si basa il codice scritto. I commenti devono essere sintetici, funzionali e non superflui.
- Si richiede che tutti i metodi implementati siano descritti con **JSDoc** (plugin di Visual Studio Code elencato nei requisiti).
  Dopo aver scritto la firma del metodo, basta digitare ***/**** nella riga precedente al metodo e automaticamente l’intellisense di JSDoc produce questo risultato, che:
  - indica cosa fa il metodo,
  - specifica i tipi di parametri e la loro funzione
  - indica i valori di ritorno ecc:

```
/**
```

```
* Calcola la somma di due numeri.
* @param {number} a Il primo numero.
* @param {number} b Il secondo numero.
* @returns {number} La somma di a e b.
*/
function somma(a, b) {
   return a + b;
}
```

SEGNALATORI

Nel framework front-end sono disponibili 4 strumenti di segnalazione, 3 dei quali disponibili mediante il servizio custom Dialogs.

TOAST

Sono notifiche toast di avviso, prive di pulsanti di azione. Fanno parte del servizio Dialogs e derivano dalla libreria alertify.js.
Per aggiungere un toast dobbiamo anzitutto importare il servizio mediante @service dialogs; e poi utilizzare il codice seguente:

Uso di dialogs.toast

```
this.dialogs.toast(
   `Benvenuto`,
   'success',
   'top-right',
   3
);
```

I parametri sono:

- message : stringa contenente il messaggio da mostrare (obbligatorio)
- type : stabilisce la colorazione. Possibili valori: message (bianco), success (verde), warning (arancio), error (rosso)
- position : stabilisce la posizione (top-left/center/right o bottom-left/center/right)
- delay : durata del tooltip in secondi. Se 0, rimane fino al click dell'utente
- close_cb : eventuale callback da eseguire alla chiusura del tooltip

ALERT

Sostituisce l'alert nativo del browser. Fanno parte del servizio Dialogs e derivano dalla libreria alertify.js.
Per aggiungere un alert dobbiamo anzitutto importare il servizio mediante @service dialogs; e poi utilizzare il codice seguente:

Uso di dialogs.alert

```
this.dialogs.alert(
   `<h6 class="text-danger">titolo</h6>`,
   `messaggio`,
   'Ok'
);
```

I parametri sono:

- title : stringa contenente il titolo della finestra di alert, anche in formato html
- message : stringa contenente il messaggio da mostrare, anche in formato html (obbligatorio)
- label : stringa contenente testo del pulsante
- cb : eventuale callback invocata al click dell'utente sul pulsante di conferma

CONFIRM

Sono finestre modali a due pulsanti. Fanno parte del servizio Dialogs e derivano dalla libreria alertify.js.
Per aggiungere una confirm dobbiamo anzitutto importare il servizio mediante @service dialogs; e poi utilizzare il codice seguente:

Uso di dialogs.confirm

```
this.dialogs.confirm(
      `<h6>titolo</h6>`,
      `<span>messaggio</span>`,
      () => {
        this.saveNewRecordConfirmed.perform();
      },
      null,
      ['Conferma', 'Annulla']
);
```

I parametri sono:

- title : stringa contenente il titolo della finestra di alert, anche in formato html
- message : stringa contenente il messaggio da mostrare, anche in formato html (obbligatorio)
- ok_cb : callback eseguita alla pressione del tasto di conferma
- cancel_cb : callback eseguita alla pressione del tasto di annullamento
- labels : array di stringhe contenente, nell'ordine, il testo del pulsante di conferma e di quello di annullamento
- autoExec : intero opzionale. Se >0, indica il tempo in secondi dopo cui verrà automaticamente premuto il pulsante di conferma. Se <0, indica il tempo in secondi di auto-cancellazione. Viene mostrato un contatore.

SWAL

Sono modali altamente personalizzabili, realizzate con sweet-alert. Non è necessario alcun servizio per utilizzarle. Si rimanda alla documentazione ufficiale per una comprensione esaustiva.

Uso di Swal

```
Swal.fire({
   title: `Titolo`,
   html: `Messaggio`,
   icon: 'success',
   showCancelButton: false,
});
```

Generalmente utilizziamo:

- toast per segnalare warning nell'immissione dati da parte dell'utente o altre segnalazioni (anche di successo) non invasive;
- alert per segnalazioni più importanti;
- confirm per consentire all'utente di confermare/rifiutare un'azione;
- swal per comunicazioni importanti e d'effetto.

ELEMENTI GIA' PRONTI

I COMPONENTI DI UTILITA'

Normalmente non dovremo mai modificare il codice dei componenti fondamentali, ovvero di quelli in components/standard/core/, sui quali si basa il framework e ai quali raramente faremo ricorso. Faremo invece un grande uso dei componenti in components/standard, molti dei quali rendono disponibili per Ember alcune librerie jQuery-based. Essendo ampiamente documentati, ci limiteremo ad elencarli:

- back-button: come dice il nome stesso è un componente con cui inserire negli HBS il pulsante "torna indietro", ottimizzato per lavorare con la *history* di Ember.
- ck-editor: un potentissimo componente che consente di creare un editor di testo visuale, basato sull'omonima libreria. È un esempio di componente *wrapper*.
- dev-ext-grid: wrapper per il plugin dataGrid della libreria DevExtreme.
- flip-switch: consente di inserire e di gestire con Ember i flip-switch di Bootstrap.
- loading-spinner: un utile componente per inserire dei loader customizzabili.
- pagination-button: probabilmente uno dei componenti più utili ed utilizzati nel framework, consente di mostrare i pulsanti di navigazione tra record paginati.
- select-two: un *wrapper* per Ember della popolare libreria Select2.
- show-image: ne parleremo in dettaglio più avanti, è un componente smart per inserire le immagini nei nostri HBS.
- t-r: è un componente dedicato al multi lingua di cui parleremo dettagliatamente più avanti.
- textarea-autosize: come dice il nome stesso, consente di inserire delle textarea la cui altezza varia dinamicamente in base al contenuto.
- tool-tip: consente di aggiungere i tooltip di Bootstrap previsti dal tema

GLI HELPER

Nella cartella helpers sono stati creati degli elementi custom di sicura utilità. Per quanto sia possibile aggiungerne altri, questa eventualità dovrebbe essere valutata collettivamente per evitare di sovraccaricare il framework con elementi di dubbia riusabilità, anche considerando la possibilità di definire helper locali (come imparato dalle guide di Ember).
Vediamo dunque gli helper più utili.

CONVERTDATE2

Un helper che consente di visualizzare, nella *lingua* dell'utente, un campo *data* restituito (ad esempio) da una chiamata API. Consente inoltre di impostare il formato desiderato.

CONVERTITIME

Come il precedente ma visualizza gli orari, occupandosi della conversione nel fuso orario dell'utente.

GET-PROP

Accetta un oggetto e il nome di una sua proprietà, restituendone il valore.

INCLUDE

Verifica l’esistenza di un valore in un array.

LENGTH

Restituisce la lunghezza della variabile fornitagli.

NUMBER-FORMAT

Formatta i numeri con i separatori previsti dall’internazionalizzazione.

SANITIZE-FORMAT

Uno degli helper più utili in assoluto, per massimizzare la **sicurezza** quando si vogliono mostrare a schermo i contenuti provenienti da fonti non sicure, come eventuali testi inseriti dagli utenti nei form di immissione dati. Esegue la sanificazione del contenuto HTML e poi lo marca come sicuro per essere utilizzato in un template.

TYPEOF

Ritorna il tipo di una variabile.

UPPERCASE

Restituisce una stringa convertita in maiuscolo.

MODELLI E RELAZIONI

Sebbene la direttive di creazione di modelli e relazioni dovrebbero essere già state acquisite durante lo studio delle guide Ember, conviene sottolineare alcuni aspetti di primaria importanza ed altre accortezze che derivano dall'esperienza.

STANDARD PER I MODELLI

Analizziamo anzitutto i 4 step necessari nella definizione di nuovi modelli:

1. generare il modello con ember-cli
    - il nome è al singolare e in kebab-case. Es: *ember g model custom-setup*
2. popolare il file del modello con gli attributi e le relazioni desiderate
    - le relazione belongsTo sono definite al **singolare**
    - le relazioni hasMany sono definite al **plurale**
    - consultare Swagger per accertarsi che le relazioni lato back-end siano definite secondo le convenzioni standard
3. aprire il file initializers/custom-inflector-rules ed aggiungere il **correttore di rotta**
    - quando il back-end non rispecchia le convenzioni standard nella definizione degli end-point è necessario aggiungere un correttore di rotta nel custom-inflector-rules.js che trasforma la rotta standard definita da Ember in quella fuori standard definita nel back-end.
4. definire l'entità nel menù Impostazioni/Setup del framework
    - definire un titolo e una descrizione per le nuove entità aiuta gli operatori che devono occuparsi dei permessi di CRUD a comprendere con più facilità la funzionalità dell'entità stessa.

Va inoltre ricordato che il valore di default degli attributi è generalmente utile e ci fornisce una valorizzazione al momento della creazione di un nuovo record. Tuttavia ci sono dei casi in cui il valore di default non deve essere specificato.
È il caso in cui nel nostro DB abbiamo un campo *nullable* con una relazione ad un'altra entità. Se fornissimo un valore di default non corrispondente ad un record esistente nell'altra entità, al momento del salvataggio otterremo ovviamente un'eccezione lato server.

ORDINE DEGLI INFLECTOR

Supponiamo di avere due modelli chiamati **custom-setup** e **ticket-custom-setup**. Nell'estrazione dei dati dal DB, Ember genera automaticamente gli indirizzi degli end-point seguendo la convenzione delle rotte pluralizzate e dasherizzate, ovvero punterà agli end-point **custom-setups** e **ticket-custom-setups**. Il back-end prevede però che i nomi degli end-point siano **customSetups** e **ticketCustomSetups**; siamo pertanto costretti a sovrascrivere lo standard definendo dei correttori di rotta. Ipotizziamo quindi di inserire le seguenti due righe nel custom-inflector-rules:

custom-inflector-rules.js - Versione errata

```
inflector.irregular('custom-setup', 'customSetups');
inflector.irregular('ticket-custom-setup', 'ticketCustomSetups');
```

Seppure il codice é formalmente corretto, otterremo un errore nel tentativo di estrazione dei dati.
L'errore deriva dall'ordine con cui sono stati definiti gli inflector.

Analizziamo meglio l'errore. Nel momento in cui andiamo a richiamare un record con ember-data, Ember analizza il custom-inflector-rules.js ed applica le correzioni di rotta disponibili.

Se stiamo richiamando un record del modello ticket-custom-setup, la rotta generata con l'applicazione degli inflector sarà *ticketcustomSetups*, con la lettera C di custom scritta in minuscolo, che non corrisponde all'end-point del backend.

Questo avviene perché il nome del primo modello è incluso nel secondo e verranno quindi applicate entrambe le trasformazioni.

Per risolvere il problema è sufficiente invertire l'ordine dei due inflector.

custom-inflector-rules.js - Versione corretta

```
inflector.irregular('ticket-custom-setup', 'ticketCustomSetups');
inflector.irregular('custom-setup', 'customSetups');
```

Dalla lettura del codice del custom-inflector-rules.js l'individuazione di questa problematica potrebbe non essere immediata (specie in presenza di molti inflector). Tuttavia quando si verificano errori nella creazione delle rotte, conviene analizzare il file alla ricerca dell'eventualità qui descritta.

CONVENZIONI PER GLI STORE METHODS

Quando utilizziamo i metodi dello Store (come findRecord, findAll, query, queryRecord, ecc.) occorre ricordare che il formato standard è quello dasherizzato. Se abbiamo quindi un modello custom-setup, useremo custom-setup anche nei metodi dello Store.

metodo corretto per gli Store Methods

```
this.store.findAll('custom-setup'); // e non customSetup
```

METODI PER SINGOLI RECORD E PER COLLEZIONI

Un errore comune quando si inizia ad utilizzare ember-data è quello di confondere il comportamento di metodi come queryRecord e query, o di findRecord e findAll o di peekRecord e peekAll.
I metodi che contengono la parola *Record* nel loro nome servono per l'estrazione di un singolo dato.
Gli altri servono per l'estrazione di collezioni (array) di record.

Ipotizziamo di avere questo codice:

```
@tracked record;
// …
```

```
this.record = this.store.query('user', {
   filter: `equals(id,'1')`,
});
```

Sorvoliamo l'errore formale (cioè quello di avere scritto record al singolare pur usando store.query), quello che ci aspettiamo come risultato è un singolo record, poiché lo stiamo richiamando tramite la chiave univoca id. Tuttavia, avendo utilizzato *query* anziché *queryRecord*, quella che otteniamo è una collezione di record, seppur di lunghezza unitaria.
Di conseguenza il seguente codice HBS:

Codice *errato* nel contesto corrente

```
L'email dell'utente è {{this.record.email}}
```

non produrrà l'effetto desiderato. Funzionerebbe invece correttamente se avessimo utilizzato queryRecord.

Le collezioni possono essere utilizzate negli HBS mediante {{#each}}

Codice funzionante

```
{{#each this.record as | rec |}}
   L'email dell'utente è {{rec.email}}
{{/each}}
```

fermo restando che avremmo dovuto definire la variabile record al plurale.

AWAIT PER GLI STORE METHODS

Tutti gli store methods restituiscono una Promessa.
Esaminiamo più attentamente un codice già visto in precedenza:

codice formalmente errato

```
@tracked record;

this.record = this.store.query('user', {
  filter: `equals(id,'1')`,
});
```

Questo codice è formalmente errato, seppur funzionante.
Il motivo è che tutti gli store methods ritornano una Promessa, ovvero i dati non sono immediatamente disponibili. Potremmo non accorgerci dell'errore perché la variabile record è definita come tracked e pertanto, una volta che la promessa si risolve, l'HBS aggiorna la visualizzazione e ci mostra i dati richiesti.
Ma ipotizziamo di modificare il codice come segue:

altro codice errato

```
@tracked records;
```

```
let rec = this.store.query('user', {
  filter: `equals(id,'1')`,
});

if (rec.length > 0) {
   this.records = rec;
}
```

In questo snippet stiamo cercando di interagire con i dati (rec.length). Questo produrrà un errore in console, appunto perché non abbiamo gestito il fatto che gli store methods sono *Promise*.

Per correggere l'errore e gestire correttamente i metodi dello store, basterà modificare il codice come segue:

codice corretto

```
@tracked records;

let records = await this.store.query('user', {
  filter: `equals(id,'1')`,
});
```

in altre parole bisogna sempre usare **await** quando si usano gli store methods.

USO CORRETTO DEI METODI PEEK

Per massimizzare la velocità del sito percepita dall'utente, è importante evitare chiamate API inutili. Ipotizziamo di avere un componente che mostra una lista di record, recuperata con ember-data, in cui selezionare un record per effettuarne la modifica. Al momento della selezione saremo reindirizzati a una pagina di modifica che riceve l'id del record scelto.
In questa pagina dovremo recuperare i dati dal DB e sicuramente potremo farlo con *store.findRecord*. Tuttavia i dati sono già presenti nella cache di ember-data poiché estratti nella pagina precedente. In questi casi conviene dunque utilizzare il metodo *peekRecord* che ci fornisce il dato senza effettuare chiamate API. Avremo poi l'accortezza di verificare che il dato esista realmente:

Uso corretto del metodo peekRecord

```
// Tenta prima di recuperare il record dalla cache locale
let record = this.store.peekRecord('model-name', id);

if (!record) {
   // Altrimenti, carica il record dal server
   record = this.store.findRecord('model-name', id);
}
```

ASINCRONIA DELLE RELAZIONI

Nei nostri modelli avremo sicuramente delle relazioni *belongsTo* e/o *hasMany*. Vediamone un esempio:

```
export default class TicketAreaModel extends Model {
 @hasMany('ticket-operator', { async: true, inverse: 'ticketArea' }) ticketOperator;

 …
}
```

Questo codice specifica che il modello ticket-area ha una relazione uno a molti con il modello ticket-operator e che la relazione è caricata in modo asincrono. Ciò significa che ember-data non caricherà immediatamente i dati della relazione quando carica il record principale. Invece, la relazione sarà caricata "on demand", ovvero solo quando si accede esplicitamente alla relazione nel codice.

```
let ticketArea = await this.store.findRecord('ticket-area', 1);
let tenantDestination = await ticketArea.tenantDestination;
```

Nel codice precedente attendiamo l'estrazione del record ticket-area. Poichè la relazione è asincrona, Ember non caricherà immediatamente il record tenant-destination. Lo farà in corrispondenza della seconda riga, in cui usiamo await per attendere che il dato sia disponibile.

Definire asincrona una relazione ha due vantaggi:

- Riduce la quantità di dati caricati inizialmente, poiché le relazioni non vengono caricate finché non sono esplicitamente richieste.
- Può migliorare i tempi di caricamento del modello principale se le relazioni sono pesanti o non sempre necessarie.

Se nel modello impostiamo *async: false*, ember-data cercherà di caricare i dati della relazione nello stesso momento in cui carica il record principale. In questo caso non occorre utilizzare await per estrarre il dato relazionato.

Definire sincrona una relazione ha due vantaggi:

- Fornisce un accesso immediato ai dati della relazione senza necessità di aspettare ulteriori caricamenti asincroni.
- Utile quando si sa che le relazioni sono sempre necessarie e si vogliono evitare ritardi nel loro accesso.

Per quanto generalmente sia preferibile la linea asincrona, la scelta dell'asincronismo tra le relazioni dipende dalla natura dell'applicazione, dalla frequenza con cui le relazioni sono effettivamente utilizzate e dal peso del caricamento di tali relazioni. Impostare relazioni non necessarie come asincrone può migliorare significativamente le prestazioni percepite dell'applicazione, mentre definire sincrone delle relazioni importanti e frequentemente usate può semplificare il codice e ridurre la complessità nella gestione degli stati di caricamento.

Quando cerchiamo di accedere a relazioni non disponibili, specie relative a query con *include* profondamente nidificati, Ember potrebbe segnalarci mediante un errore la necessità di usare get() per accedere al dato.

TRASFORMATORI

Nella definizione dei Model potete utilizzare tre trasformatori custom che semplificano notevolmente la gestione e lo scambio dati con il server.
Essendo trasformatori custom, non esistono nativamente in Ember e si aggiungono a quelli di default (string, number, date, boolean); sono:

- arrays
- objects
- date-utc

Per analizzarli, ipotizziamo di avere un model che li contenga tutti e tre:

Esempio di model con i 3 trasformatori custom

```
export default class exampleModel extends Model {
    @attr('objects', { defaultValue: () => {} }) param1;
    @attr('arrays', { defaultValue: () => [] }) param2;
    @attr('date-utc', {
        defaultValue() {
            return new Date();
        },
    })
    param3;
}
```

TRASFORMATORE ARRAYS

Ci consente di trattare localmente gli array, risparmiandoci l'onere di serializzarli in stringhe quando scriviamo sul db e deserializzati nuovamente in array quando leggiamo dal db. Lato server il campo corrispondente sarà di tipo stringa.
Potremo quindi assegnare un array a *param1* senza occuparci della sua conversione:

Il trasformatore array ci risparmia la serializzazione

```
let d = this.store.createRecord('example', {
  param1 = [`A`, 'B'];
});
```

E potremo leggere direttamente un array senza occuparci delle deserializzazione:

Il trasformatore array ci risparmia la deserializzazione

```
let d = this.store.queryRecord('example', 1);
let firstElement = d.param1[0];
```

TRASFORMATORE OBJECTS

Come Arrays, ma lavora con gli oggetti.

TRASFORMATORE DATE-UTC

Consente di inviare facilmente orari UTC al server, così da gestire gli orari multi fuso orario.
Ipotizziamo di avere un campo input di tipo *datetime-local* con cui l'utente inserisce una data nel suo fuso orario locale, valorizzando una variabile *newAppointment.date*.

```
let localDateObject =
    new Date(this.newAppointment.date); // Creiamo un oggetto Date a partire
                                           dalla data locale inserita
var now_utc = Date.UTC(
   localDateObject.getUTCFullYear(),
   localDateObject.getUTCMonth(),
   localDateObject.getUTCDate(),
   localDateObject.getUTCHours(),
   localDateObject.getUTCMinutes(),
   localDateObject.getUTCSeconds()
);

let d = this.store.createRecord('example', {
   param3 = new Date(now_utc).toISOString();
});
```

In questo modo invieremo al server una data in formato UTC.

Inoltre per semplificare la procedura, nelle ultime versioni del framework è stata aggiunta al servizio js-utility un'apposita funzione *getUTC* che riduce il codice dell'esempio precedente.

Utilizzo del metodo *getUTC* di js-utility per adattare la data locale al formato del trasformatore date-utc

```
let d = this.store.createRecord('example', {
  param3 = this.jsUtility.getUTC(this.newAppointment.date);
});
```

OBSERVABLE ARRAY/OBJECT

Conosciamo ormai bene il significato del decoratore @tracked ed abbiamo compreso che alla variazione di una variabile tracked nel Javascript corrisponde un re-rendering dell'HBS che contiene quella variabile.
Questo è vero solo se le variabili sono stringhe, numeriche e booleane ma **non** funziona per oggetti ed array.

Per risolvere questo *inconveniente* è stato installato nel framework l'apposito plugin ***tracked-built-ins***.
Con esso potremo definire un oggetto (o un array) le cui proprietà sono a loro volta tracked.

Per utilizzarlo dovremo importare dal plugin *l'utility* che ci interessa:

```
import { TrackedObject } from 'tracked-built-ins';
// oppure
import { TrackedArray } from 'tracked-built-ins';
```

Forniamo un semplice esempio per dimostrarne il funzionamento, immaginando di voler creare un form per l'inserimento di nuovi record. Non ci occuperemo del salvataggio del dato, quello che ci interessa è verificare che le proprietà di un *trackedObject* siano realmente tracked.

Ipotizziamo dunque di aver il seguente modello, con relativi valori di default:

```
export default class GenericModel extends Model {
 @attr('string', { defaultValue: 'Default String' }) stringAttribute;

 @attr('number', { defaultValue: 42 }) integerAttribute;

 @attr('date', {
   defaultValue() {
     return new Date();
   }
 }) dateAttribute;
}
```

Definiamo poi il javascript del componente:

```
export default class DataEntryFormComponent extends Component {
 @tracked record;

 constructor() {
   super(...arguments);
   let modelInstance = this.store.createRecord('generic');

   // Crea un semplice oggetto JavaScript con tutte le proprietà del modello
   let modelData = {};
   modelInstance.eachAttribute((key) => {
     modelData[key] = modelInstance.get(key);
   });

   // Crea un nuovo TrackedObject da modelData
   this.record = new TrackedObject(modelData);
 }
}
```

Semplicemente abbiamo creato un record del modello "generic" ed abbiamo poi creato un oggetto javascript con tutte le proprietà del modello e relativi valori di default. Per intenderci *modelData.integerAttribute* avrà valore 42. Successivamente abbiamo creato un *trackedObject* a partire dall'oggetto.

Ora nell'HBS:

```
<form>
 <div>
   <label for="stringAttribute">String Attribute:</label>
   <input type="text" id="stringAttribute" value={{this.record.stringAttribute}}/>
```

```
  </div>
  <div>
    <label for="integerAttribute">Integer Attribute:</label>
    <input type="number" id="integerAttribute" value={{this.record.integerAttribute}}/>
  </div>
  <div>
    <label for="dateAttribute">Date Attribute:</label>
    <input type="date" id="dateAttribute" value={{this.record.dateAttribute}}/>
  </div>
</form>
```

Verificheremo che, pur essendo la variabile *record* di tipo oggetto, i campi input avranno il valore di default valorizzato.

UTILITY

Il framework ci mette a disposizione diverse utility (alcune native di Ember, altre aggiunte) che ci torneranno utili durante lo sviluppo. Ecco un elenco non esaustivo delle più utilizzate nel framework.

GUID

Se abbiamo bisogno di generare un guid, basta importare:

```
import { v4 } from 'ember-uuid';
```

ed utilizzarlo come segue:

```
let guid = v4();
```

CAMELIZE

Se vogliamo convertire una stringa in camel-case:

```
import { camelize } from '@ember/string';
......
/* all'interno della classe di un componente o di un controller */
let code = camelize(`Stringa da convertire`);
```

ENV e CONFIG

Ember ci consente di predisporre tre ambienti diversi (*development*, *test*, *production*) quindi può tornare utile prelevare delle variabili dal file config.js che variano in base all'ambiente. Come pure potremmo voler leggere i valori di alcune variabili che pur non dipendendo dall'ambiente, abbiamo definito per comodità nel config.js.

Importare una variabile che non dipende dall'ambiente

```
import config from 'poc-nuovo-fwk/config/environment';
......
/* all'interno della classe di un componente o controller */
let host = config.namespaceHost;
```

Importare una variabile che dipende dall'ambiente

```
import ENV from 'poc-nuovo-fwk/config/environment';
......
/* all'interno della classe di un componente o controller */
let apiHostDev = ENV.apiHostDev;
```

QUERY COMPLESSE IN JSON:API

INTRODUZIONE

Se dobbiamo realizzare una query di estrazione dati in ember-data, possiamo senza dubbio scrivere i filtri manualmente:

esempio di query semplice

```
let usersTenant = await this.store.query('user-tenant', {
   filter: `equals(tenantId,'${tenantId}')`,
   include: `user.userProfile`,
   sort: `user.userProfile.lastName,user.userProfile.firstName`,
   page: {
       size: this.recordePerPage,
       number: this.pageNumber,
   },
});
```

In questo esempio abbiamo utilizzato il parametro filter per indicare una condizione di uguaglianza sull'attributo *tenatId*. Abbiamo specificato una Join mediante il parametro include, un ordinamento sui campi della Join e una paginazione, seguendo la convenzione json:api e i *diktat* dell'implementazione back-end consultabili qui.

In questo caso il filtro è piuttosto semplice, ma quando le condizioni diventano numerose, magari basate sulla selezione di diversi campi di filtraggio da interfaccia utente, la scrittura del parametro filter potrebbe diventare complicata e poco agevole.

Per semplificare il problema è stato implementato nel framework front-end il servizio custom json-api.js.
Iniziamo importandolo in un nostro componente o controller mediante: @service jsonApi;
Definiamo quindi un oggetto iniziale:

Oggetto iniziale di filtraggio.

```
let query = {
   page: {},
   filter: [],
   sort: [],
   fields: [],
   include: [],
}
```

Poiché ad oggi il servizio json-api gestisce ed ottimizza il solo parametro filter, aggiustiamo il nostro oggetto query riportando gli stessi include, sort e page di nostro interesse:

```
let query = {
   page: {
      size: this.recordePerPage,
      number: this.pageNumber,
   },
   sort: `user.userProfile.lastName,user.userProfile.firstName`,
   include: `user.userProfile`,
   filter: [],
}
```

Concentriamoci dunque sul parametro filter che, come visibile, per ora è un array vuoto.

IL PARAMETRO FILTER

Il parametro filter del nostro servizio json-api è un array di oggetti.
Ciascun oggetto rappresenta una condizione di filtraggio ed è così strutturato:

Struttura degli oggetti del parametro filter

```
{
   function: 'contains',
   column: 'tag',
   value: 'valoreX',
   value2: 'valoreY',
   negation: false,
}
```

Ogni condizione che vogliamo aggiungere alla nostra query verrà definita tramite un oggetto da aggiungere all'array filter. Tutti gli oggetti definiti nell'array filter saranno posti in AND.

Analizziamo ogni singola voce dell'oggetto:

- **function**: specifica il tipo di operazione di filtraggio nello standard JSON:API (es: *equals*, *contains*, *lessOrEqual* ecc). Stando all'esempio di partenza useremo *equals*.
- **column**: indica la colonna a cui applicare la funzione. Stando all'esempio di partenza useremo *tenantId*.
- **value**: è il valore da applicare alla funzione. Un valore *null* verrà convertito in un valore nullo del campo di ricerca. Nel nostro esempio useremo '${tenantId}'
- **value2**: è un campo facoltativo, default *null*. Lo useremo per le *function and*/*or* quando sono previsti due soli parametri. Nel nostro esempio non è necessario.
- **negation**: è un campo facoltativo, default *false*, usato per negare la condizione (not). Nel nostro esempio non è necessario.

Tornando quindi all'esempio, aggiungiamo all'array filter un unico oggetto:

```
query.filter.push({
   function: 'equals',
   column: 'tenantId',
   value: tenantId, // variabile pre-valorizzata
})
```

Se avessimo bisogno di ulteriori filtri, li aggiungeremo a *filter* con altre push.

GENERAZIONE DELLA QUERY FORMATTATA

Una volta terminata l'aggiunta dei filtri, dobbiamo chiedere al servizio json-api di restituirci una query formattata secondo lo standard JSON:API. A questo scopo è previsto il metodo ***queryBuilder***.

Generazione ed utilizzo della query

```
let real-query = this.jsonApi.queryBuilder(query);
```

```
let usersTenant = await this.store.query('user-tenant', real-query);
```

In questo esempio abbiamo generato la query opportunamente formattata e l'abbiamo utilizzata nel metodo store.query di ember-data.

ALTRE CASISTICHE ED ECCEZIONI

Poiché le funzioni di filtraggio di JSON:API sono numerose, analizziamo tutte le casistiche e le eccezioni degli oggetti filter contemplate dal nostro servizio custom json-api.

- funzioni **and** e **or**:
  - se sono previsti 2 soli valori in and/or, il parametro *column* è ignorato e in *value* e *value2* inseriamo le condizioni nello standard JSON:API (eventualmente contenenti le colonne)

    esempio di oggetto per funzioni *and*/*or*

    ```
    query.filter.push({
       function: 'and', // oppure or
       column: null,
       value: `equals(tenantId,'${tenantId}')`,
       value2: `not(equals(id,'3'))`,
    })
    ```
  - Se sono previsti più di due valori da mettere in and/or, vedere la funzione "extra".

- funzione **any**:
  - stando allo standard JSON:API, la funzione any ha questa struttura: *filter=any(chapter,'Intro','Summary','Conclusion')* che verrà tradotta in questo oggetto:

    esempio di oggetto per clausola any

    ```
    query.filter.push({
       function: 'any',
       column: 'chapter',
       value: 'Intro','Summary','Conclusion',
    });
    ```

    ovvero nel campo *value* inseriremo i valori separati da virgole e contenuti tra apici singoli.

- funzione **has**:
  - stando allo standard JSON:API, la funzione has ha questa struttura: *filter=has(articles)* che verrà tradotta in questo oggetto:

    esempio di oggetto per clausola has

    ```
    query.filter.push({
       function: 'has',
       value: `articles`,
    });
    ```

ovvero il parametro *column* viene ignorato ed in *value* inseriamo l'entità di nostro interesse.

- funzione ***extra***:
    - quando le casistiche sopra indicate non soddisfano le esigenze, il componente JSON:API ci consente un'ulteriore flessibilità. Ponendo infatti il campo *function* a *null*, possiamo inserire in *value* la nostra condizione, formattata secondo lo standard richiesto dal BE. Il campo *column* viene ignorato.

esempio di oggetto per caso "extra"

```
query.filter.push({
   function: null,
   value: `and(equals(tenantId,'1'),not(equals(id,3)))`,
});
```

All'atto pratico la stringa di filtraggio non viene modulata. Eventualmente, se oltre alla funzione "extra" sono presenti altri oggetti, la stringa viene posta in and con gli altri filtri.

NOTA: il campo negation è applicabile a ciascuna casistica e trasforma la condizione in `not(${condizione})`.

PASSAGGIO DI PARAMETRI TRA PAGINE

Durante gli sviluppi avremo sicuramente bisogno di passare dei parametri da una pagina all'altra. Ipotizziamo ad esempio di avere una pagina che mostra un elenco di record con un pulsante per ciascun record che rimanda a una pagina di dettaglio e di editing.

Abbiamo a disposizione diverse soluzioni per il passaggio dei parametri.

1. USARE I PARAMETRI DI QUERY

Apriamo il file route.js e modifichiamo la rotta della pagina a cui vogliamo passare i parametri:

route.js

```
this.route('manage-tenant'); // da così...
this.route('manage-tenant', { queryParams: ['tenant_id'] }); // ...a così
```

abbiamo cioè aggiunto il parametro *queryParams* alla rotta che punta alla pagina manage-tenant.

Nel gestore di rotta della pagina impostiamo:

routes/manage-tenant.js

```
export default class ManageTenantRoute extends Route {
 model(_, transition) {
   return {
     tid: transition.to.queryParams.tenant_id,
   };
 }
}
```

Nell'hook model andiamo a recuperare i dati che il template associato potrà utilizzare.
Il parametro *transition* è un oggetto che Ember passa al metodo model quando avviene la transizione verso la rotta. Contiene informazioni sulla transizione in corso, inclusi i parametri di query. Nello specifico *transition.to.queryParams* è un oggetto che contiene tutti i parametri di query presenti nell'URL al momento della transizione. In questo caso, si sta accedendo al parametro di query *tenant_id* da questo oggetto.

In definitiva stiamo ritornando un oggetto con la proprietà *tid* che nel template potremo usare a nostro piacimento:

template/manage-tenant.hbs

```
<h1>Gestione Tenant</h1>
<p>Id: {{this.model.tid}}</p>

<MyComponent @tid={{this.model.tid}} />
```

In questo codice vediamo come utilizzare il dato nel template e come passarlo ad un componente presente nel template.

Il passaggio di parametri tramite queryParams offre le seguenti caratteristiche:

- **Parametri Non Obbligatori**: i parametri di query non sono obbligatori, possono essere omessi senza influenzare la capacità di Ember di risolvere la rotta. Se non forniti, la rotta si caricherà normalmente, eventualmente con dei valori di default o senza filtraggi specifici.
- **Visibilità e persistenza**: sono visibili nell'URL come *?tenant_id=value*. Questo li rende utili per la conservazione dello stato della UI che può essere facilmente condiviso tramite URL. In altre parole, in caso di refresh della pagina i parametri restano disponibili.

I *queryParams* sono usati spesso per permettere all'utente di ripristinare uno stato specifico della pagina o per passare dati non critici che influenzano la visualizzazione o il comportamento della pagina, come impostazioni di filtraggio o ordinamento. Va notato che aggiornando la pagina, i parametri permangono.

Per effettuare il passaggio da una pagina all'altra, useremo via javascript il seguente codice:

redirect con queryParams in Javascript

```
this.transitionTo('some-route, { queryParams: { foo: 'bar' }});
```

e da HBS useremo il seguente codice:

redirect con queryParams in HBS

```
<LinkTo @route="some-route" @query={{hash foo="bar"}}>
 Vai alla pagina
</LinkTo>
```

APPROFONDIMENTO:
Va menzionata una possibilità che Ember ci mette a disposizione. Aggiungendo un **controller** alla pagina possiamo modificare la visibilità e il comportamento dei *queryParams*.

controllers/some-route.js

```
export default class SomeRouteController extends Controller {
 queryParams = [{
   foo: {
     refreshModel: true,
     replace: true
   }
 }];

 foo = null;
}
```

in questo codice abbiamo definito la proprietà array queryParams. Tale proprietà nel controller definisce come Ember deve gestire le variazioni dei parametri di query. È un array di oggetti che mappano ogni parametro di query a una configurazione specifica.

La proprietà *foo = null;* inizializza il parametro di query *foo*, stabilendo un valore di default quando la rotta viene caricata per la prima volta o il parametro non è specificato nell'URL, evitando errori di "undefined" quando si tenta di accedervi.

L'oggetto associato a *foo* nell'array *queryParams* ha due opzioni, refreshModel e replace.

Quando **refreshModel** è impostato a true, indica a Ember di ricaricare il modello della route ogni volta che il parametro di query cambia. È utile quando i dati mostrati nella route dipendono dal valore del parametro di query e devono essere aggiornati quando il parametro cambia. Ad esempio, se *foo* influisce sui dati recuperati in una richiesta API nel model hook della rotta, avere *refreshModel: true* garantirà che quei dati siano ricaricati ogni volta che *foo* cambia.

Quando **replace** è impostato a true, indica a Ember di modificare l'URL al cambiamento del parametro ma senza creare una nuova voce nella cronologia del browser. Questo significa che se l'utente naviga indietro nel browser, non ci sarà alcun passaggio intermedio per ogni cambio di *foo*. È utile per evitare la proliferazione di stati nella cronologia del browser che potrebbero confondere l'utente. È comunemente usato in situazioni come filtri di ricerca, dove l'utente potrebbe cambiare spesso i parametri di query ma non si desidera che ogni piccolo cambiamento sia registrato come un passo distinto nella cronologia.

L'uso di *refreshModel* e di *replace* fornisce flessibilità e può migliorare l'esperienza utente in applicazioni web dinamiche e interattive.

2. USARE I PARAMETRI DINAMICI

Apriamo il file route.js e modifichiamo la rotta della pagina a cui vogliamo passare dei parametri:

route.js

```
this.route('solution'); // da così...
this.route('solution', { path: 'solution/:appointment_id' }); // ...a così!
```

abbiamo cioè aggiunto il parametro *path* alla rotta che punta alla pagina solution.

Nel gestore di rotta della pagina impostiamo:

routes/solution.js

```
export default class SolutionRoute extends Route {
 model(params) {
   return { appointmentId: params.appointment_id };
 }
}
```

Nell'hook model andiamo a restituire un oggetto che nella chiave *appointmentId* contiene il parametro *appointment_id* presente nell'url, che nel template potremo usare a nostro piacimento:

template/solution.hbs

```
<h1>Solution</h1>
<p>Id: {{this.model.appointmentId}}</p>

<MyComponent @aid={{this.model.appointmentId}} />
```

In questo codice vediamo come utilizzare il dato nel template e come passarlo ad un componente presente nel template.

Il passaggio dei parametri dinamici ha le seguenti caratteristiche:

- **Parametri Obbligatori**: per risolvere correttamente la rotta, è necessario che il parametro sia presente (esempio: solution/123). Una chiamata alla rotta senza il parametro comporterà un 404.
- Indica una Relazione: solitamente, un parametro dinamico nel path indica una relazione diretta con l'entità specificata, ad esempio, caricare le informazioni specifiche di una *solution* con un dato ID.
- **Persistenza**: in caso di refresh della pagina, i parametri restano disponibili.

Viene generalmente utilizzato quando l'accesso a una specifica entità è essenziale per la funzionalità della pagina, come visualizzare o modificare i dettagli di uno specifico record.

Per effettuare il passaggio da una pagina all'altra, useremo via javascript il seguente codice:

redirect con parametri dinamici in Javascript

```
this.router.transitionTo('solution', param1 /*... , paramN */);
```

e da HBS useremo il seguente codice:

redirect con parametri dinamici in HBS

```
<LinkTo @route="solution" @model="123">
 Vai alla pagina
</LinkTo>

<!-- se abbiamo più parametri -->
<LinkTo @route="profile" @model={{hash param1=12 param2="abc"}}>
 Go to Profile
</LinkTo>
```

Se dobbiamo passare più parametri dinamici, è importante passarli nell'ordine corretto come definito nella rotta in route.js, altrimenti otterremo comportamenti anomali ed errori.

3. USARE IL SERVIZIO STATUS-SERVICE

Si tratta di un servizio custom che possiamo utilizzare per memorizzare dei dati da condividere in diversi punti dell'applicazione.

L'utilizzo è semplice e si articola in 3 step:
- definiamo una nuova variabile nel file del servizio (services/status-service.js);
- prima di effettuare il redirect, valorizziamo la suddetta variabile con il dato di nostro interesse;
- nella pagina di destinazione andiamo a leggere il valore della variabile nel servizio status-service (attraverso il controller della pagina o attraverso un componente inserito nel template).

Come tutti i servizi di Ember, status-service è un singleton e mantiene il suo stato finché l'applicazione è in esecuzione. Ciò significa anche che al refresh della pagina il dato non sarà conservato.

Il passaggio di parametri tramite status-service ha queste caratteristiche:
- **Parametri nascosti**, poichè non esposti nell'url. La condivisione dell'URL non consentirà di accedere alla risorsa (e dovremo implementare, nel controller o nel componente, la logica per gestire il caso di parametri non valorizzati).

- **Non persistenza** al refresh, che può essere un vantaggio in particolari situazioni.

È buona norma implementare una logica (ad esempio nel controller o nel componente stesso) che, quando si esce dalla pagina o quando l'istanza del componente viene distrutta, resetti le variabili del servizio status-service utilizzate.

COMPONENTI ANNIDATI

Quella dei componenti annidati è una casistica molto frequente nel framework che ci consente di condividere dati comuni, risparmiando il numero di chiamate al server e di far comunicare i componenti tra loro.

Ipotizziamo di avere una pagina con due tab, una contenente una lista di record selezionabili per la modifica ed una dedicata alla modifica stessa.
Optiamo per suddividere queste due funzionalità in due componenti, chiamati record-list e record-edit.

Quello che ci aspettiamo, quantomeno, è che:

- quando scelgo il record da modificare in *record-list*, *record-edit* carichi i dati del record da modificare;
- quando salvo le modifiche in *record-edit*, *record-list* aggiorni la lista per mostrare gli eventuali cambiamenti apportati.

Abbiamo cioè bisogno di far dialogare i due componenti.
A tale scopo definiamo un terzo componente, chiamato ***record-master*** con questo codice:

record-master.js

```
export default class ParentComponent extends Component {
  @tracked lastRefresh = 0;
  @tracked selectedRecord = ``;

   @action
   refresh() {
       this.lastRefresh = new Date().getTime();
   }

   @action
   selectRecord(id) {
       this.selectedRecord = id;
   }
}
```

record-master.hbs

```
<RecordList @selectRecord={{this.selectRecord}} @lastRefresh={{this.lastRefresh}}/>

<RecordEdit @refresh={{this.refresh}} @selectedRecord={{this.selectedRecord}}/>
```

Nel javascript del master abbiamo predisposto due azioni che modificano due variabili tracked.
Nell'HBS abbiamo inserito i due componenti *figli* passando al primo l'azione *selectRecord* e la variabile tracked *lastRefresh*, e al secondo l'azione *refresh* e la variabile tracked *selectedRecord*.

Analizziamo ora i due restanti componenti, concentrandoci solo sul codice necessario alla comunicazione tra i due.

record-list.js

```
export default class RecordListComponent extends Component {
```

```
    selectRecord = null;

    constructor(...attributes) {
        super(...attributes);
        this.selectRecord = this.args.selectRecord;
    }

    @action
    updateList() {
        // Logica per aggiornare la lista dei record
    }

    // chiamata quando selezioniamo il record da modificare
    @action
    setRecordToEdit(id) {
        this.selectRecord(id);
    }
}
```

record-list.hbs

```
<div class="record-list" {{did-update this.updateList @lastRefresh}} ...attributes>
    <!-- altro codice -->
</div>
```

Banalmente, abbiamo memorizzato in una variabile interna la funzione (del componente *padre*) passata come parametro. Abbiamo poi definito un'azione *setRecordToEdit* che sarà richiamata quando selezioniamo il record da modificare, passandole l'id.
(NOTA: per semplificare il codice non sono riportati i controlli di esistenza dell'argomento *arg.selectRecord* nel costruttore, né la verifica di non nullità della variabile *selectedRecord* dentro il metodo *setRecordToEdit;* tutto ciò è ammissibile a fini didattici ma **non è accettabile negli sviluppi reali!**).

Il risultato è che *setRecordToEdit* richiama la funzione *selectRecord* del master, andando così a valorizzare la variabile *selectedRecord* sempre del master. Essendo questa di tipo tracked, il secondo componente figlio, *record-edit*, si accorgerà della variazione di *selectedRecord* (avendola come parametro).

In modo del tutto analogo, quando in *record-edit* effettuiamo il salvataggio delle modifiche, richiameremo la funzione *refresh* del master. Tale funzione aggiorna la variabile *lastRefresh* del master con il timestamp attuale ed essendo tracked, tale variazione si propaga anche a *record-list* (poiché *lastRefresh* è nei suoi parametri).

Ora analizzando l'hbs di *record-list* notiamo la presenza di **did-update**, plugin che dovreste già conoscere perchè elencato tra i plugin aggiuntivi indispensabili del framework. Il did-update ascolta la variazione del parametro *lastRefresh* e quando si verifica, chiama la funzione *updateList* dello stesso componente, causando l'aggiornamento della lista dei record.

Abbiamo dunque soddisfatto il requisito di invocare l'aggiornamento dei record al salvataggio delle modifiche, costruendo di fatto una comunicazione tra i componenti.

Del tutto analoga sarà la struttura del codice nel componente record-edit.

COMUNICAZIONE TRA COMPONENTI E CONTROLLER

Il dialogo tra Componenti e Controller non è una pratica molto utilizzata nel framework, eppure c'è una casistica degna di nota.

Supponiamo di avere un template chiamato *test*, contenente l'usuale componente *app-page-title* e un altro componente che chiameremo *my-component*.

Ipotizziamo ora che il componente *my-component* abbia la necessità di modificare il titolo della pagina. Sembrerebbe impossibile perché i due componenti (*app-page-title* e *my-component*) non sono annidati e perchè, volendo mantenere lo standard per il template, non vogliamo includere il componente *app-page-title* all'interno del *my-component* (che sarebbe una soluzione funzionante, ma siamo abituati a mantenere gli standard!)

La soluzione è quella di creare un controller per il template e di farlo dialogare con il componente. Vediamo come.

Generiamo anzitutto il controller da associare al template, quindi con nome "test".

Controller test.js

```
export default class TestController extends Controller {
    @tracked customTitle = `titolo iniziale!`;

    @action
    setTitle(title) {
        this.customTitle = title;
    }
}
```

e nel template:

Template test.hbs

```
<ArchitectUi::AppMain::AppPageTitle
  @title={{this.customTitle}}
  @titleKey=""
  @description="Pagina con titolo variabile"
  @descriptionKey=""
  @icon="pe-7s-tools"
/>

<MyComponent @externalAction={{this.setTitle}}/>
```

Come vediamo, al parametro *title* del componente *app-page-title* è stata associata la variabile tracked del controller, che al momento è valorizzata a "*titolo iniziale!*".
Inoltre abbiamo aggiunto il nostro componente passandogli il metodo *setTitle* del controller mediante il parametro *externalAction*.

Ipotizziamo di avere questo codice nel javascript del componente:

Javascript del componente my-component

```
export default class MyComponent extends Component {
   changeTitle = null
   constructor(...attributes) {
       super(...attributes);
       this.changeTitle = this.args.externalAction;

       setTimeout(() => {
          this.changeTitle(`Nuovo titolo!`);
       }, 4000);
  }
}
```

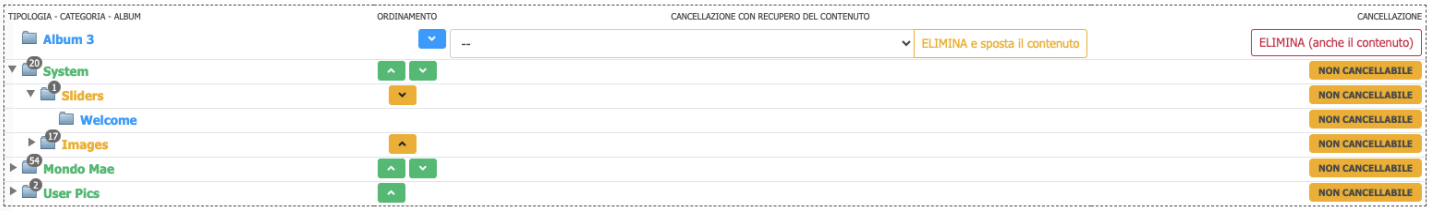
Come vediamo, nel costruttore assegniamo ad una variabile interna (*changeTitle*) il parametro *externalAction* che avevamo valorizzato con il metodo *setTitle* del controller. In altre parole ora la variabile *changeTitle* del componente è il metodo *setTile* del controller.
Dopo un'attesa di 4 secondi invochiamo *changeTitle* passandogli una stringa. In altre parole stiamo invocando il metodo *setTitle* del controller, il quale assegnerà alla variabile *customTitle* la stringa "*Nuovo titolo!*" e, essendo tracked, il componente *app-page-title* si accorgerà della variazione, aggiornando il titolo della pagina.

CARICAMENTO E GESTIONE DI FOTO E FILE

TIPOLOGIE, CATEGORIE, ALBUM

Il primo step per la gestione dei file è la definizione di Tipologia, Categoria ed Album. Queste entità sono gerarchiche: una tipologia può contenere più Categorie, ed una Categoria può contenere più Album.
Lo scopo di tali entità è quello di organizzare al meglio i nostri file. Non è possibile caricare un file sul server senza specificare la terna di entità.

Apriamo il menù *Media*/*Categorie* del Master Tenant.



Il framework fornisce già delle entità di default ma generalmente sarà necessario definire entità aggiuntive. Andremo sicuramente a definire una Tipologia ed una Categoria. Quanto all'album dovremo valutare se sia sufficiente creare un unico album o se è conveniente dividere le foto in più album. In quest'ultimo caso potremo spostare la logica di creazione degli album nel software gestisce l'upload dei file. La scelta dipende ovviamente dalle esigenze dell'applicativo che stiamo sviluppando.

UPLOAD DEI FILE

Il framework mette a disposizione tutti gli strumenti necessari all'upload dei file sia da web che in ambiente Cordova.
La sezione web si basa sul plugin ember-file-upload (la cui conoscenza è utile per meglio comprendere i passaggi successivi).
La sezione Cordova ad oggi implementata consente il caricamento di sole immagini e video (è già pianificato uno sviluppo per consentire l'upload di ogni altro file) e si basa sul servizio custom *cordovaUpload* (consultare i commenti del servizio per l'elenco dei plugin Cordova necessari).

Ipotizziamo di avere un componente custom "my-upload".
Nell'hbs del componente inseriamo:

my-upload.hbs

```
<!-- ALTRO CODICE CUSTOM -->
{{#if this.isCordova}}
  {{!-- CARICAMENTO PER AMBIENTE CORDOVA --}}
  <button class="btn btn-sm btn-success btn-hover-shine mr-1"
          type="button"
          {{on 'click' (perform this.getImageFromCamera 'camera')}}>
      <i class="fa {{if this.getImageFromCamera.isRunning 'fa-spinner fa-spin' 'fa-camera'}}"></i>
      SCATTA FOTO
  </button>
```

```
  <button class="btn btn-sm btn-success btn-hover-shine mr-1" type="button"
          {{on 'click' (perform this.getImageFromCamera 'album')}}>
      <i class="fa {{if this.getImageFromCamera.isRunning 'fa-spinner fa-spin' 'fa-upload'}}"></i>
      CARICA FOTO
  </button>
{{else}}
  {{!-- CARICAMENTO PER AMBIENTE WEB --}}
  {{#let (file-queue name="NOME-UNIVOCO" onFileAdded=(perform this.uploadPhoto)
            onUploadSucceeded=this.uploadSucceeded onUploadFailed=this.uploadFailed) as |queue|}}
       <div class="upload-area">
          <FileDropzone @queue={{queue}} @filter={{this.validateFile}} as |dropzone|>
              {{#if dropzone.active}}
              <p class="pt-4">Rilascia il file per avviare il caricamento</p>
              {{else if queue.files.length}}
              Caricamento in corso: ({{queue.progress}}%)
              {{else if dropzone.supported}}
              <input type="file" id="MY-ID"
                     {{queue.selectFile filter=this.validateFile }} class="d-none">
              <button class="btn btn-dark btn-hover-shine btn-sm" type="button"
                      {{on 'click' this.triggerFileInput}}>
                  Scegli il file
              </button>
              <br /><br />
              <span>o trascina il file qui!</span>
              {{/if}}
          </FileDropzone>
      </div>
  {{/let}}
{{/if}}
```

avendo l'accortezza di:

- modificare la stringa "NOME-UNIVOCO" con un nome univoco per tutto il sito (non devono cioè esistere due file-queue con lo stesso nome). Optare per un nome inglese che sia *parlante*.
- personalizzare l'ID "MY-ID" del campo input type="file"

Nel codice sopra riportato è incluso Dropzone per la sezione Web, quindi è attivo anche il drag-&-drop dei file. Per la sezione Cordova abbiamo la possibilità di fare scegliere all'utente se caricare un'immagine esistente nella gallery del telefono o se utilizzare la fotocamera per scattare una foto da caricare.

La classe *upload-area* applicata al *div* principale della sezione web ne standardizza la grafica.

Nel Javascript del componente, dopo aver incluso i **servizi**:

```
@service cordovaUpload;
@service fileQueue;
@service session;
@service dialogs;
@service store;
```

andremo a definire alcune funzioni accessorie:

my-upload.js

```
@tracked isCordova = false;

constructor(...attributes) {
  super(...attributes);
  this.isCordova = typeof window.cordova !== 'undefined';
}

@action
```

```
async uploadSucceeded(file) {
  //console.warn(file);
}

// errore di caricamento della foto
@action
async uploadFailed(file) {
  //console.warn(file);
}

// verifica il formato della foto da caricare (jpg, png)
@action
validateFile(file) {
  const allowedTypes = ['image/jpeg', 'image/png', 'image/jpg'];
  if (allowedTypes.includes(file.type)) {
    return true;
  } else {
    this.dialogs.toast(
      'Formato non consentito!<br />Sono ammesse solo immagini .jpg e .png',
      'error',
      'bottom-right',
      5,
      null
    );
    return false;
  }
}

@action
triggerFileInput() {
  let fileInput = document.getElementById('MY-ID');
  if (fileInput) {
    fileInput.click(); // Triggers the file input click event
  }
}
```

La funzione *validateFile* definisce le estensioni ammesse, potete personalizzarla aggiungendo o rimuovendo i mime-types.

Nella funzione *triggerFileInput* rinominiamo l'ID "MY-ID" con quello definito nell'HBS.

Aggiungiamo nel file javascript del componente anche il codice necessario all'upload web:

my-upload.js

```
uploadPhoto = task({ drop: true }, async (file) => {
  try {
    let self = this;
    let queue = this.fileQueue.find('NOME-UNIVOCO');

    let d = new Date();
    let endpoint = `${config.apiHost}/${config.namespaceHost}/fileUpload`;
    let currentLocation = window.location;

    let baseEndpoint = '';
    if (typeof window.cordova === 'undefined') {
```

```
      baseEndpoint = `${currentLocation.protocol}//${
        currentLocation.hostname
      }${currentLocation.port !== '' ? ':' + currentLocation.port : ''}`;
    } else {
      baseEndpoint = config.feHost;
    }

    file
      .upload(endpoint, {
        method: 'POST',
        headers: {
          authorization: `${this.session.get('data.access_token')}`,
          accept: 'application/vnd.api+json',
          platform: typeof window.cordova !== 'undefined' ? 'app' : 'web',
          'access-Control-Allow-Origin': '*',
          fingerprint: self.session.getFingerprint(),
          baseEndpoint: baseEndpoint,
          tenantId: self.session.get('data.tenantId') || 1,
        },
        data: {
          typologyArea: typologyId, // DEFINIRE L'ID DELLA TIPOLOGIA
          category: categoryId, // DEFINIRE L'ID DELLA CATEGORIA
          album: albumId, // DEFINIRE L'ID DELL'ALBUM
          tenantId: this.session.get('data.tenantId').toString(),
          userGuid: this.session.get('data.id').toString(),
          alt: '',
          tag: '',
          extension: '',
          base64: '',
          fileUrl: '',
          mongoGuid: '',
          originalFileName: '',
          uploadDate: `${d.getFullYear()}/${
            d.getMonth() + 1
          }/${d.getDate()} ${d.getHours()}:${d.getMinutes()}:${d.getSeconds()}`,
          type: 'mediaFiles',
          global: false,
        },
      })
      .then((response) => {
        if (response && response.ok) {
          return response.json();
        } else {
          throw new Error('Errore nella richiesta HTTP');
        }
      })
      .then(async (res) => {
        // caricamento foto avvenuto correttamente
        res = res.data.attributes;
        let imageId = res.StringId;

        // LOGICA CUSTOM QUI!
      })
      .catch((e) => {
        console.error(e);
        queue.remove(file);
        this.dialogs.toast(
          'Errore nel caricamento. Riprovare!',
          'error',
          'bottom-right',
          4,
          null
        );
```

```
      });
  } catch (e) {
    console.error(e);
  }
});
```

in cui dovremo:

- sostituire il nome della queue “*NOME-UNIVOCO*” con il nome inserito nell’HBS;
- implementare una logica custom per definire gli id della Tipologia, della Categoria e dell’Album;
- implementare la logica custom in caso di successo;

infine aggiungiamo, sempre nel file javascript del componente, il codice necessario per l’upload con Cordova:

my-upload.js

```
getImageFromCamera = task({ drop: true }, async (origin) => {
  await this.cordovaUpload
    .getPicture(origin, 'back', 95, 1500, false, 'base64')
    .then((data) => {
      let base64 = data[1];

      return this.cordovaUpload.uploadPicture(
        base64,
        typologyId, // DEFINIRE L'ID DELLA TIPOLOGIA
        categoryId, // DEFINIRE L'ID DELLA CATEGORIA
        albumId, // DEFINIRE L'ID DELL'ALBUM
        false,
        this.session
      );
    })
    .then(async (res) => {
      // caricamento foto avvenuto correttamente
      res = res.data.attributes;
      let imageId = res.StringId;
      console.warn(imageId);
      // LOGICA CUSTOM QUI!
    })
    .catch((e) => {
      console.error(e);
      this.dialogs.toast(
        'Errore nel caricamento. Riprovare!',
        'error',
        'bottom-right',
        4,
        null
      );
    });
});
```

ovviamente dovremo:

- implementare una logica custom per definire gli id della Tipologia, della Categoria e dell’Album;
- implementare la logica custom in caso di successo;
- modificare secondo necessità i parametri passati al metodo getPicture del servizio *cordovaUpload*.

Sia per l'upload web che per l'upload Cordova, inoltre, va specificato se la foto è liberamente fruibile o meno, attraverso il flag booleano "*global*" (nel codice dell'upload web il parametro global è visibile come attributo mentre nell'upload Cordova è il 5° parametro del metodo *uploadPicture* del servizio *cordovaUpload*.

I FILE CARICATI

Mediante l'end-point di caricamento appena utilizzato otteniamo quanto segue:

- per ciascun file viene creato un record mysql;
- per ciascun file vengono creati 3 record su mongo-db contenenti, rispettivamente, il base64 del file in versione originale, il base64 del file ridotto a grandezza intermedia e il base64 della miniatura del file;
- i file sono fisicamente caricati sul server in tre diverse dimensioni (originale, intermedia, piccola) a questo url:
  https://*tuosito.it*/assets/virtual/uploads/TYPOLOGY-ID/CATEGORY-ID/ALBUM-ID/DIMENSIONE/*GUID-MY-SQL*.png

Nell'url si notano:

- gli ID di Tipologia, Categoria ed Album
- la dimensione del file, i cui valori possono essere: **small, medium, big**
- il nome del file che corrisponde al guid del record my-sql. L'estensione è la medesima del file originale.

Per ottimizzare il consumo di banda è stato predisposto un apposito end-point **custom** che, specificando la dimensione desiderata, ci consente di prelevare dal DB i record associati a ciascun file, con il relativo base64.
Si tratta dell'end-point ***mediaFiles*** richiamabile con il servizio fetch, filtrabile con i parametri del model media-file.
Analizziamo il modello di media-file:

models/media-files.js

```
export default class MediaFileModel extends Model {
 @belongsTo('media-category', { async: true, inverse: 'typologyMediaFiles' })
 typologyAreaRel;
 @belongsTo('media-category', { async: true, inverse: 'categoryMediaFiles' })
 categoryRel;
 @belongsTo('media-category', { async: true, inverse: 'albumMediaFiles' })
 albumRel;

 @attr('string', { defaultValue: '' }) album;
 @attr('string', { defaultValue: '' }) alt;
 @attr('string', { defaultValue: '' }) base64;
 @attr('string', { defaultValue: '' }) category;
 @attr('string', { defaultValue: '' }) extension;
 @attr('string', { defaultValue: '' }) primaryContentType;
 @attr('string', { defaultValue: '' }) fileUrl;
 @attr('string', { defaultValue: '' }) originalFileName;
 @attr('string', { defaultValue: '' }) tag;
 @attr('number', { defaultValue: '' }) tenantId;
 @attr('string', { defaultValue: '' }) typologyArea;
 @attr('date, {
```

```
    defaultValue() {
      return new Date();
    },
  })
  uploadDate;
  @attr('string', { defaultValue: '' }) userGuid;
  @attr('boolean', { defaultValue: false }) global;
}
```

Aldilà delle relazioni, notiamo che negli attributi non è presente alcun parametro che identifica la dimensione e che sono presenti gli attributi base64 e fileUrl.
Questo perché nel database principale (my-sql o ms-sql) il record unico che identifica il file ha il solo scopo di memorizzare le informazioni di base (nome originale, estensione, la terna di entità d'appartenenza, tag, alt, tenantId, userGuid, uploadDate e global.
Mentre l'url e il base64 sono memorizzate in mongo-db e dipendono dalla grandezza desiderata.

L'end-point mediaFiles prevede un campo di ricerca aggiuntivo (del resto è un end-point custom che non richiamiamo con ember-data!) che è **size**; questo parametro determina il record di mongo-db dal quale verranno estratti il base64 e l'url, ed i possibili valori del campo size sono: **sm, md, lg** (stessa notazione di bootstrap). Se non specificato, viene estratto il base64 della miniatura.
Per un maggiore approfondimento sulle operazioni di CRUD relative a media-file potete analizzare il codice del componente media-crud del framework.

VISUALIZZAZIONE DI UN'IMMAGINE

Per la visualizzazione di un'immagine non utilizzeremo il tag <img> ma il componente dedicato **show-image** che presenta diversi vantaggi:

- visualizza un'immagine di loading fino ad avvenuto caricamento dell'immagine desiderata;
- in caso di errore nel caricamento mostra un'apposita immagine di 404
- consente di visualizzare immagini via url ma anche tramite guid, provvedendo in modo autonomo all'estrazione del base64 dal DB.
- È asincrono rispetto al caricamento della pagina

L'utilizzo è semplice:

```
// MEDIANTE URL
<Standard::ShowImage @image="http://...." @size="" @byUrl="true"/>

// MEDIANTE GUID
<Standard::ShowImage @image="a123-b4f9-...." @size="md" @byUrl=""/>
```

Nel primo caso il componente viene utilizzato per visualizzare un'immagine via url.
Quindi nel parametro @image inseriamo l'url e imposteremo il parametro @byUrl a "true". Il parametro @size viene ignorato.
Generalmente lo utilizzeremo così quando vogliamo mostrare un'immagine locale.

Nel secondo caso il componente viene utilizzato per mostrare un'immagine caricata sul server, di cui conosciamo il guid.

Andremo quindi ad inserire il guid nel parametro @image e lasceremo vuoto il parametro @byUrl per specificare che la lettura avviene tramite guid. In tal caso è necessario indicare la dimensione desiderata mediante il parametro @size, altrimenti viene estratta la versione small.
In questo caso il componente provvederà autonomamente a scaricare il base64 dal DB e a mostrare l'immagine.

Il componente show-image si adatta automaticamente alla dimensione dell'elemento genitore. Ciò significa che l'elemento genitore (ad esempio un) dovrà avere una larghezza e un'altezza definite (e maggiori di zero), altrimenti la foto non sarà visibile.

NOTA: è evidente che per visualizzare un'immagine di cui conosciamo il base64, senza effettuare il download dal server, useremo il componente impostando @byUrl a "true" ed inserendo il base64, comprensivo di mimeType, direttamente nel parametro @image.

INTERNAZIONALIZZAZIONE

INTRODUZIONE

Nel framework è stato predisposto un sistema **custom** per la gestione delle traduzioni multilingua.

Sebbene siano disponibili plugin Ember dedicati per la gestione completa dell'internazionalizzazione (traduzioni, valute e date), molto completi ma anche complessi, il sistema custom implementato non gestisce né le valute né le date, fattore di secondaria importanza dati gli strumenti di localizzazione disponibili nelle ultime release del Javascript (basti pensare al metodo toLocaleString disponibile sia per le date che per i numeri, potenziato con la specifica ECMA-402 del 2012).

L'utilizzo delle funzioni custom di internazionalizzazione è opzionale ma predisporre il multilingua fin dall'inizio fornisce due notevoli vantaggi:

- predispone il sito ad eventuali future traduzioni, senza dover apportare modifiche al codice;
- consente di modificare i testi direttamente da interfaccia, anche on-the-fly, senza la necessità di ripubblicazione. Questo è particolarmente significativo in caso di App, dato l'oneroso iter di rilascio degli aggiornamenti.

COME FUNZIONA IL SERVIZIO

Quando il servizio multilingua è attivo, il sito web analizza la lingua del browser e verifica se è disponibile la relativa traduzione. In caso affermativo verrà caricata dal DB la traduzione nella lingua dell'utente, altrimenti viene mostrata la lingua di default.
Nell'header e nel footer del sito viene inoltre mostrato un menù che consente all'utente di selezionare la traduzione preferita (tra quelle disponibili).

Affinché i testi del sito possano essere tradotti, devono essere inseriti attraverso un apposito componente chiamato TR, da utilizzare così:

```
<Standard::TR @key="component.gdpr.accept" @default="Accetto tutto"/>
```

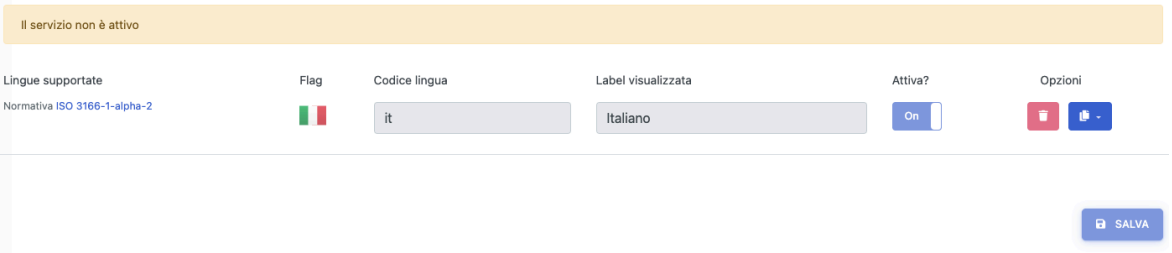
Senza scendere nel dettaglio dei parametri, possiamo già notare la presenza di una chiave (che richiamerà il valore tradotto) e di un valore di default che sarà mostrato se il servizio di traduzione viene disattivato.

Quando il servizio multilingua è attivo inoltre, gli utenti autorizzati (cioè quelli aventi il ruolo *Translator*) avranno accesso a un menù laterale destro dedicato alle traduzioni in cui potranno attivare la traduzione on-the-fly. Questa funzionalità fa apparire un'icona di editing vicino ad ogni testo, premendo la quale potranno modificare real-time la traduzione nella lingua corrente.
Quindi se abbiamo l'accortezza di inserire tutti i testi del sito mediante il componente TR (anche se al momento non è previsto il multilingua), non solo predisponiamo il sito a future traduzioni ma consentiamo sin da subito le modifiche on-the-fly.

Altra caratteristica interessante del servizio multilingua è che, per ciascun testo, possiamo specificare una traduzione diversa per App e per Web, permettendo così di gestire al meglio l'occupazione sugli schermi più piccoli.

ATTIVAZIONE DEL SERVIZIO

Per attivare il servizio multilingua aprire il menù Impostazioni/Setup/”Multi-Lingua”.

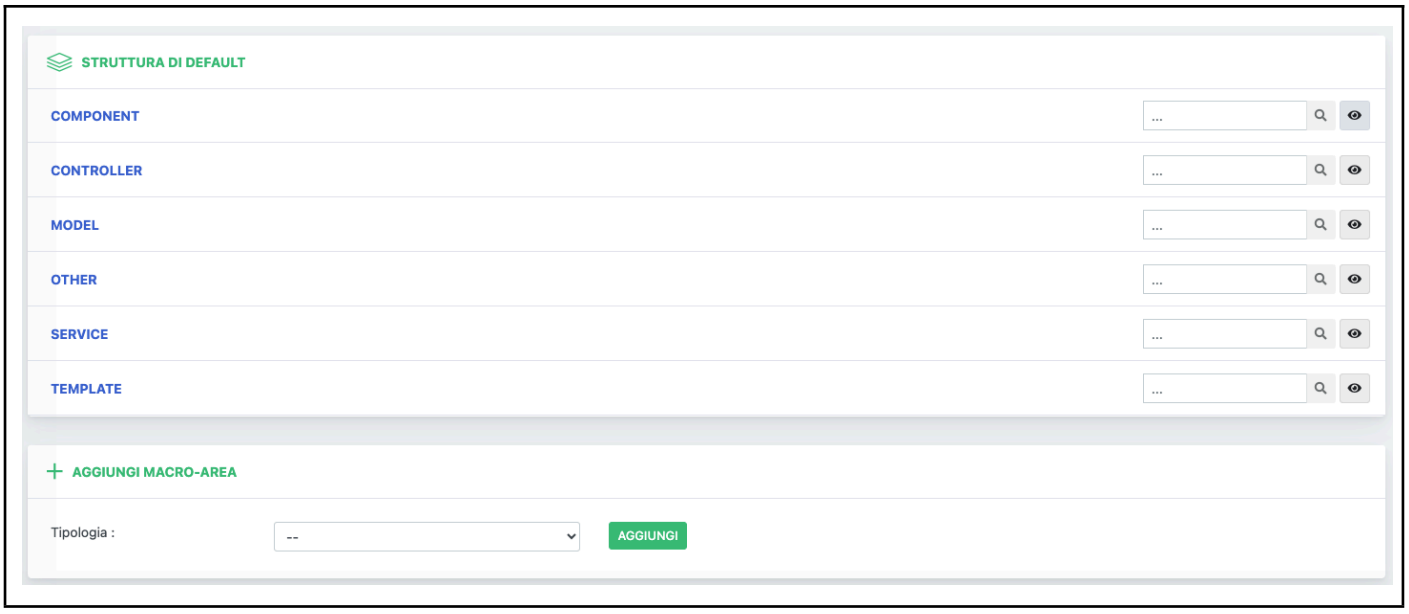


Notiamo che di default è sempre attiva la lingua italiana.
Premiamo il secondo pulsante sotto la voce “Opzioni” e, nel popup che compare, selezioniamo una lingua da aggiungere. Inizialmente la nuova lingua aggiunta non é attiva, provvederemo ad attivarla una volta completata tutta la procedura di traduzione, di cui parleremo a breve.

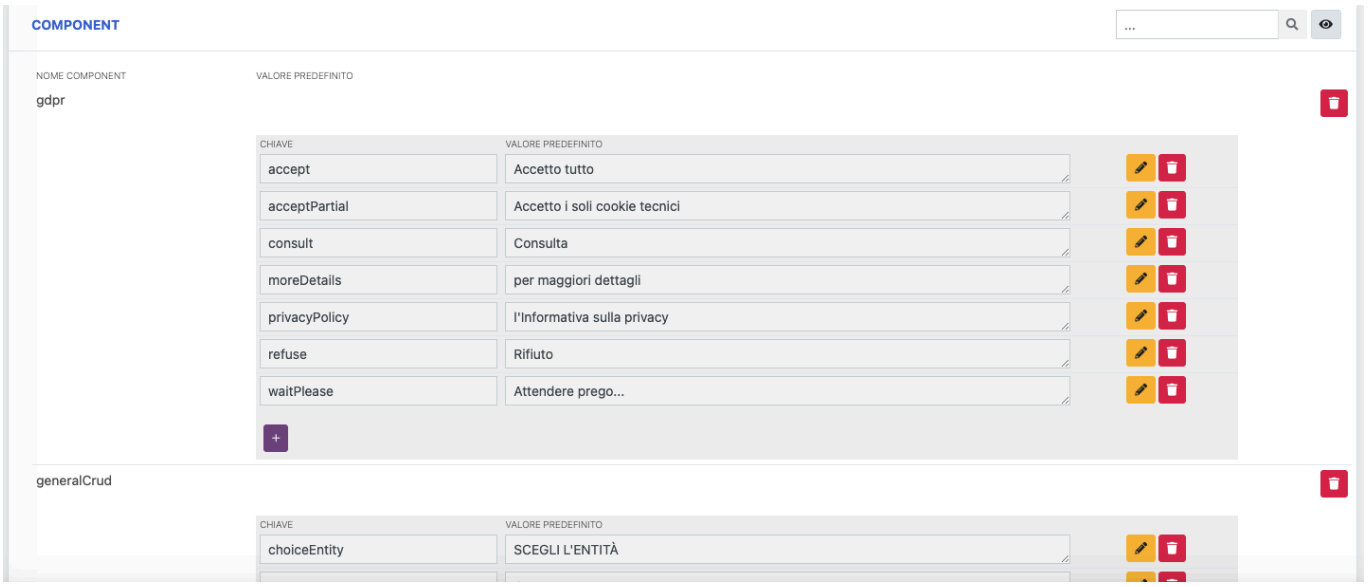
NOTA: occorrono almeno due lingue attive affinché il servizio multilingua si attivi.

STRUTTURA DELLE TRADUZIONI

Poiché i testi da gestire all’interno di un sito possono essere numerosi, il framework ci fornisce uno strumento organizzativo molto potente.
Apriamo dunque il menù Impostazioni/”Struttura traduzioni”.



e notiamo delle macro-aree espandibili piuttosto comuni in Ember, come componenti, controller, template ecc. La prima suddivisione dei testi è appunto per macro-area (all’occorrenza possono esserne aggiunte ulteriori). Espendendone una, ad esempio component, notiamo un’ulteriore suddivisione:



ovvero ogni singolo componente ha i propri testi (altrettanto dicasi per i template, controller ecc).
Ogni singolo testo è costituito da una coppia chiave-valore:

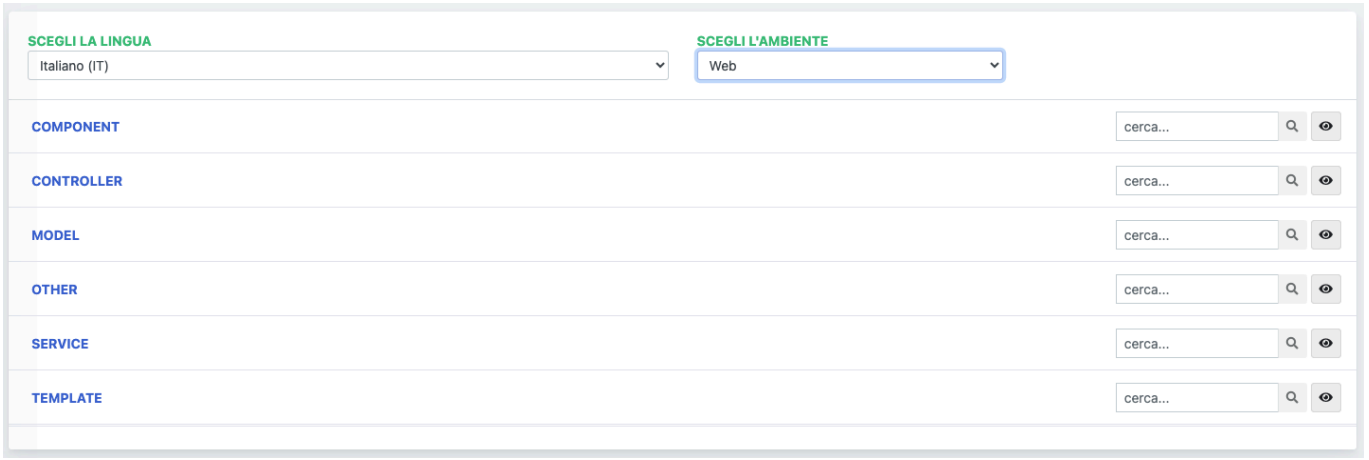
- Le chiavi vanno scritte in inglese e nel formato camel-case.
- I testi, per nostra comodità, sono scritti in italiano.

Ciò che è importante notare è che la “Struttura traduzioni” non è una traduzione vera e propria ma è una definizione di tutti i testi del sito.

Se creiamo una nuova rotta, aggiungiamo sotto la macro-voce template il nome del template associato alla rotta e lo popoliamo con tante coppie chiave-valore quanti sono i testi che vogliamo inserire. Nell’HBS del template inseriremo i testi con il componente TR già visto in precedenza e di cui parleremo più in dettaglio in un prossimo capitolo.

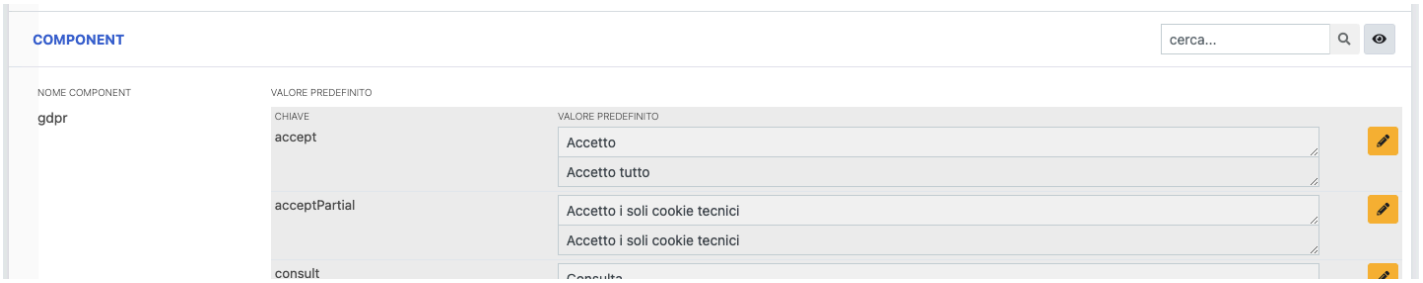
LE TRADUZIONI

Come detto, la "Struttura traduzioni” è solo un elenco strutturato che definisce le chiavi e i relativi testi. Il menù Impostazioni/Traduzioni ci consente di definire le traduzioni vere e proprie.



Dopo aver scelto la lingua di nostro interesse (tra quelle attivate in precedenza nel menù Impostazioni/Setup/"Multi-Lingua") dobbiamo scegliere l'ambiente di interesse, tra Web ed App. Come anticipato, infatti, possiamo distinguere i testi dedicati ai due ambienti. Se non ci interessa tale distinzione scegliamo l'ambiente Web (in questo ambiente possiamo scegliere di salvare le traduzioni per il solo ambiente Web o per entrambi).

Si presenta dunque l'elenco delle macro-aree definito nella "Struttura traduzioni".
Scegliamo ad esempio la macro-area Component:



Come si può vedere, per ciascuna chiave di ogni componente compaiono due campi:

- il valore inserito nella "Struttura traduzioni", in sola lettura;
- una text-area in cui inserire la traduzione del suddetto testo.

Per ciascuna lingua andremo quindi ad inserire le traduzioni di tutti i testi.
È evidente che possiamo anche ridefinire il testo per la lingua italiana, oltre che per le altre lingue.

Alcune considerazioni importanti:

- ogni volta che generiamo un elemento di ember tramite la ember-cli, sarebbe opportuno accedere alla "Struttura traduzioni" ed aggiungere il relativo elemento nella macro-area corrispondente.
- Aggiungendo una coppia chiave-valore alla "Struttura traduzioni", questa verrà aggiunta a tutte le lingue attive; il testo sarà quello da noi inserito in italiano, anche nelle altre lingue (dovremo poi occuparci delle traduzioni).
- Le altre lingue possono essere aggiunte in qualunque momento.

IL COMPONENTE TR

```
<Standard::TR @key="component.gdpr.accept" @default="Accetto tutto"/>
```

Come abbiamo già visto, il componente TR serve ad inserire i testi negli HBS. Prevede due soli parametri:

- @default : è il testo di default che viene inserito quando il servizio multi-lingua viene disattivato o quando la traduzione non viene trovata (ad esempio perchè abbiamo scritto male la chiave);
- @key : è la chiave che identifica univocamente il testo da inserire. È composto da tre parti separate da punti:
  - la macro-area della "Struttura traduzioni", ad esempio component;
  - il nome del componente (o del controller, template ecc);
  - la chiave del testo

Il componente preleva e mostra la traduzione corrispondente. Se non la trova (ad esempio perché abbiamo scritto la @key in modo errato o perché abbiamo disattivato il servizio multi-lingua) viene mostrato il testo di default.
Inoltre, come già detto, il componente TR rende possibile la traduzione on-the-fly.

Per quanto riguarda i testi da inserire nei file Javascript (ad esempio i messaggi di notifica di dialogs.toast o i messaggi di conferma dei dialogs.modal) useremo un servizio dedicato, oggetto del prossimo capitolo.

IL SERVIZIO TRANSLATION

Tutte le funzionalità del multilingua si basano sul servizio translation. All'avvio dell'applicazione tale servizio effettua una serie di operazioni:

1. verifica se il servizio multilingua è attivo (almeno due lingue attivate nel menù Impostazioni/Setup/"Multi-lingua"). In caso affermativo comanda la visualizzazione del selettore lingua nell'header e nel footer e predispone la visualizzazione del menù laterale destro per gli utenti autorizzati.
2. verifica se l'utente ha già selezionato una lingua preferita. In caso negativo verifica se è disponibile una traduzione nella lingua del browser dell'utente (altrimenti imposta la lingua di default).
3. preleva dal DB la traduzione nella lingua del punto precedente e la rende disponibile per tutti i componenti TR del sito.

Il servizio translation rende disponibili 4 variabili tracked che conviene analizzare:

- languageAvailable: di tipo booleano, indica se è il servizio di traduzione attivo.
- currentLang: di tipo stringa, contiene il codice della lingua corrente (quella impostata al punto 2)
- translationOnTheFly: di tipo booleano. Quando vale true, i componenti TR mostrano l'icona di editing vicino al testo. Difficilmente avremo bisogno di utilizzare questa variabile!
- **languageTranslation**: di tipo array, contiene la traduzione vera e propria nella lingua corrente ed è la variabile che utilizzeremo più spesso.

Vediamo come **importare la traduzione dei testi nei file Javascript**, ipotizzando di avere un componente chiamato "my-component" che mostra una notifica toast nella lingua corrente.

my-component.js

```
@service translation;

translatedMessage = '';

constructor(...attributes) {
   super(...attributes);
   try {
            this.translatedMessage =
this.translation.languageTranslation.component.myComponent.alertMessage;

       this.dialogs.toast(
           this.translatedMessage, // il messaggio del toast tradotto
           'warning',
           'bottom-right',
           3
```

```
        );
    } catch (e) {
        console.error(e);
    }
}
```

Anzitutto abbiamo importato il servizio *translation* poi, ipotizzando che la chiave del testo di nostro interesse (definita in “Struttura traduzioni”) sia *alertMessage*, abbiamo estratto la traduzione del messaggio.

Per estrarre la traduzione, quindi, è sufficiente aggiungere a translation.languageTranslation la terna, separata da punti, costituita da:

- il nome della macro-area (in questo caso *component*)
- il nome dell’elemento in formato camel-case (in questo caso *myComponent*)
- la chiave del testo (in questo caso *alertMessage*)

ATTENZIONE:
È importante utilizzare sempre il try-catch quando si estrae la traduzione, per evitare che eventuali errori nella scrittura della terna possano bloccare il caricamento dell’intero sito!

BROADCAST-CHANNEL

Se apriamo diverse istanze (finestre) di un sito realizzato con il framework, queste saranno tutte sincronizzate. Ad esempio se effettuiamo il logout, questo si propagherà in tutte le istanze, perché previsto in modo nativo da Ember.

Laddove avessimo bisogno di sincronizzare eventi custom, è stato predisposto un servizio apposito chiamato broadcast-channel.js. Possiamo vederlo come una sorta di web-socket locale che consente la propagazione di un evento tra istanze diverse.
Lo scopo è quello di eseguire un'azione su tutte le istanze quando in una di esse si verifica un evento.

L'utilizzo è molto semplice. Per consentire ad un componente (o a un controller) di invocare un'azione collettiva, importiamo anzitutto il servizio mediante *@service broadcastChannel;*

Quindi usiamo il metodo postMessage per richiamare l'azione comune:

invocare un'azione del broadcast-channel

```
this.broadcastChannel.postMessage('refresh-tenant');
```

(tra le parentesi va inserito il nome dell'azione da eseguire).

Nel file del servizio definiamo infine l'azione da eseguire:

istance-initializers/broadcast-channel.js

```
switch (event.data) {
   case 'refresh-tenant':
       // richiesto reload per cambio tenant
       window.location.replace('/');
       break;
}
```

In questo esempio (realmente esistente nel framework) abbiamo definito un *case* per l'azione 'refresh-tenant'. In pratica se cambiamo tenant in una scheda, tutte le altre faranno un redirect alla home (grazie a questo Ember aggiornerà il tenant in tutte le schede).
Se avremo bisogno di ulteriori azioni di sincronizzazione, basterà aggiungere i relativi *case*.

I SERVIZI CUSTOM

AUDIO

Il servizio audio.js è stato creato per la riproduzione di file audio, sia per il Web che per App. E' utilizzato ad esempio per la riproduzione del suono all'arrivo di una notifica push quando l'app è in foreground.
Per funzionare in ambiente Cordova è necessario installare i plugin *cordova-plugin-media* e *cordova-plugin-device*.

I suoni da riprodurre vanno posti nella apposita cartella **public/assets/sounds/** ed avranno una doppia estensione:

- .m4a per iOS e Web
- .aac per Android

È quindi sottinteso che se i nostri file audio hanno estensioni diverse, dovremo occuparci della loro conversione.
Il servizio prevede un unico metodo pubblico:

metodo del servizio audio per la riproduzione di suoni

```
play(fileName, device)
```

i cui parametri sono:

- filename: nome del file senza estensione. Quest'ultima sarà impostata automaticamente (tra .m4a e .aac) in base alla piattaforma (Web, iOS, Android)
- device: questo parametro sarà *null* in ambiente Web o la variabile *device* del plugin *cordova-plugin-device* in ambiente Cordova. Possiamo cioè valorizzarlo così:

```
@service audio;
// ...

cordovaDevice = typeof device !== 'undefined' ? device : null; // device è varia
cordova
this.audio.play(fileName, cordovaDevice);
```

CORDOVA-UPLOAD

Abbiamo già visto questo plugin nella sezione dedicata al caricamento dei file.

DIALOGS

Abbiamo già visto questo file nella sezione *Segnalatori*.

DOWNLOAD

E’ un servizio per il *download* dei file, sia in ambiente Web che su App.
All’atto pratico converte un base64 in un file da scaricare, risparmiandoci la procedura manuale che, specie in ambiente Cordova, è particolarmente articolata.
Se abbiamo un file sul server, potremmo prelevarne il base64 e passarlo al servizio per effettuarne, di fatto, il download.
Per funzionare in ambiente Cordova, richiede i seguenti plugin:

- *cordova-plugin-camera*
- *cordova-plugin-file*
- *cordova-plugin-file-transfer*
- *cordova-plugin-file-opener2*
- *cordova-plugin-device*
- *cordova-plugin-inappbrowser*

Sebbene preveda diversi metodi di utilità, l’unico metodo da richiamare per effettuare il download è il seguente:

metodo del servizio download per scaricare file

```
download(base64, filename)
```

i cui parametri sono:

- base64: è il base64 del file, privo di mime-type
- filename: è il nome da attribuire al file da scaricare, comprensivo di estensione

Tipicamente effettueremo una chiamata fetch al server per farci restituire il base64 che poi passeremo al metodo download.

FETCH

Le chiamate alle API sono tipicamente gestite in automatico da *ember-data*. Tuttavia può presentarsi la necessità di chiamare API che non rispondono con lo standard JSON:API. Sebbene questa eventualità sia da evitare, perché non consente di sfruttare le notevoli potenzialità di ember-data, all’occorrenza useremo il metodo fetch che, rispetto all’istruzione fetch nativa del Javascript, provvede automaticamente all’inserimento di tutti gli headers richiesti dal framework back-end (bearer, fingerprint ecc).
Gestisce inoltre in modo automatico le procedure di refresh-token in caso di 401 e in generale è ottimizzato per lavorare con il framework front-end.

Prevede un unico metodo pubblico da chiamare, che restituisce una Promise:

Metodo unico del servizio fetch per effettuare chiamate non JSON:API.

```
async call(endpoint, method, data, headers, authorization, session)
```

Analizziamo i parametri:

- **endpoint**: stringa contenente il nome dell'endpoint da chiamare. Il servizio provvederà autonomamente ad aggiungere tutti gli altri parametri (host, 'api', versione delle API ecc).
  Ad esempio se usiamo 'categories', la chiamata verrà effettuata verso https://*maefwk6-dev.maestrale.it*/api/v6/categories.
  Notare che ***gli inflector non hanno alcun effetto*** sull'URL generato.
- **method**: stringa contenente il metodo con cui effettuare la chiamata (rispetto a JSON:API, è stato previsto anche il metodo PUT che, nelle API non standard, viene spesso erroneamente usato per le chiamate di aggiornamento)
- **data**: oggetto Javascript contenente gli eventuali dati da passare al server. In caso di chiamate GET i dati dell'oggetto verranno automaticamente *convertiti* in query-string.
- **headers**: oggetto nullable per passare eventuali headers aggiuntivi (oltre a quelli già inseriti di default)
- **authorization**: booleano per indicare al servizio se aggiungere o meno gli headers di autenticazione
- **session**: tramite questo parametro passeremo un'istanza del servizio di sessione (useremo semplicemente this.session).

Un esempio di utilizzo può essere il seguente:

```
try {
   let res = await this.fetch.call('endpoint2', 'POST', {foo: bar}, {'Content-Type':
'application/json'}, true, this.session);
} catch (e) {
   console.error(e);
}
```

notare l'*await* con cui attendiamo la risoluzione della Promise e il *try-catch* con cui gestiamo eventuali errori di chiamata.

HEADERS

Questo servizio, insieme ad altri file annessi, svolge numerose funzionalità e merita un dettagliato approfondimento. Vediamo dapprima le sue proprietà tracked:

- **internalNotifications**: booleano che indica se il servizio di comunicazioni interne è attivo, cioè se è stato attivato dal menù Impostazioni/Setup/Comunicazioni
- **notifications**: intero che indica il numero di notifiche non lette
- **internalChat**: intero che indica lo stato del servizio di chat (0 = non attivo, 1 = attivo solo tra utenti e admin, 2 = attivo tra tutti gli utenti)
- **messages**: intero che indica il numero di messaggi non letti
- **search**: booleano che indica se mostrare o meno il campo di ricerca previsto dal tema, in base alla configurazione di Setup.
- **incomplete**: array che contiene l'elenco delle configurazioni mancanti (approfondiremo a breve questa proprietà)
- **advicesList**: oggetto Javascript che contiene la lista dei messaggi non letti e delle notifiche non lette (approfondiremo a breve anche questa proprietà)
- **updatingAdvices**: booleano da porre a true nella fase di aggiornamento di *advicesList*, per far comparire un *loader* nel componente di notifiche/messaggi dell'header.

Variabile *incomplete*

Il framework prevede un sistema di auto-monitoraggio che mostra agli utenti autorizzati (ovvero a quelli aventi il claims ***canSeeIncompleteConfigurations***) eventuali problemi di configurazioni mancanti o incomplete. Il suddetto array viene popolato da un'apposita sezione custom del framework, non prevista in un'applicazione Ember standard.

Per meglio comprenderne il funzionamento, apriamo il file **utils-incomplete-config.js** all'interno della cartella **utility/**. Questo file si occupa di interrogare il server alla ricerca delle informazioni incomplete. Se decidiamo di monitorare ulteriori configurazioni rispetto a quelle già presenti, dovremo chiedere agli sviluppatori back-end di implementare la logica lato server. Sempre nel nostro file poi, notiamo il metodo privato *_humanSyntax* che si occupa di convertire il dato proveniente dal BE in una notazione umana; chiaramente dovremo aggiungere anche qui dei *case* specifici.

Ciò che è importante notare è che la variabile *incomplete* non è autoaggiornante. Viene valorizzata all'avvio dell'applicazione ma abbiamo l'onere di ricaricarla laddove nel nostro codice apportiamo delle modifiche che ne possano modificare lo stato (ad esempio quando consentiamo all'utente di completare una configurazione incompleta). Per fare ciò dovremo importare il suddetto file utils-incompete-config in questo modo:

import dell'utility di aggiornamento delle configurazioni incomplete

```
import { getIncomplete } from 'poc-nuovo-fwk/utility/utils-incomplete-config';
```

e poi aggiornare la variabile così:

Aggiornamento della variabile *incomplete* del servizio header

```
@service session;
@service header;
@service fetch;

// ...

this.header.incomplete = await getIncomplete(this.fetch, this.session);
```

Variabile advicesList

In modo del tutto analogo ad *incomplete*, dentro la cartella utility/ è stato definito il file **utils-get-advices.js** che si occupa di estrapolare dal DB l'elenco delle notifiche e dei messaggi non letti.

Per aggiornare la variabile useremo questo import:

import dell'utility di aggiornamento di notifiche e messaggi non letti

```
import { getAdvices } from 'poc-nuovo-fwk/utility/utils-get-advices';
```

e richiameremo l'aggiornamento in questo modo:

Aggiornamento della variabili *advicesList* (e correlate) del servizio header

```
@service session;
@service header;
@service store;

// ...

this.header.updatingAdvices = true;
this.header.advicesList = await getAdvices(
     this.store,
     this.session,
     this.header
);
this.header.notifications = this.header.advicesList.notifications.length;
this.header.messages = this.header.advicesList.messages.length;
this.header.updatingAdvices = false;
```

Qui notiamo delle leggere differenze rispetto al caso precedente. Anzitutto valorizziamo a true la proprietà *updatingAdvices* per informare l'utente che le notifiche e i messaggi non letti sono in corso di aggiornamento. Quindi attendiamo l'aggiornamento dell'oggetto *advicesList* chiamando l'apposito metodo; aggiorniamo il numero di notifiche non lette e di messaggi non letti, valorizzando le proprietà *notifications* e *messages* ed infine ripristiniamo a false *updatingAdvices*.

INTEGRATION

Questo servizio si occupa dell'integrazione con servizi di autenticazione esterni.
Questa parte di guida verrà completata in un secondo momento!

JS-UTILITY

È un servizio contenente funzioni Javascript che torneranno molto utili. Va importato con @service jsUtility. I metodi principali sono:

- **regex**: metodo che restituisce la regex corrispondente a una specifica tipologia passatagli come argomento in formato stringa. Prevede diverse tipologie comuni di espressioni regolari (email, codice fiscale, guid ecc). Esempio di utilizzo:

  utilizzo dell'utility per regex

  ```
  let regex = this.jsUtility.regex('email');
  if (!regex.test('aa.bb@test')) {
    // email non valida
  }
  ```
- **getUTC**: abbiamo già visto questo metodo nella sezione trasformatori/date-utc.
- **data**: un utile metodo per formattare una data nel formato di lingua dell'utente, specificando i parametri desiderati.
- **time**: come l'opzione precedente ma restituisce un orario anziché una data.
- **nations**: restituisce un array di oggetti contenenti nomi e sigle delle nazioni mondiali secondo lo standard ISO 3166-1 alpha-2

Consultare il file per l'elenco completo.

JSON-API

Abbiamo già visto questo file nella sezione ”*Query complesse in JSON:API*”.

PUSH-CALLBACK e PUSH-NOTIFICATIONS

Analizzeremo approfonditamente questi servizi nella sezione dedicata alla notifiche push.

SESSION e SESSION ACCOUNT

Necessari all'implementazione del plugin ember-simple-auth, non avremo bisogno di modificare in alcun modo questi servizi ma li importeremo nella maggior parte dei nostri elementi Javascript (componenti, gestori di rotta, controller…).

Vediamo alcuni dei casi d’uso più comuni:

- **Variabili di sessione**
  Se vogliamo memorizzare dei dati che siano automaticamente rimossi al logout, il servizio di sessione ci consente di farlo e provvederà alla loro cancellazione quando l’utente si disconnette. A tal scopo useremo il codice seguente:

Memorizzare e recuperare dati di sessione

```
this.session.set('nomeVariabile', 'valore'); // memorizzo il dato
this.session.get('data.nomeVariabile'); // recupero il dato
```

  Lo stesso codice è anche utilizzato al login per memorizzare in sessione i dati dell’utente. Potremo quindi utilizzare this.session.get per recuperare i seguenti dati (in grassetto quelli più comuni):

Dati dell’utente memorizzati in sessione	
access_token	Access-token ottenuto al login
associatedTenants	Array di oggetti che elenca i Tenant a cui l’utente é associato. Ogni oggetto ha questa struttura: *{tenantId: 1, name: 'MASTER TENANT'}*
authorizationExpires	Data di scadenza dell’access-token
authorizationExpiresIn	Durata dell’access-token, in minuti
authorizationRefreshExpires	Data di scadenza del refresh-token
authorizationRefreshExpiresIn	Durata del refresh-token, in minuti
currentTenantActive	Booleano che indica se il tenant corrente è attivo
email	E-mail dell’utente

firstName	Nome dell'utente
id	Guid dell'utente
lastName	Cognome dell'utente
permissions	Array di stringhe contenente i claims dell'utente.
profileImageId	Guid dell'eventuale immagine-profilo dell'utente
refresh_token	Refresh-token ottenuto al login
tenantId	ID del Tenant corrente
termsAcceptanceDate	Data di ultima accettazione di *Termini e Condizioni* e *Privacy Policy*

- Gestori di rotta
  Per limitare l'accesso ad una pagina ai soli utenti loggati, o solo a quelli non loggati, importeremo il servizio di sessione ed useremo i metodi *requireAuthentication* o *prohibitAuthentication* come mostrato negli esempi che seguono:

Accesso ad una rotta per i soli utenti loggati

```
export default class AuthenticatedRoute extends Route {
 @service session;

 async beforeModel(transition) {
   await this.session.setup();
   this.session.requireAuthentication(transition, 'login'); // solo utenti logga
 }
}
```

Accesso a una rotta per i soli utenti non loggati

```
export default class LoginRoute extends Route {
 @service session;

 async beforeModel() {
   await this.session.setup();
   this.session.prohibitAuthentication('');
 }
}
```

- **Verificare se l'utente è loggato**
  Possiamo verificarlo attraverso il metodo *isAuthenticated* del servizio session. Ecco un esempio:

Utilizzo del servizio session per verificare se l'utente è loggato

```
if (!this.session.isAuthenticated) {
   this.router.transitionTo('login');
}
```

- **Redirect post-login alla rotta pre-login**
  Questa è un'utile funzionalità del framework. Ipotizziamo che un utente non loggato cerchi di raggiungere una rotta riservata agli iscritti (cioè nel cui gestore di rotta abbiamo usato il metodo *requireAuthentication*). Ovviamente verrà rediretto alla pagina di login (i più attenti avranno capito che in questo caso il redirect è dipeso dal fatto che non abbiamo aggiunto la rotta all'array *unloggedEnabledPages*. Se lo facessimo, il redirect avverrebbe comunque ma per opera di *requireAuthentication*). Ipotizziamo ora che l'utente faccia effettivamente l'accesso e di volere che, dopo il login, sia automaticamente rediretto alla pagina che aveva tentato di raggiungere prima del login. Nel servizio di sessione abbiamo implementato questa azione custom, che si predispone facilmente in due passaggi:
  - aggiungere la rotta all'array *unloggedEnabledPages* di routes/application.js
  - aggiungere il metodo ***setAfterLogin*** all'hook model del gestore di rotta di interesse:

```
async beforeModel(transition) {
   await this.session.setup();

   // Memorizzo l'intento di transizione per poterlo utilizzare dopo l'accesso
   await this.session.setAfterLogin(transition);

   this.session.requireAuthentication(transition, 'login');
}
```

- **forzare il logout**
  Benché il framework gestisca già il logout, nell'eventualità in cui vogliate predisporre un logout custom ci sono 2 diversi metodi del servizio session:
  - il metodo custom *session.invalidateSessionGeneral()* che mostra una *modale* di conferma;
  - il metodo *session.invalidate()* nativo di ember-simple-auth che forza immediatamente il logout.

PERMISSIONS

Uno dei servizi più importanti del framework. Non è presente nella cartella services/ poiché, come già acquisito dalla relativa documentazione, viene installato tramite addon di Ember. Non entreremo nei dettagli implementativi, ci basterà sapere che il login (come già visto) restituisce i claims dell'utente che vengono poi passati a ember-permissions. La visibilità delle pagine, delle voci nel menù laterale e di molti altri elementi del framework si basa proprio su questo servizio.

Nei nostri file Javascript possiamo verificare se un utente ha uno o più permessi importando il servizio ed usando il metodo *hasPermissions* come segue:

Verificare i permessi da Javascript

```
@service permissions;

// ...

if (this.permissions.hasPermissions(['canSeeRightBAr', 'canSeeAllTenants'])) {
   // l'utente ha i suddetti permessi
}
```

Per effettuare la verifica nei file HBS, invece, useremo un apposito helper predisposto dall'addon (e che quindi non troviamo nella cartella helpers/) come da esempio:

Verificare i permessi da HBS

```
{{#if (has-permissions 'canSeeRightBar' 'canSeeAllTenants')}}

{{/if}}
```

Vale la pena approfondire il concetto di *permesso* o *claim*. Il framework ne prevede 3 tipologie:

- **permessi di crud**
  Sono i permessi che definiscono quali operazioni di CRUD possono essere eseguite. Più esattamente, per ogni Ruolo e per ciascuna Entità, possiamo definire se è possibile creare, leggere, aggiornare e/o cancellare i record.
- **permessi di rotta**
  Possiamo definire per ogni Ruolo quali sono le rotte accessibili.
- **permessi custom**
  Abbiamo inoltre la possibilità di definire dei claims custom e di associarli ai ruoli di nostro interesse. Possiamo quindi abilitare o nascondere una specifica funzionalità del sito verificando nel codice se l'utente ha lo specifico permesso.

Vediamo come poter gestire queste tipologie:

Permessi Custom.

Apriamo il menù **Impostazioni**/**Autorizzazioni** e dopo aver scelto il Tenant selezioniamo la voce "**Associazioni R÷P**" (R÷P sta per ruoli ÷ permessi).
Per ciascun Ruolo abbiamo la possibilità di digitare il permesso custom nel campo "Nome del Permesso".
Quando lo stesso permesso deve essere associato a molti ruoli, si può velocizzare l'assegnazione ricorrendo a un'altra features del framework. Apriamo dunque il menù **Impostazioni**/**Setup**/**Sicurezza** e individuiamo la voce "*Inserire i Claims di default*". Troviamo una serie di claims custom che sono funzionali alle diverse features del framework. Inseriremo qui gli eventuali permessi che riteniamo debbano essere associati a diversi ruoli.
Torniamo quindi nel menù Impostazioni/Autorizzazioni/"Associazioni R÷P" e per ciascun ruolo di nostro interesse, selezioniamo nella select "Permessi Predefiniti" il claim appena creato.

Permessi di Rotta.

Nel menù **Impostazioni**/**Autorizzazioni**/**Rotte** definiamo, per ciascun ruolo, quali rotte sono accessibili e quali no (è ovvio che parliamo delle sole pagine riservate agli utenti loggati).
E' essenziale che il menù **Impostazioni**/**Setup**/**Rotte** sia popolato con tutte le rotte definite in router.js (sempre limitatamente a quelle per utenti loggati), come dovrebbe già essere se abbiamo seguito le indicazioni già viste nel primo capitolo di Regole Generali/Routers.

Permessi di CRUD.

Apriamo il menù **Impostazioni**/**Autorizzazioni**/"**Permessi CRUD**" e, per ciascun ruolo e per ciascuna entità, selezioniamo quali delle 4 operazioni di CRUD sono concesse.
Per meglio distinguere le entità, nel menù **Impostazioni**/**Setup**/**Entità** definiamo per ciascuna rotta (l'elenco è fornito direttamente dal back-end) un titolo e una descrizione che rendano l'entità più facilmente riconoscibile.

SITE-LAYOUT

È il servizio su cui si basa il tema e la sua customizzazione. Difficilmente avremo bisogno di importarlo nei nostri componenti o controller. Nell'eventualità in cui volessimo assegnare ai nostri elementi lo stesso stile dell'header e del footer impostato da Setup, faremo riferimento alle seguenti variabili tracked del servizio:

- **headerBackground**: stringa contenente la classe di sfondo (bootstrap) assegnata all'header;
- **headerLight**: stringa che indica la tonalità del testo dell'header. I possibili valori sono:
    - white: associato a sfondi scuri, indica che il testo deve essere chiaro;
    - black: associato a sfondi chiari, indica che il testo deve essere scuro;
    - "": stringa vuota associata a sfondi bianchi, indica che il testo deve essere scuro.
- **sidebarBackground**: stringa contenente la classe di sfondo (bootstrap) assegnata al footer;
- **sidebarLight**: come *headerLight*.

SITE-SETUP

Anche questo è un servizio accessorio contenente tutte le impostazioni di Setup, che vengono caricate dal DB all'avvio del sito. Difficilmente avremo bisogno di importarlo nei nostri componenti o controller (ma è largamente utilizzato nei componenti *core*).

STATUS-SERVICE

Abbiamo già parlato di questo servizio nella sezione "*Passaggio di parametri tra pagine*". In generale ci può tornare utile per memorizzare dati come se si trattasse di variabili globali. La raccomandazione è di utilizzarlo il meno possibile e solo se strettamente necessario, avendo in più l'accortezza di cancellare (o meglio rendere *null*) le variabili in esso valorizzate quando il loro valore non è più necessario poiché, essendo un servizio, è ovviamente un singleton!

TRANSLATION

Abbiamo già parlato di questo servizio nella sezione *Internazionalizzazione*.

WEB-SOCKET

Questo servizio gestisce la connessione al web-socket per la ricezione delle notifiche push via web. Per quanto preveda la possibilità di inviare messaggi sul web-socket tramite l'apposito metodo *sendMessage*, l'utilità principale è quella di ricevere i messaggi web-socket inviati dal back-end e di passarli automaticamente al servizio push-callback. In altre parole è un servizio autonomo di cui non dovremo occuparci.

TASK: EMBER-CONCURRENCY

Probabilmente il plugin più utile di tutto il framework e di certo uno dei più utilizzati. La lettura della relativa documentazione vi avrà già fatto capire la sua **magnificenza** e senza dubbio lo avrete già etichettato come il vostro più fedele aiutante!!! …Se non è così, tornate a leggere l'intera documentazione.

Data la sua importanza nel framework, la conoscenza del plugin è essenziale e poiché la documentazione é molto chiara, non ci occuperemo qui del suo funzionamento.

È però importante avere chiara la differenza tra i diversi metodi operativi: ***restartable***, ***enqueue***, ***drop***, ***keepLatest*** (se questi non sono chiari, come detto, tornate a leggere la documentazione).

La semplificazione nel migliorare la user-experience è ovvia, e ciò che quasi stupisce è quanto codice faccia risparmiare rispetto a quando si cercava di ottenere la stessa UX senza di esso!
Anche per questo è normale pretendere che nello sviluppo di applicativi con il framework venga posta la dovuta attenzione alla UI e alla UX.

Un aspetto che conviene chiarire è che non sempre è conveniente abusarne. Consideriamo questo esempio:
Un nostro componente *my-component* prevede un form di inserimento dati. Vogliamo che alla pressione del pulsante di salvataggio venga mostrata una finestra di *confirm* e, solo dopo conferma, effettueremo una chiamata alle API per salvare nel DB i dati inseriti. Un loader deve indicare che il salvataggio è in corso.

Verosimilmente avremo bisogno di due metodi, uno per mostrare la finestra modale (magari dopo aver controllato l'integrità dei dati immessi), ed uno per avviare l'effettiva chiamata al server.
Per quest'ultimo metodo ha perfettamente senso utilizzare un task di Ember-concurrency, e ciò ci consente facilmente di mostrare un loader di caricamento finché la chiamata API non si conclude.
Per il primo metodo, però, non avrebbe alcun senso né utilità l'utilizzo di un task ed anzi comporterebbe un immotivato aumento del codice post-build. Del resto, finché l'utente non decide se confermare o meno il salvataggio, vedrà la finestra modale e non occorre mostrare il loader.
Quindi, usiamo i task ogni volta che effettivamente sono necessari.

E non dimentichiamoci di usare un try-catch all'interno di essi per gestire le eccezioni e mostrare i toast di errore. Semplifichiamo il tutto analizzando il codice:

Estratto dell'HBS di my-component

```
<button class="btn-hover-shine btn btn-light" type="button" {{on "click" this.save}}>
   <i class="fa {{if this.saveConfirmed.isRunning "fa-spinner fa-spin" "fa-save"}} mr-2"></i>
   SALVA
</button>
```

Notiamo che al click chiamiamo il metodo di conferma (che non è un task) e che il loader è stato inserito all'interno del pulsante, semplicemente modificando la classe dell'icona quando il task (il secondo metodo) è in esecuzione.

Estratto del Javascript di my-component

```
@action
save() {
   // eventuale codice di verifica dei dati immessi nel form

   // mostriamo la modale
   this.dialogs.confirm(
     `CONFERMA`,
     `Confermi il salvataggio?`,
     () => {
       // in caso di conferma avviamo il task
       this.saveConfirmed.perform();
     },
     null,
     ['Conferma', 'Annulla']
   );
};

saveConfirmed = task({ drop: true }, async () => { // notare il DROP
   try {
       let newRecord = this.store.createRecord('my-model');
       // valorizzare newRecord con i dati del form
       //...

       // memorizzo il record nel DB
       await newRecord.save();
   } catch (e) {
      console.error(e);
   }
});
```

Qui dobbiamo notare alcuni aspetti:

- il metodo *save* è decorato con @action in quanto richiamabile da HBS. I task non devono invece mai avere il decoratore (nemmeno se richiamabili da HBS) altrimenti riceveremo un errore.
- È essenziale aggiungere l'await al salvataggio altrimenti il task terminerebbe immediatamente (senza attendere l'avvenuto salvataggio) e non vedremo il loader.
- Abbiamo scelto l'opzione DROP per il task. Chiaramente nelle applicazioni reali useremo l'opzione che più si adatta alla funzionalità implementata.
- È essenziale usare il try-catch.

NOTIFICHE PUSH

INTRODUZIONE

Il framework prevede un sistema di notifiche push sia per la versione Web che per la versione Cordova.

La gestione delle notifiche riguarda sia il back-end, che si occupa di inviarle, che il front-end che si occupa di riceverle, di visualizzarle e di eseguire eventuali call-back.

Per la versione Web vengono inviate tramite web-socket mentre per le App sono gestite dalla piattaforma Firebase di Google ma, per semplificare il lavoro dello sviluppatore, convergono entrambe sullo stesso codice di gestione strutturato in tre servizi custom appositi:

- **push-notifications**: si occupa di tutte le attività di gestione del sistema di notifiche su App tra cui: richiesta dei permessi al sistema operativo mobile, inizializzazione, invio del token al server, richiamo delle callback, ecc; non avremo bisogno di modificarne il codice.
- **web-socket**: si occupa di tutte le attività di gestione del sistema di notifiche su Web e anche su App se le notifiche push non sono state attivate da Setup. Non avremo bisogno di modificarne il codice.
- **push-callback**: contiene il codice custom per la gestione di ogni singola notifica. Questo è l'unico file in cui andremo a scrivere il codice di call-back delle notifiche push.

Prima di entrare nei dettagli delle call-back, forniamo un quadro generale di come il framework gestisce la sezione notifiche.

Anzitutto chiariamo il concetto di **call-back** associate alle notifiche, utilizzando un esempio.
Ipotizziamo di ricevere una notifica informativa sul meteo odierno e di non aver particolari pretese alla sua ricezione; ci limiteremo cioè a visualizzare il testo della notifica all'interno di un alert. Questa è una notifica senza call-back, che ci limitiamo semplicemente a visualizzare.

Potremmo però migliorare il servizio. Ipotizziamo che la notifica ricevuta ci informi sulle previsioni meteo di un giorno futuro e che l'utente possa scegliere, attraverso una *confirm*, di passare alla pagina di dettaglio meteo di quel giorno. Avremo dunque bisogno di una call-back che fa comparire la *confirm* e che, se l'utente conferma, effettua redirect alla pagina opportuna.
Avremo bisogno anche di un'altra informazione, cioè la data del giorno da visualizzare. Questo dato deve essere fornito dal back-end come parte integrante della notifica.

Un ultimo approfondimento riguarda le notifiche push ricevute sull'App. Qui possiamo distinguere tra notifiche in background (ricevute quando l'app non è attiva o non è in primo piano) e notifiche in foreground (ricevute quando l'app è aperta), che essendo gestite con due flussi diversi, richiedono che le nostre callback prevedano tale distinzione. In particolare:

- **notifica in background**: la notifica è gestita da sistema operativo (Android e iOS), che provvede autonomamente a mostrarne il testo e ad emettere il suono di avviso. Cliccando sulla notifica, si apre l'app e viene invocata la relativa callback, passandole un parametro che ci informa che la ricezione è avvenuta in background.
- **notifica in foreground**: la notifica è gestita interamente dal nostro applicativo che invoca la callback passandole un parametro che ci informa che la ricezione è avvenuta in foreground. In

questo caso la callback deve occuparsi del suono di notifica, di mostrare il testo e di eseguire eventuali azioni aggiuntive.

Chiarito il concetto di call-back passiamo all'attivazione del servizio notifiche che avviene configurando due voci dedicate nel menù *Impostazioni*/*Setup*/*Comunicazioni*:

- "**Attivare le notifiche interne?**": durante l'avvio dell'applicativo il framework verifica se le notifiche interne sono attivate e in caso affermativo predispone una serie di features dedicate:
  - mostra un'icona nell'header che consente di accedere alle notifiche non lette e allo storico delle notifiche ricevute;
  - predispone il servizio web-socket all'ascolto di eventuali notifiche;
  - attiva la voce di menù seguente.
- "**Attivare le notifiche push per l'App?**": se non attiviamo questa voce, le notifiche in app saranno gestite come quelle su web, ovvero tramite web-socket. Generalmente usiamo questa scelta solo se l'applicativo non prevede l'app.
  Quando attiviamo questa voce, invece, il framework predispone una serie di features aggiuntive:
  - attiva il codice associato al plugin *cordova-plugin-firebasex* all'ascolto delle notifiche push (questo richiede sia la configurazione del backend con le chiavi Firebase di attivazione sia, lato frontend, l'installazione del suddetto plugin e dei file di attivazione necessari per l'App, in assenza dei quali la build Cordova fallirà).
  - disattiva l'ascolto del web-socket sull'app (che sarebbe ridondante).

Altro aspetto da conoscere riguarda la possibilità di visualizzare, mediante un'apposita pagina raggiungibile dal menù delle notifiche presente nell'header, lo storico delle notifiche ricevute. Si tratta di un elenco in cui consultare le notifiche ricevute e che permette, di conseguenza, di lanciare le call-back associate. La gestione di questa pagina è del tutto automatizzata e verranno richiamate le callback già definite nel servizio push-callback, che ora analizziamo.

SERVIZIO PUSH-CALLBACK

Di default il servizio in questione contiene un unico metodo, *refreshHeaderNotifications*, che richiameremo in tutte le callback per aggiornare l'elenco delle notifiche presente nell'header.

Metodo *refreshHeaderNotifications* del servizio notifications-callbak.

```
async refreshHeaderNotifications(res, translation, store, session, header,
getAdvices, router)
```

Lato backend ogni singola notifica è inviata definendo un apposito payload. Uno dei campi salienti è il ***PushType***, contenente una stringa che la identifica univocamente.
Lato frontend dovremo definire una callback per ciascuna notifica prevista, ovvero dovremo definire un metodo nel servizio push-callback il cui nome è lo stesso di PushType.

Il servizio push-notifications provvederà autonomamente ad invocare il metodo corrispondente inviandogli lo stesso payload del metodo *refreshHeaderNotifications*.

In altre parole tutte le nostre callback avranno un nome uguale al *PushType* inviato dal backend ed i 7 parametri *data*, *translation*, *store*, *session*, *header*, *getAdvices*, *router*. Gli ultimi 6 sono semplicemente

istanze degli omonimi servizi, che potremo utilizzare se necessario. Ad esempio se dobbiamo utilizzare il servizio store per effettuare chiamate API all'interno della call-back, abbiamo il servizio già istanziato nella firma del metodo.
Analizziamo quindi il parametro restante, ***res***. Si tratta di un oggetto con i seguenti campi:

- **tap**: ha valore "**foreground**" quando la notifica è stata ricevuta in foreground, altrimenti la notifica è stata ricevuta in background. Un utilizzo tipico è quello di emettere il suono di notifica:

```
cb1(res, translation, store, session, header, getAdvices, router) {
   if (!res.tap || res.tap === 'foreground') { // L'app è in uso e in primo pian
       this.audio.play('new_message', this.cordovaDevice);
   }
   // ...
}
```

- **pushData**: Si tratta di un oggetto serializzato (dovremo dunque deserializzarlo per accedervi) contenente i dati aggiuntivi forniti dal backend, ove previsti.

È importante segnare nei commenti il payload atteso, cioè quali dati vengono forniti dal backend per ciascuna notifica, in modo che in futuro potremo recuperare tale informazione senza dover ispezionare il codice backend.

Ecco dunque un esempio di una callback ben strutturata:

```
startGame(res, translation, store, session, header, getAdvices, router) {
   /**
    * PAYLOAD ATTESO:
    * duration = durata del timer in secondi
    * startingTime = DateTime di inizio gioco
   */

   let datas = JSON.parse(res.pushData);

   this.statusService.gameDuration = datas.duration;
   this.statusService.gameStart = new Date(datas.startingTime);

   if (!res.tap || res.tap === 'foreground') {
     this.audio.play('new_message', this.cordovaDevice);

     Swal.fire({
       title: 'INIZIO GIOCO',
       html: `Inizia a rispondere alle domande!`,
       icon: 'success',
       showCancelButton: true,
       confirmButtonColor: '#3085d6',
       cancelButtonColor: '#6c757d',
       confirmButtonText: 'Vai al gioco',
       cancelButtonText: 'Ignora',
     }).then((result) => {
       if (result.isConfirmed) {
         router.transitionTo('quiz');
       }
     });
   } else {
     router.transitionTo('quiz');
```

```
    }
}
```

PREPARARE L’APP PER CORDOVA

Quando lo sviluppo del nostro applicativo è terminato, ed abbiamo testato che la visualizzazione su *mobile* (sia tablet che smartphone) sia corretta, possiamo effettivamente procedere alla build per Cordova che si articola in due fasi:

1. lanciare il comando **ember build environment=production**
   Come noto, questo ridurrà la nostra applicazione ad un unico file Javascript, un unico file Css e ad un unico file html, oltre ovviamente ad un’unica cartella che contiene i file accessori (immagini, suoni ecc). Tutto ciò verrà generato nella cartella **dist/** dell’applicativo, ovvero all’interno della macro-cartella WEB.
2. Copiamo tutto il contenuto della cartella dist/ all’interno della cartella www/ della macro-cartella APP.

Tutto qui! Ora potremo usare il *prompt dei comandi* (o *Terminale* su Mac) all’interno della cartella APP per lanciare i comandi Cordova.

ULTIMI CONSIGLI

- Quando abbiamo effettivamente terminato lo sviluppo e pushato tutto su GIT, è buona norma cancellare l'intero repository dal PC in quanto la cartella node_modules occupa uno spazio considerevole.
  Se in futuro dovremo apportare ulteriori sviluppi, potremo sempre ri-clonare il progetto dal repository GIT.
- Questa guida è in continuo aggiornamento perché le features del framework potrebbero migliorare o aumentare nel tempo.
- La vostra partecipazione è fondamentale per il miglioramento continuo di questa guida. Sentitevi liberi di segnalare eventuali errori o carenze riscontrate. Ogni feedback contribuisce non solo a correggere imprecisioni, ma anche a perfezionare e arricchire il contenuto, garantendo che sia il più accurato e utile possibile.
  Quando condividete le vostre osservazioni, fornite dettagli che possono aiutarci a comprendere il contesto e la natura del problema. Questo non solo aiuta a identificare rapidamente le soluzioni più efficaci, ma arricchisce anche la base di conoscenza comune, beneficiando l'intera comunità che utilizza questa guida.
  Il vostro contributo, dunque, va oltre la semplice segnalazione di un errore; è un atto di collaborazione che aiuta a costruire una risorsa sempre più affidabile e completa per tutti. Ricordate, il miglioramento continuo dei contenuti si basa significativamente sulle vostre esperienze e suggerimenti.