beSharp. we make IT run.

GO SERVERLESS! PARTE 3: SOFTWARE EVENT DRIVEN E TRIGGER

AWS Lambda | CI/CD | Continuous Delivery | DevOps | Serverless

beSharp | 10 Ottobre 2018

Leggi la prima parte | Leggi la seconda parte

Questa è la **terza e ultima parte** della serie di tre articoli su come costruire una piattaforma di file sharing completamente serverless. In questo articolo presenteremo la **soluzione software completa di codice** sia per il front-end che per il back-end.

Analizzeremo nel dettaglio l'architettura software, le scelte di progettazione adottate e l'utilizzo dei trigger lato AWS che permette alla soluzione di funzionare correttamente.

Come anticipato nel primo articolo della serie, ci occuperemo anche delle **pipeline di Continuous Integration e Continuous Delivery**, in modo da poter eseguire deploy ripetibili ed affidabili semplicemente effettuando un push sul repository.

Nell'articolo precedente abbiamo creato a mano una infrastruttura analoga a quella che andremo a raffinare e definire. Di quegli elementi bisognerà eliminare l'API Gateway che abbiamo creato come test, in quanto il deploy delle risorse utilizzate dal back-end sarà gestito dalla pipeline automatica.

Per terminare il deploy della soluzione occorrerà:

1. **Scaricare il sorgente del front-end e del back-end**
2. **Configurare due repository di CodeCommit per entrambi**
3. **Configurare le pipeline**
4. **Fare un "Push" del codice sui repository per avviare il processo di deploy automatico**

Ecco i repository per clonare il codice sorgente:

Front-end: https://github.com/besharpsrl/serverless-day-fe
Back-end: https://github.com/besharpsrl/serverless-day-be

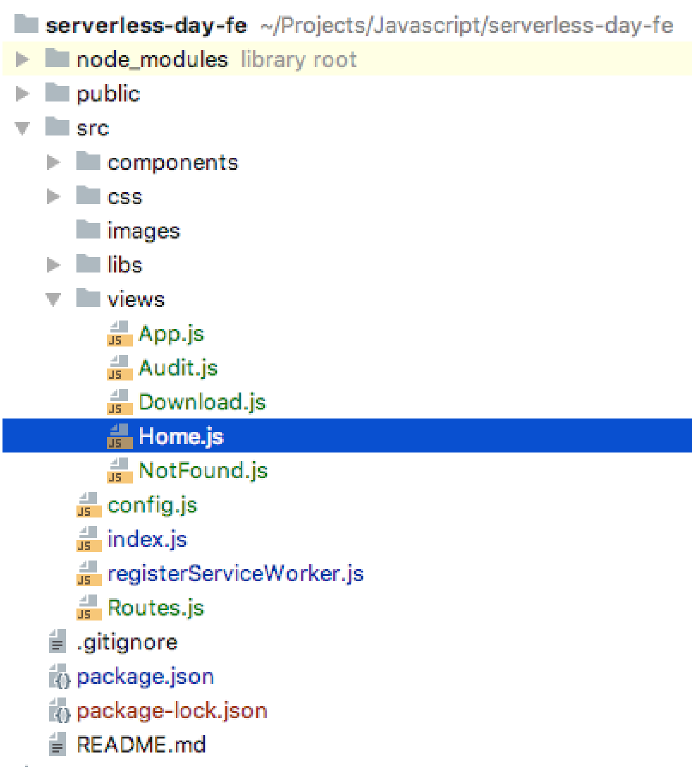
Cominciamo quindi presentando le soluzioni adottate nello sviluppo del software

Il front-end

Una parte fondamentale dell'applicazione è rappresentata dal front-end, applicazione scritta in **React.js** che permette all'utente di loggarsi in un'area sicura, caricare, condividere e scaricare documenti.

Seguendo questa guida sarà possibile integrare il front-end con il nostro back-end serverless in modo da permettere alle azioni compiute dall'utente in interfaccia di riflettersi realmente su S3 e sulle tabelle di DynamoDB.

Per cominciare, possiamo **clonare il codice sorgente** dal repository oppure **scaricare lo zip**. La struttura del pacchetto deve rispecchiare quanto segue:



Assicuriamoci di essere nella root di progetto di modo da poter lanciare i seguenti comandi:

```
npm install
npm start
```

Possiamo quindi procedere a **configurare la libreria AWS Amplify**.

Amplify è una libreria consigliata da AWS che solleva il programmatore dalla gestione nel dettaglio di alcuni aspetti del ciclo di vita dell'applicazione legati ad Amazon AWS, in particolare, nel nostro caso specifico, i servizi **Cognito, S3 e API Gateway**.

Per configurare correttamente il tutto abbiamo provveduto a creare un semplice custom component che ha la funzione di **file di configurazione**:

```
export default {
    s3: {
        REGION: "<YOUR REGION>",
        BUCKET: "<YOUR BUCKET NAME>"
    },
    apiGateway: {
        NAME: "<A UNIQUE NAME FOR YOUR API CALL>",
        REGION: "<YOUR REGION>",
        URL: "<YOUR COMPLETE URL TO API GATEWAY>"
    },
    cognito: {
        REGION: "<YOUR REGION>",
        USER_POOL_ID: "<YOUR USER POOL ID>",
        APP_CLIENT_ID: "<YOUR APP CLIENT ID>",
        IDENTITY_POOL_ID: "<YOUR IDENTITY POOL ID>"
    },
    app: {
        URL: "<YOUR APP URL>"
    }
};
```

Come si può vedere, tutti i parametri di configurazione di Aws Amplify sono contenuti qui e vanno sostituiti con **id** e **arn** delle risorse create nell'articolo precedente.

Una volta completata la configurazione ed installate tutte le dipendenze, possiamo **avviare l'applicazione** e visualizzare la seguente schermata di login:

Sign in to your account

Username *

Enter your username

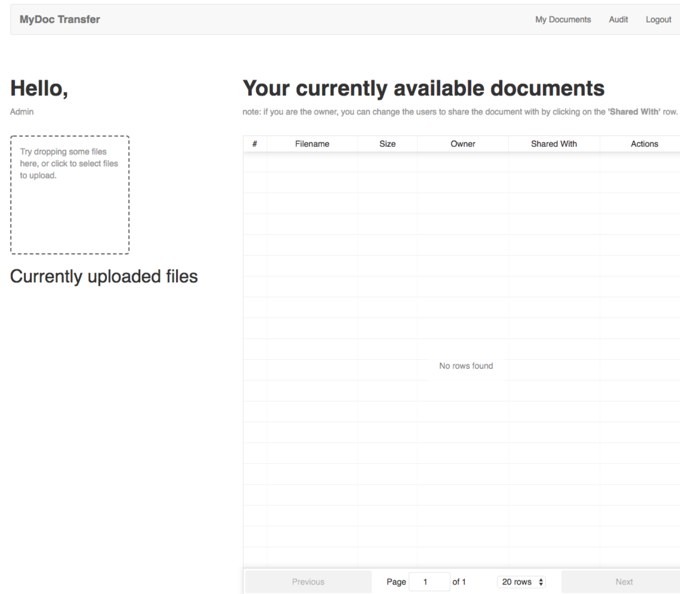
Password *

Enter your password

Forget your password? Reset password

SIGN IN

Se aggiungiamo manualmente degli utenti nella CognitoUserPool sarà possibile effettuare login e visualizzare la **schermata principale dell'applicazione**.



Il codice fornito è ampiamente commentato per facilitarne la comprensione. Una caratteristica notevole di React.js è il meccanismo di **auto reload**. Durante lo sviluppo, ogni modifica al codice, allo stile css o alle librerie si riflette quasi immediatamente nella applicazione visualizzata nel browser.

Un'altra nota particolare riguarda l'approccio alla **gestione del DOM** da parte di React.js. In un front-end Javascript tradizionale, siamo noi ad occuparci direttamente della modifica del DOM. Con React.js, invece, questa operazione viene **gestita interamente dal Framework**.

Ogni componente di React.js è assimilabile ad un oggetto Javascript complesso, il cui contesto, rappresentato da *this***,** permette di accedere ad una chiave chiamata *state* che rappresenta lo stato del componente. Ogni variazione degli attributi del componente provoca un cambiamento nel metodo *render*. Questo significa che la manipolazione diretta del DOM non è più necessaria ed è gestita direttamente dal framework alla modifica dello *state*.

A tal proposito, si può vedere nel componente Table.js come tale sistema permetta di visualizzare dei modali che altrimenti non sarebbero gestibili con i metodi tradizionali.

Back-end

La parte di back-end applicativa viene servita mediante **API Gateway e Lambda**.

Entrambe queste risorse possono richiedere modifiche ad ogni deploy (in base alle modifiche effettuate al sorgente e alla struttura del back-end), per questa ragione abbiamo adottato una scelta che ci permette di ottenere in modo semplice ed efficace una gestione automatizzata dello stack AWS.

La scelta riguarda il framework impiegato, che è ricaduta su **Chalice**: https://github.com/aws/chalice. Si tratta di un **framework Python progettato per AWS**, che incorpora un router simile a quello di Flask e un sistema di decorator che implementa le integrazioni con i servizi AWS supportati.

Mediante semplici dichiarazioni possiamo aggiungere al nostro back-end **trigger AWS**, per invocare metodi in risposta a determinati eventi, oppure configurare in modo gestito integrazioni complesse come quella di CognitoUserPool e degli authorizer di API Gateway.

All'interno del repository troverete la **soluzione completa con tutto il codice**, i file di configurazione e quelli di utility. Non è necessario preparare environment o installare dipendenze perchè abbiamo incluso Dockerfile e docker-compose per rendere l'avvio dell'applicazione automatico.

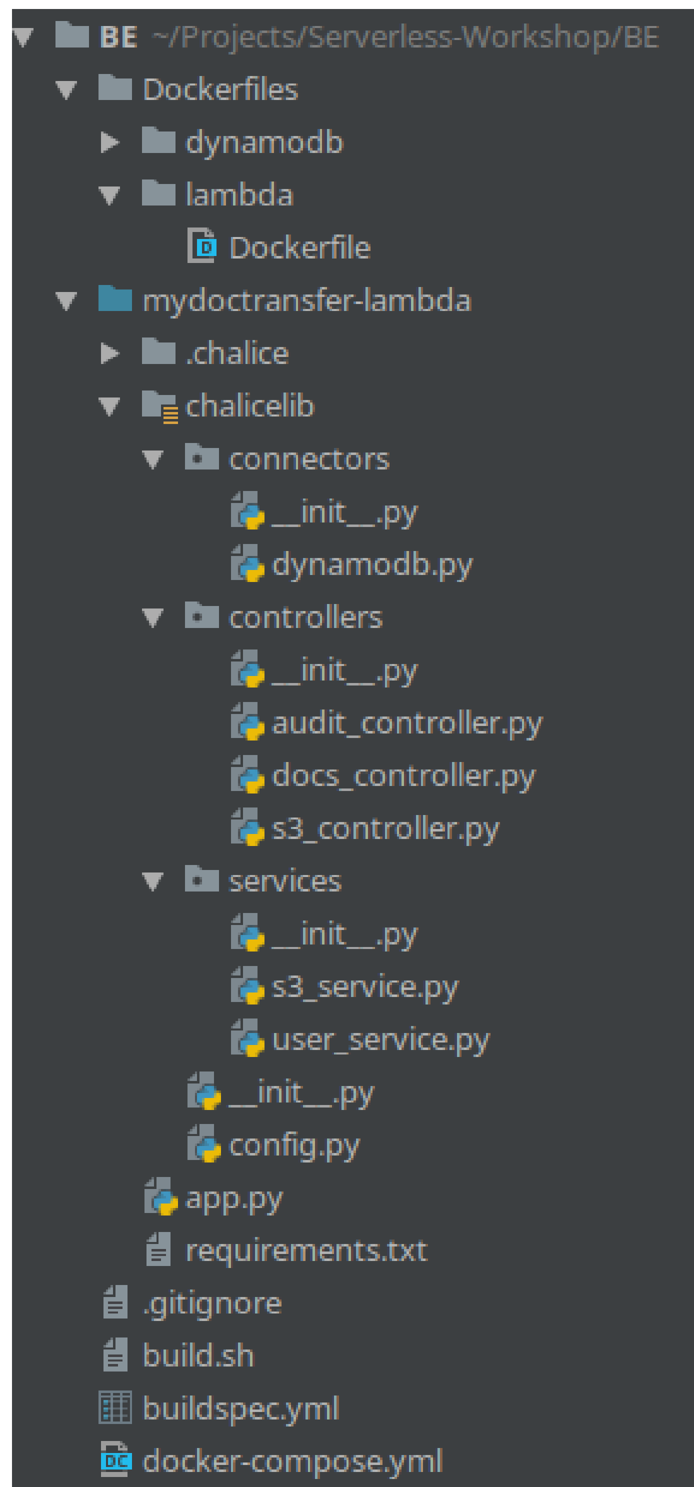
Per avviare l'applicazione usando Docker basta entrare nella root della soluzione e invocare

```
docker-compose up
```

Nel caso si preferisca invece evitare docker e installare le dipendenze nel proprio sistema, occorre installare python 3.6, pip e successivamente tutte le dipendenze contenute in requirements.txt. Consigliamo vivamente di utilizzare un VirtualEnv dedicato nel caso in cui Docker non sia un'opzione praticabile.

```
pip3 install -r requirements.txt --user
```

Di seguito la struttura del progetto:



Da notare la presenza della cartella chalicelib, che è utilizzata per aggiungere classi custom al progetto in un modo riconosciuto dal framework. Salvare i moduli custom all'interno di questa cartella serve a sfruttare la funzione di Chalice in grado di generare automaticamente un **pacchetto pronto per il deploy su AWS Lambda**.

Come per il front-end, è presente un file config.py per configurare tutti i parametri dipendenti dall'account AWS che l'applicazione e Chalice hanno bisogno di conoscere.

Abbiamo diviso i moduli in servizi, controller e connector; questa distinzione è utile per mantenere ordine e favorire il riutilizzo del codice. Mediante questa organizzazione è possibile mantenere DRY il codice di app.py, che contiene di fatto tutte le rotte del back-end opportunamente decorate. Tutta la logica applicativa, al netto delle rotte e della gestione dell'output, si trova nei controller e nei servizi.

L’intera interfaccia di comunicazione con il database è stata implementata in un modulo separato (un connector), così da essere più facilmente modificabile, ottimizzabile e sostituibile.

Chalice è in grado provvedere al deploy e alla configurazione di API Gateway e Lambda; le configurazioni sono espresse in modo idiomatico nel codice applicativo.

Una volta pronto il progetto si può effettuare il deploy automatico semplicemente invocando

```
chalice deploy
```

Passiamo quindi ad analizzare la parte più interessante del progetto, ovvero **i decorator** che permettono di integrare in modo semplice i servizi di AWS usando trigger e configurazioni di API Gateway.

L’intera definizione delle rotte e dei trigger avviene in app.py, il file principale del back-end. Scorrendo il codice, è possibile notare un gran numero di decorator. Questi sono il meccanismo utilizzato dal framework per dichiarare le rotte e le integrazioni con AWS. Nel progetto abbiamo utilizzato tre decorator: **@app.route, @app.on_s3_event, @app.schedule.**

@app.route

Chalice effettua il deploy di buona parte del back-end (eccetto metodi di reazione ad eventi, come vedremo in seguito) in una sola funzione Lambda. Il framework include un router, che si occupa di invocare il metodo corretto ad ogni invocazione. Questo decorator si rivolge al router: indica di creare e gestire una rotta su API Gateway che risponde ad uno o più verbi HTTP. Il metodo che viene così “decorato” è quindi invocato dal router alle richieste che corrispondono alla rotta specificata.

Oltre a definire la rotta, il decorator accetta un numero di parametri per sfruttare al meglio l’integrazione con API Gateway. Ad esempio, è possibile configurare i settaggi per le CORS o l’integrazione con una CognitoUserPool per l’autenticazione delle API.

Esempio della definizione di un metodo autenticato, con settaggi CORS e un authorizer:

```
@app.route('/test, methods=['GET'], authorizer=authorizer,
cors=CORS_CONFIG)
def mymethod():
      return {“message”: “ok”}
```

@app.on_s3_event

In modo simile al decorator utilizzato per le rotte, è possibile indicare al framework che un determinato metodo deve essere invocato quando si verifica un evento su S3.

Nel nostro progetto sfruttiamo questo trigger per reagire e trasformare i file caricati dagli utenti. Il decorator prende in input il tipo di evento di interesse e il bucket di riferimento.

Esempio:

```
@app.on_s3_event(bucket=config.S3_UPLOADS_BUCKET_NAME,
events=['s3:ObjectCreated:*'])
def react_to_s3_upload(event):
    …
```

Dietro le quinte, questo provoca il deploy di una Lambda separata da quella principale, che ha come trigger il caricamento di un file su S3 e che viene invocata automaticamente da AWS ad ogni evento

```
s3:ObjectCreated:*
```

sul bucket specificato.

@app.schedule

Come è facile intuire dal nome, questo decorator indica di eseguire il metodo ad intervalli regolari o comunque secondo uno scheduler.

Crea e gestisce una regola su CloudWatch events per avviare una Lambda dedicata con il metodo da eseguire.

```
@app.schedule(Rate(5, unit=Rate.MINUTES))
def expire_files(event):
```

Pipeline di CD/CI

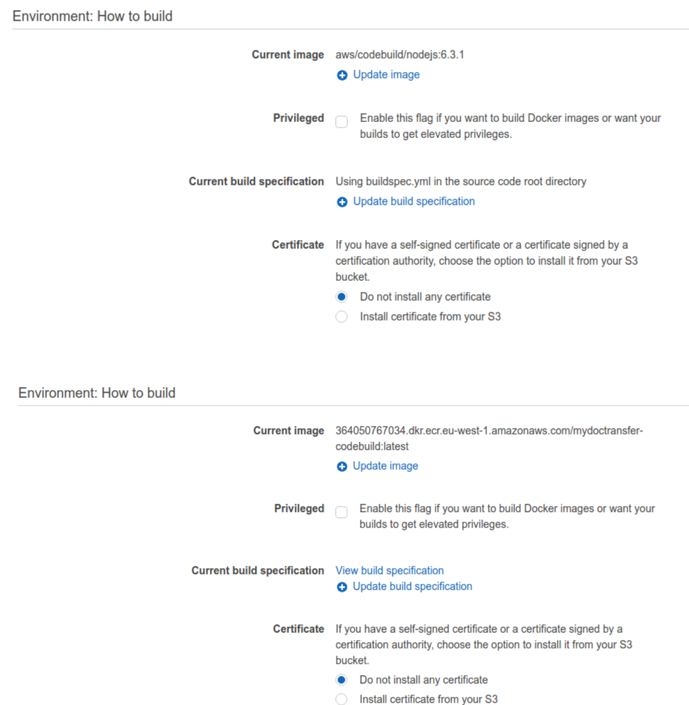
Sebbene sia possibile caricare il Front-End compilato sul bucket s3 preparato nei precedenti articoli, ed eseguire un chalice deploy per avere tutto correttamente configurato, vogliamo completare l'architettura accennando alle **pipeline automatiche per il deploy del codice**.

Abbiamo già parlato di CD/CI in questo articolo https://blog.besharp.it/2017/03/10/full-stack-continuous-delivery-su-aws/

Questa volta la soluzione sarà leggermente diversa.

Come prima cosa occorre creare due nuove applicazioni su CodeBuild, una per il front-end ed una per il back-end. Occorre configurare CodeBuild per utilizzare il file buildspec.yml presente nella root del repository.

Il CodeBuild di front-end può utilizzare un'immagine standard di nodejs, e quello del back-end una con Python3.6.



In ognuno dei repositories sono presenti i file buildspec.yml. Questi file definiscono i passi eseguiti da CodeBuild per produrre il pacchetto di cui effettuare il deploy ed anche le istruzioni di deploy stesse.

Una volta create le applicazioni su CodeBuild, basta creare altrettante Pipeline che facciano il source da CodeCommit e poi passino il codice al corrispondente CodeBuild. In nessuna delle due pipeline occorre specificare lo stadio di deploy, perchè per questo esempio abbiamo incluso le istruzioni nel buildspec.yml in modo che siano eseguite dopo il processo di build.

Di conseguenza, gli IAM role associati ai CodeBuild devono avere il permesso di caricare ed eliminare file da S3, gestire API Gateway e Lambda, e invalidare la distribuzione di CloudFront.

Analizzando i buildspec possiamo osservare le operazioni eseguite per il build e deploy delle applicazioni.

Cominciando dal Front-End che esegue i seguenti passi

```
- npm install
- npm run build
- cd ./build && aws s3 sync --delete . s3://$S3_BUCKET_NAME/
- aws cloudfront create-invalidation --distribution-id $DISTRIBUTION_ID --paths "/*"
```

In ordine:

1. Installare le dipendenze
2. Creare il pacchetto di React.js
3. Sincronizzare l'output della build con il bucket s3 del front-end (ovvero aggiornare i file presenti, aggiungere quelli nuovi ed eliminare quelli non più presenti)
4. Invalidare il contenuto della distribuzione CloudFront per rendere effettive le modifiche.

I passi eseguiti dal back-end invece sono molto diversi

```
    - rm -rf mydoctransfer-lambda/.chalice/deployed
    - mkdir mydoctransfer-lambda/.chalice/deployed
    - cd mydoctransfer-lambda/.chalice/deployed && aws s3 sync --delete s3://$RELEASE
S_S3_BUCKET/backend/chalice .
    - cd mydoctransfer-lambda && pip3.6 install -r requirements.txt --user
    - aws s3 cp s3://$CONFIG_S3_BUCKET/backend/$ENV/modsam.py .
    - cd mydoctransfer-lambda/.chalice && aws s3 cp s3://$CONFIG_S3_BUCKET/backend/$EN
V/config.json .
    - ./build.sh
    - cd mydoctransfer-lambda/.chalice/deployed && aws s3 sync --delete . s3://$RELEA
SES_S3_BUCKET/backend/chalice
```

Il cuore del processo è build.sh, che contiene solo una minima configurazione della shell e il comando chalice deploy.

I restanti passi servono a preparare l'ambiente locale. Oltre ad installare le dipendenze provvedono anche a persistere, mediante un bucket S3 di appoggio, lo stato dei deploy memorizzato in /.chalice/deployed.

A questo punto basta effettuare push sulle pipeline per effettuare deploy automatici di front-end e back-end ed avere finalmente pronta un'applicazione di file sharing completamente serverless.

Enjoy 🙂

Leggi la prima parte | Leggi la seconda parte

beSharp

Dal 2011 beSharp guida le aziende italiane sul Cloud. Dalla piccola impresa alla grande multinazionale, dal manifatturiero al terziario avanzato, aiutiamo le realtà più all'avanguardia a realizzare progetti innovativi in campo IT.

Get in touch

beSharp.it
proud2becloud@besharp.it

Copyright © 2011-2021 by beSharp srl - P.IVA IT02415160189