UniversaMente
una comunità per gli Atenei

UNIVERSAMENTE
CONTRATTO CRUI - RED HAT

PROFILO BLU

plurimedia

Red Hat

CRUI
Conferenza dei Rettori delle Università Italiane

1. Il contratto CRUI - RED HAT
2. Serverless Application - Red Hat
3. Esempi e casi d’uso - Plurimedia
4. Domande e risposte

Cosa contiene il contratto CRUI - RED HAT?

Sono presenti due package Red Hat: **Campus 1** e **Campus 2**
Oltre ad essi è possibile acquistare: **Ansible**, **RHEL add-on**, **TU** ed altro

Campus 1 contiene:

Red Hat Enterprise Linux,
Red Hat Enterprise Linux Workstation,
Red Hat Enterprise Linux Desktop,
Red Hat x86_64 Linux Server,
Red Hat Linux HPC,
Red Hat Linux for SAP Applications,
Red Hat Linux Power Big Endian,
Red hat Software Collections,
Red Hat Developer Toolset,
Red Hat Linux Atomic Host.

Campus 2 contiene:

La distribuzione CAMPUS 1,

Red Hat Enterprise Virtualization (RHEV),

Red Hat OpenStack,

Red Hat Gluster Storage,

Red Hat Insight.

Quali i vantaggi per l'Ateneo o il Centro di ricerca?

Non ci sono limiti nel numero di sistemi Red Hat gestibili.

L'Ateneo può distribuire le sottoscrizioni Red Hat anche **a tutti gli studenti.**

Le distribuzioni acquisite tramite l'accordo CRUI - RED HAT possono essere utilizzate per fini **formativi e/o gestionali,** indifferentemente.

Per aderire al contratto CRUI - RED HAT

Iscriversi all'applicazione Plurimedia:
universamente.plurimedia.it

L'adesione al contratto CRUI - RED HAT da parte di un Ateneo può coincidere con **l'intero valore in FTE** dell'Ateneo stesso.

L'adesione al contratto CRUI - RED HAT può avvenire utilizzando la **soglia minima di 500 FTE**, indipendentemente dal valore in FTE complessivo dell'Ateneo.

Serverless Application

Cosa sono?
Perché sono importanti?
Quali sono i vantaggi che ne derivano dallo sviluppo serverless?
Cosa cambia rispetto allo sviluppo cloud nativo tradizionale?
Quando ha senso utilizzare le tecnologie Serverless?
Citeremo anche qualche caso d'uso concreto e gli ambienti cloud utilizzati.

Giuseppe Magnotta
Red Hat

Lorenzo Galotti
Plurimedia

Red Hat technologies that enable Serverless model

Giuseppe Magnotta
Associate Solution Architect - Red Hat

Previously software architect with *(some years of)* experience in the whole software development lifecycle, operations, containers and Kubernetes.

Global System Integration experience.

Open Source contributor.

Giuseppe Magnotta

Agenda

- Containers, Cloud-Native and Microservices
- Typical Kubernetes Deployment
- Introducing Serverless
- OpenShift Serverless
- Introducing Quarkus

Containers

Containers allows decoupling applications from the environment where they will be executed in.

They can be seen as black boxes that include everything needed to run an application: code, runtime, system tools, system libraries and settings.

At runtime, containers, will run completely isolated from each other and in a reproducible manner.

Deployment process is the same for all the containers regardless of the application.

Cloud-Native Development

Containers changed the approach to software development giving birth to the so called "Cloud-Native".

Cloud-native development is not focused on **WHERE** applications are deployed, but instead **HOW** applications are built, deployed and managed.

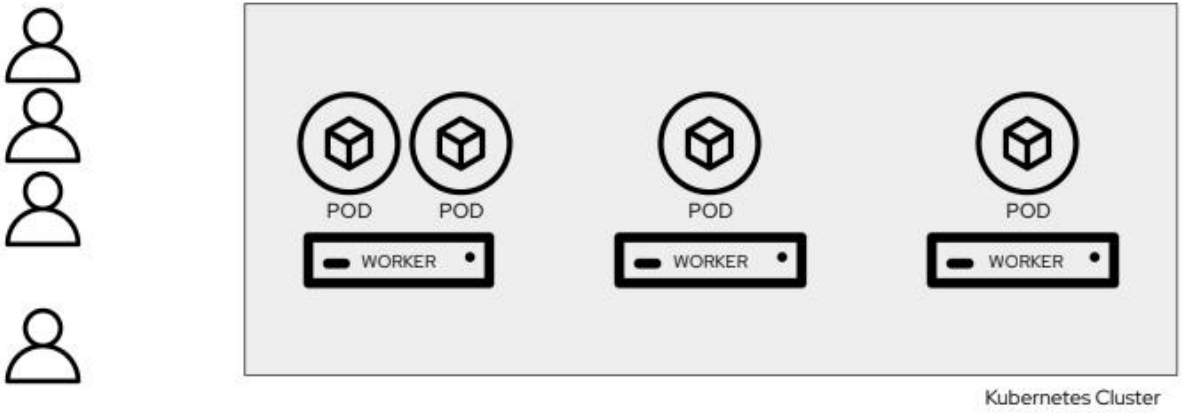
Cloud-native application development is based on four tenets:

- Service based architecture
- Api based communication
- Container based infrastructure
- DevOps process

CLOUD NATIVE
COMPUTING FOUNDATION

Cloud-native applications are **loosely coupled**, **resilient**, **manageable** and **observable**. They allow engineers to make high-impact changes frequently and predictably with minimal toil.

Typical Kubernetes deployment



What is Serverless?

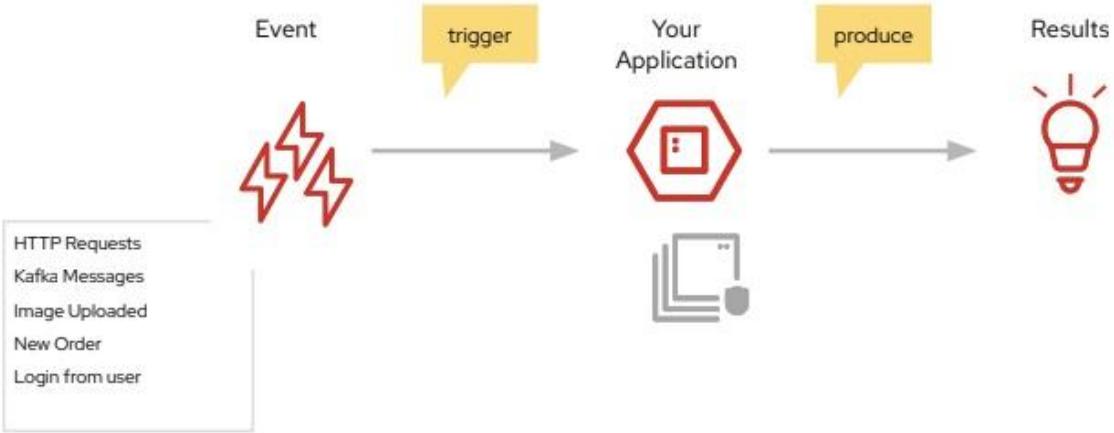
Serverless is a cloud-native deployment model for containers and functions that does resource allocation **on demand**.

There are still servers in serverless, but they are abstracted away from application development.

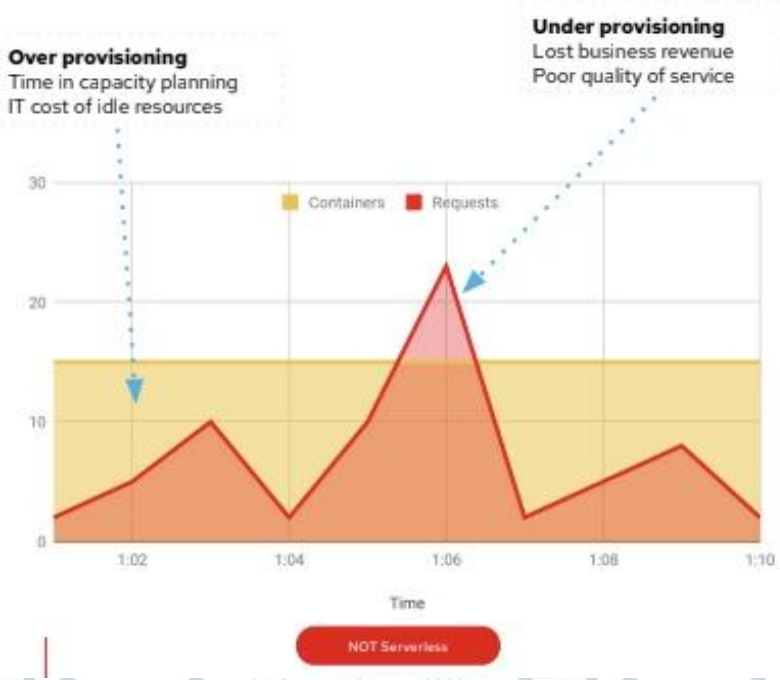
Serverless means that developers can build and run applications without having to manage servers.

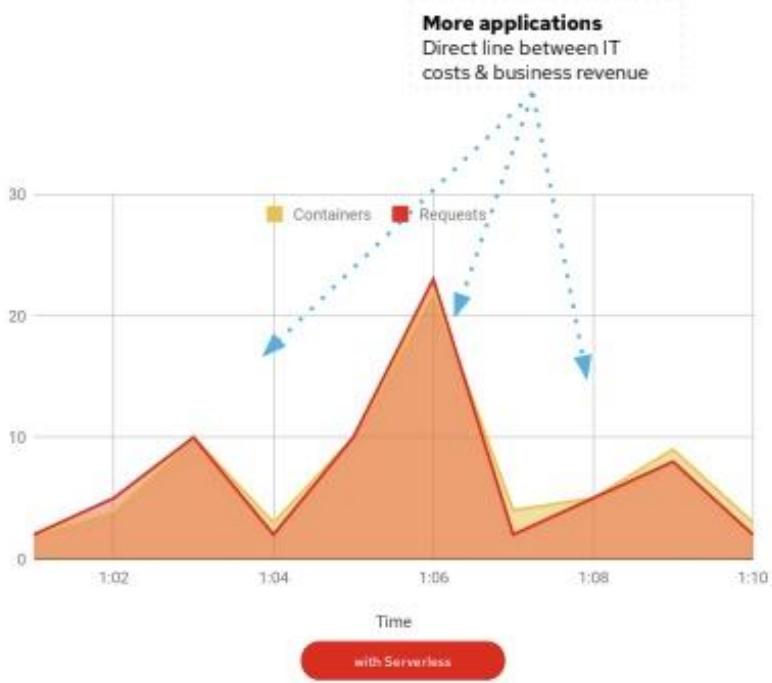
The **cloud provider** is **responsible** for managing both the cloud infrastructure and the scaling of applications.

The "Serverless Pattern"



Serverless Operational Benefits





When to go Serverless?

- Asynchronous, stateless applications that can be started instantaneously
- Application with unpredictable or bursty number of requests
- Applications that have to deal incoming data streams, chat bots, scheduled tasks
- Business process automation
- Integration across multiple systems

Serverless Pros vs Cons

Pros

- Serverless computing can increase developer productivity and reduce operational costs. By offloading the routine tasks of provisioning and managing servers, developers have more time to focus on their apps.
- Serverless helps enable DevOps adoption by reducing the need for developers to explicitly describe the infrastructure they need operations to provision for them.
- It's possible to streamline app development even further by incorporating entire components from third-party BaaS offerings.
- Operational costs are reduced in a serverless model because you can pay for cloud-based compute time as it's needed, as opposed to running and managing your own servers all the time.

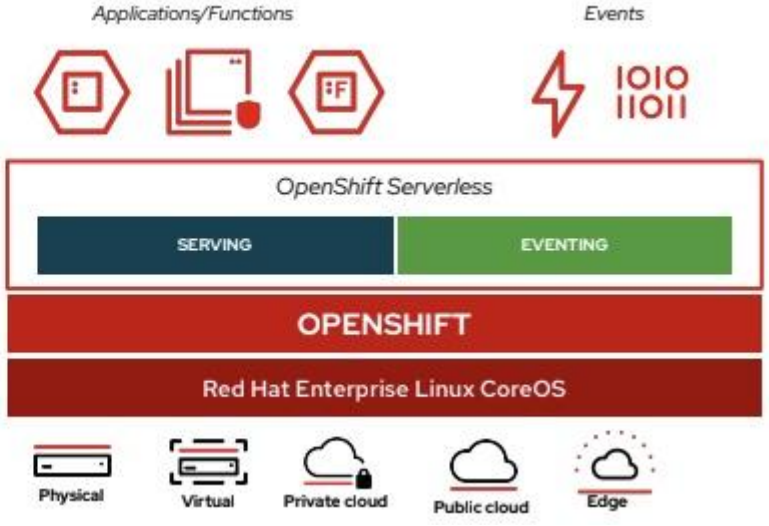
Cons

- Not running your own server or controlling your own server-side logic can have drawbacks.
- Cloud providers may have strict constraints on how their components can be interacted with, in turn affecting how flexible and customized your own systems can be. In the case of BaaS environments, developers may be beholden to services whose code is outside their control.
- Ceding control of these aspects of your IT stack also opens you up to vendor lock-in. Deciding to change providers will also likely come with the cost of upgrading your systems to adhere to the new vendor's specifications.

Openshift Serverless

Event-driven serverless containers and functions

- Deploy and run **serverless containers**
- Use any programming language or runtime
- Modernize existing applications to run serverless
- Powered by a rich ecosystem of event sources
- Manage serverless apps natively in Kubernetes
- Based on open source project **Knative**
- Run anywhere OpenShift runs



OpenShift Serverless

Key Features

Containers made easy

Simplified developer experience to deploy applications/code on serverless containers abstracting infrastructure & focusing on what matters.

Ready for the Hybrid Cloud

Truly portable serverless running anywhere OpenShift runs, that is on-premises or on any public cloud. Leverage data locality and SaaS when needed.

Immutable revisions

Deploy new features: performing canary, A/B or blue-green testing with gradual traffic rollout with no sweat and following best practices.

Any programming language

Use any programming language or runtime of choice. From Java, Python, Go and JavaScript to Quarkus, SpringBoot or Node.js.

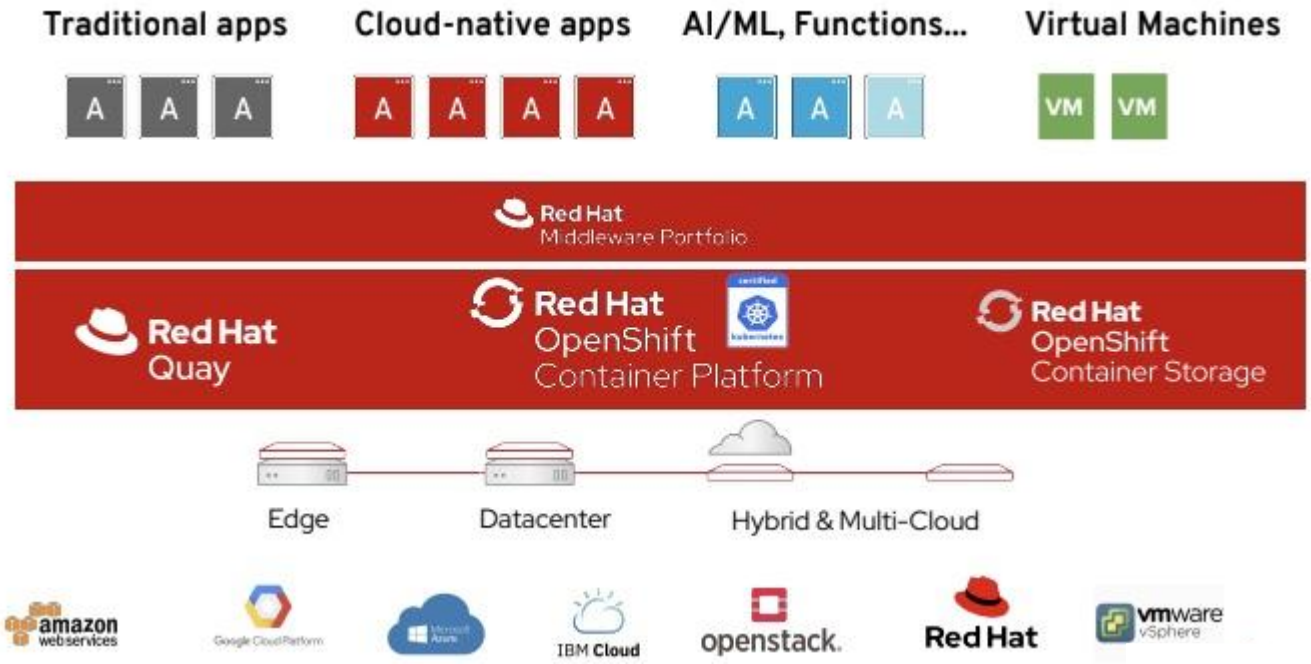
Automatic scaling

No need to configure number of replicas, or idling. Scale to zero when not in use, auto scale to thousands during peak, with built-in reliability and fault-tolerance.

Event Driven Architectures

Build loosely coupled & distributed apps connecting with a variety of built-in or third-party event sources or connectors powered by Operators.

With OpenShift you decide the best place to run your applications



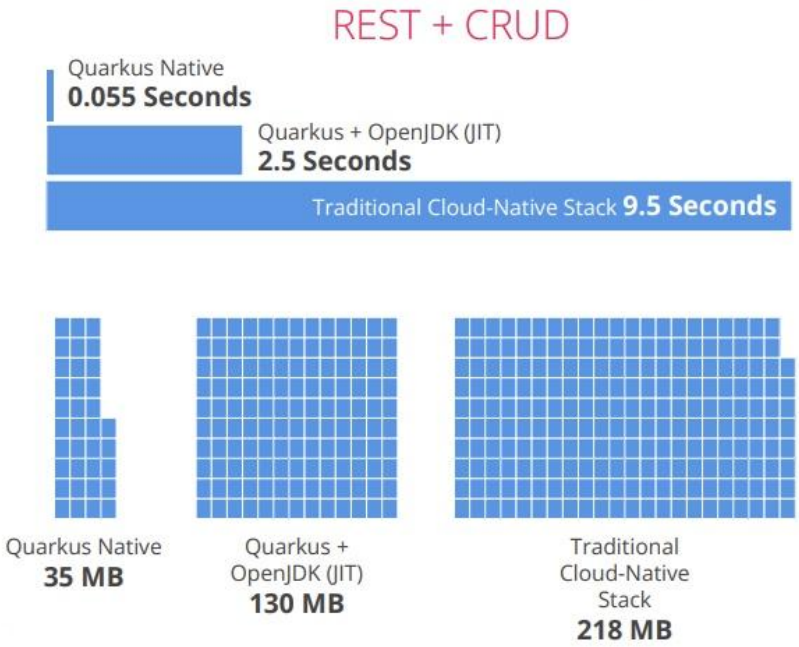
Java...
the Enterprise Workhorse

QUARKUS

Supersonic. Subatomic. Java.

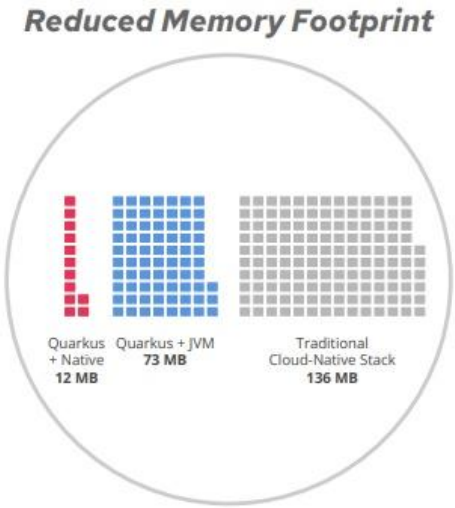
Experts from cloud-native Java OS projects

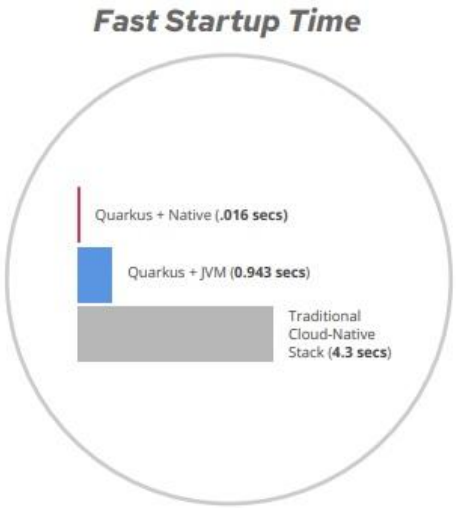
Building next-gen applications with Serverless and Quarkus

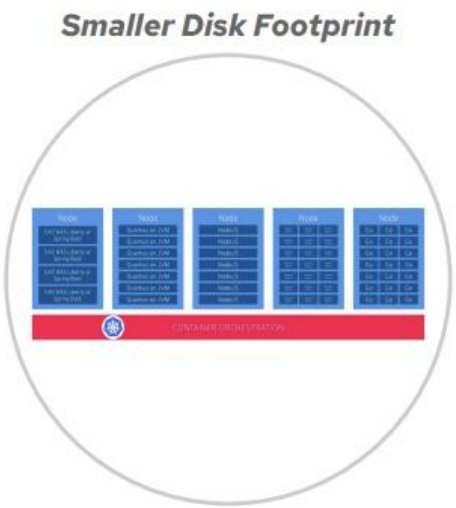


Cost Savings

"both jvm & native compilation modes"





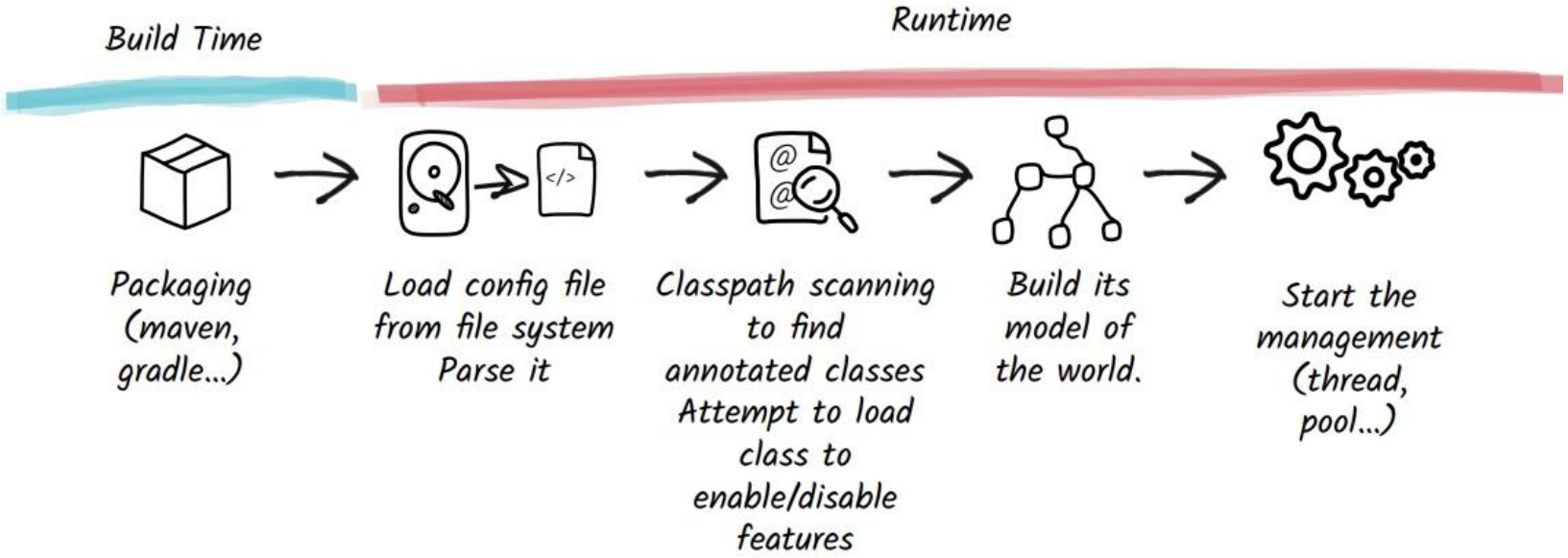


Why Quarkus?

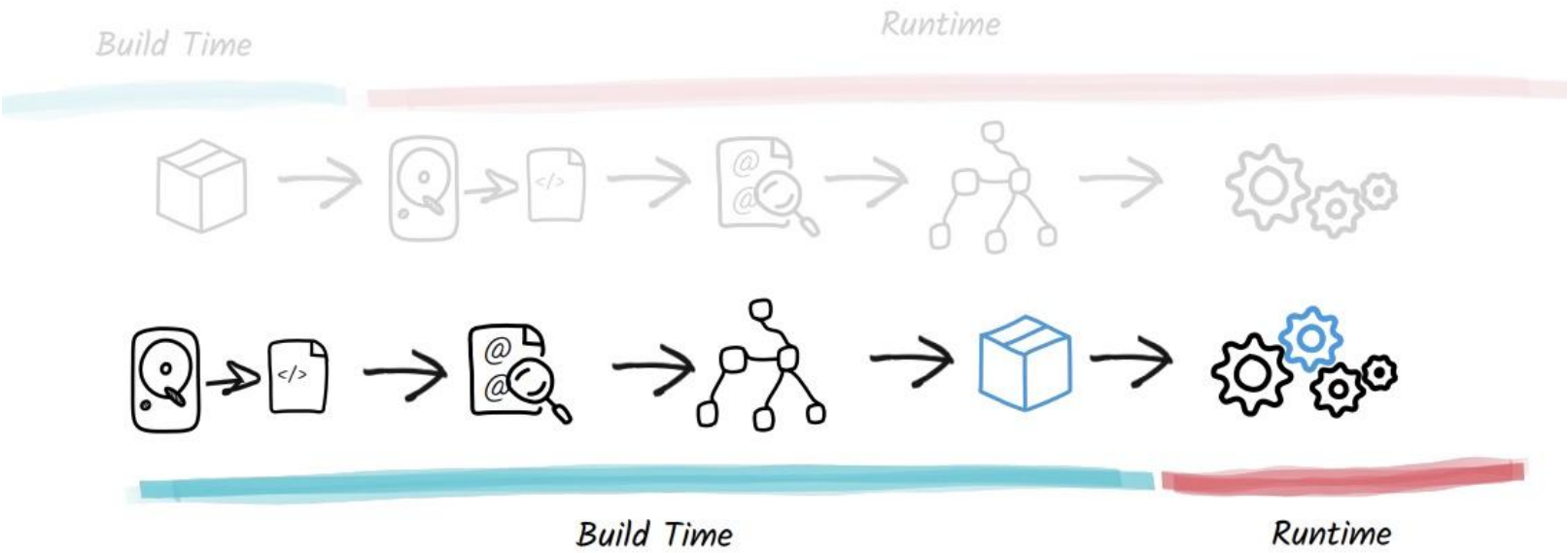
"Quarkus is a great fit if you are looking for…"

IMPROVED APP PERFORMANCE & COST	❑ Reduce memory consumption ❑ Increase startup speeds ❑ Reduce application size ❑ Looking for Spring or Java alternatives
DIGITAL TRANSFORMATION	❑ Modernize or optimize existing applications ❑ Move to a cloud-native or microservices architecture ❑ Need lightweight connectivity between services ❑ Next generation business automation
SERVERLESS	❑ Have seldom used but critical services ❑ Interested in serverless or FaaS (function as a service)

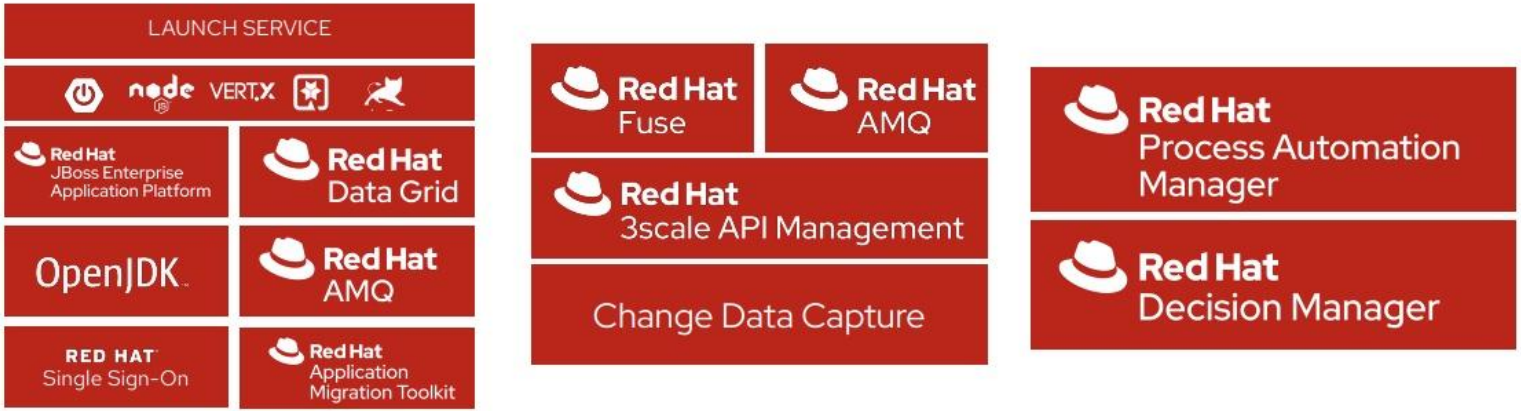
How Does a Framework Start?



The Quarkus Way



Red Hat Middleware



La nostra esperienza con le Serverless

Porting di una web app tradizionale verso un paradigma serverless

Lorenzo Galotti
Plurimedia

Il progetto e gli obiettivi

Il progetto da cui siamo partiti per questo esperimento è una semplice Single Page Application per la gestione dei trasporti, sviluppata per un nostro cliente.

Gli obiettivi erano diversi:

- Valutare lo sforzo, in termini di tempo e risorse, per un porting di piccole dimensioni verso un paradigma serverless
- Mantenere quanto più invariato possibile il codice della web app di partenza
- Sperimentare l'uso e l'integrazione degli strumenti serverless offerti da AWS (Amazon Web Services)

Generazione e setup iniziali con Amplify

Il primo passo è stato generare una nuova applicazione e configurarla secondo le specifiche di AWS Amplify: il servizio che si sarebbe poi occupato del build e del deploy della nostra serverless application

Dopo aver generato lo scheletro dell'applicazione utilizzando il framework prescelto (nel nostro caso Quasar per Vue3) abbiamo installato Amplify come pacchetto aggiuntivo del progetto:

```
$ quasar create serverless_test
$ npm install -g @aws-amplify/cli
```

Generazione e setup iniziali con Amplify

A questo punto, dopo aver seguito il wizard di Amplify, dobbiamo aggiungere un file di configurazione YAML per specificare i comandi di build dell'applicazione. Senza queste istruzioni Amplify non è in grado di buildare e tantomeno deployare l'applicazione.

In applicazioni avanzate e con diversi moduli queste istruzioni possono diventare piuttosto complesse, ma nel nostro caso ce la siamo cavata con sole 24 righe di codice.

amplify.yml > # version

```yaml
version: 1
applications:
  - appRoot: .
    backend:
      phases:
        build:
          commands:
            - amplifyPush --simple
    frontend:
      phases:
        preBuild:
          commands:
            - nvm install 14.17.0
        build:
          commands:
            - npm install
            - npx quasar build
      artifacts:
        baseDirectory: dist/spa
        files:
          - '**/*'
      cache:
        paths:
          - node_modules
```

Autenticazione e login

Per la parte di autenticazione e login abbiamo voluto utilizzare Cognito, il servizio AWS dedicato.

La configurazione e l'integrazione di Cognito con la nostra serverless app sono stati relativamente semplici, ma la mancanza di supporto nativo per il framework di sviluppo da noi prescelto ci ha costretti a modificare alcuni file di configurazione di Quasar per rendere compatibile il processo di build dell'applicazione con i moduli di Amplify/UI.

Modifiche di questo tipo e di questa natura richiedono conoscenze estremamente avanzate, non solo di programmazione, ma anche riguardo al funzionamento "dietro le quinte" dei potenti framework di frontend development a cui ci siamo abituati.

Amplify e' nativamente compatibile solo con i tre grandi framework diffusi oggi: Vue, Angular, React. Purtroppo, senza interventi specifici, l'utilizzo di framework diversi, o di livello più alto ("*framework di framework*") è impossibile.

API

- Le API vengono scritte in maniera sostanzialmente identica a una web app tradizionale
- Alcune task di routine possono essere ridotte o evitate
- Amplify genera una “sub-applicazione” all’interno del progetto che richiede il suo *package.json* dedicato dove scrivere le API

```
app.get('/api/user/listini/list', async (req, res) => {
  const data = await sql`
    select listino_id, trasportatore, fine_validita, triangolazione, data
      from listini
     where not deleted
     order by trasportatore
  `
  res.status(200).json(data)
})
```

```
router.get('/list', isAuth, async (req, res) => {
  const data = await sql`
    select listino_id, trasportatore, fine_validita, triangolazione, data
      from listini
     where not deleted
     order by trasportatore
  `
  res.status(200).json(data)
})
```

Confronto tra API “tradizionale” (dx) e “serverless” (sx)

Conclusioni

L'intero porting ha richiesto circa 10 giorni lavorativi, compreso lo studio dei servizi serverless AWS e la loro implementazione

- I vantaggi sono principalmente a livello operations *(provisioning e gestione dei server, costi allinenati all'uso reale)*
- La complessità per gli sviluppatori aumenta, soprattutto se non utilizzano strumenti o framework previsti dal vendor
- L'assenza di controllo delle logiche server-side rende difficile implementare le richieste più specifiche dei clienti *(es: extra security layers)*
- Si riduce il carico di lavoro per i sistemisti, in quanto il processo di provisioning, dimensionamento, e gestione dei server è del tutto demandato al vendor prescelto

Domande e risposte

“

Quali sono le skill più importanti da acquisire per orientarsi in questo nuovo mondo?

“

In quali casi bisognerebbe andare verso il cloud?

“

Quali sono i linguaggi di programmazione più adatti in questo nuovo mondo? Ha ancora senso scrivere applicazioni in java?

“

Quando si hanno applicazioni monolitiche che fanno uso di database relazionali molto grandi, qual è l’approccio per smembrare il monolite e portarlo su cloud ibrido?

“

Quando si hanno applicazioni monolitiche che fanno uso di database relazionali molto grandi, qual è l’approccio per smembrare il monolite e portarlo su cloud ibrido?

“

Visti gli svantaggi e i vantaggi che hai descritto, cosa consiglieresti a chi ti chiede se vale la pena di sviluppare qualcosa in serverless?

Grazie!

LINK UTILI

universamente
Il gruppo in linkedin
www.universamente.it
il portale della community
universamente.plurimedia.it
l'App di community

Giuseppe Magnotta
Email: gmagnott@redhat.com
Lorenzo Galotti
Email: lorenzo.galotti@plurimedia.it

UniversaMente

plurimedia Red Hat CRUI Conferenza dei Rettori delle Università Italiane