**What is Serverless - 2022**

* Serverless is a [cloud-native](https://www.redhat.com/en/topics/cloud-native-apps) development model that allows developers to build and run applications without managing servers.
* A [cloud provider](https://www.redhat.com/en/topics/cloud-computing/what-are-cloud-providers) handles the routine work of [provisioning](https://www.redhat.com/en/topics/automation/what-is-provisioning), maintaining, and scaling the server [infrastructure](https://www.redhat.com/en/topics/cloud-computing/what-is-it-infrastructure). There are still servers in serverless, but they are abstracted away from app development.
* Developers can simply package their code in [containers](https://www.redhat.com/en/topics/containers) for deployment.
* Once deployed, serverless apps respond to demand and [automatically](https://www.redhat.com/en/topics/automation) scale up and down as needed. As a result, when a serverless function is sitting idle, it doesn’t cost anything.
* Serverless differs from other [cloud computing](https://www.redhat.com/en/topics/cloud) models in that the cloud provider is responsible for managing both the [cloud infrastructure](https://www.redhat.com/en/topics/cloud-computing/what-is-cloud-infrastructure) and the scaling of apps. Serverless apps are deployed in containers that automatically launch on demand when called.
* Under a standard [Infrastructure-as-a-Service (IaaS)](https://www.redhat.com/en/topics/cloud-computing/what-is-iaas) cloud computing model, users prepurchase units of capacity, meaning you pay a public cloud provider for always-on server components to run your apps. It’s the user’s responsibility to scale up server capacity during times of high demand and to scale down when that capacity is no longer needed. The cloud infrastructure necessary to run an app is active even when the app isn’t being used.
* With serverless architecture, by contrast, **apps are launched only as needed**. When an event triggers app code to run, the public cloud provider dynamically allocates resources for that code. The user stops paying when the code finishes executing. In addition to the cost and efficiency benefits, serverless frees developers from routine and menial tasks associated with app scaling and server provisioning.
* Serverless computing offerings typically fall into two groups, **Backend-as-a-Service (BaaS)** and [**Function-as-a-Service (FaaS)**](https://www.redhat.com/en/topics/cloud-native-apps/what-is-faas).

**What is BaaS (Backend-as-a-Service)**

* BaaS gives developers access to a variety of third-party services and apps. For instance, a cloud-provider may offer **authentication services**, **extra encryption**, **cloud-accessible databases**, and high-fidelity usage data. With BaaS, serverless functions are usually called through [application programming interfaces (APIs)](https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces).

**What is FaaS (Function-as-a-Service**)

* More commonly, when developers refer to serverless, they’re talking about a FaaS model. Under FaaS, **developers still write custom server-side logic**, but it’s run in containers fully managed by a cloud services provider.
* Function-as-a-Service (FaaS) is an event-driven computing execution model where developers write logic that is deployed in containers fully managed by a platform, then executed on demand.
* In contrast to BaaS, FaaS affords a greater degree of control to the developers, who create custom apps rather than relying on a library of prewritten services.

## What are some serverless use cases?

* Serverless architecture is ideal for asynchronous, stateless apps that can be started instantaneously. Likewise, serverless is a good fit for use cases that see infrequent, unpredictable surges in demand.
* Think of a task like batch processing of incoming image files, which might run infrequently but also must be ready when a large batch of images arrives all at once.

**What are the pros and cons of serverless computing?**

**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](https://www.redhat.com/en/topics/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.

**Serverless databases**

Several serverless [databases](https://en.wikipedia.org/wiki/Distributed_data_store) have emerged in the last few years. These systems extend the serverless execution model to the [RDBMS](https://en.wikipedia.org/wiki/Relational_database_management_system), eliminating the need to provision or scale [virtualized](https://en.wikipedia.org/wiki/Hardware_virtualization) or physical database hardware.

[Nutanix](https://en.wikipedia.org/wiki/Nutanix) offers a solution named Era which turns an existing as Oracle, MariaDB, Postgress and MS SQL server into serverless service.

[Amazon Aurora](https://en.wikipedia.org/wiki/Amazon_Aurora) offers a serverless version of its databases, based on MySQL and PostgreSQL, providing on-demand, auto-scaling configurations.

[Azure Data Lake](https://en.wikipedia.org/wiki/Azure_Data_Lake) is a highly scalable data storage and analytics service.

**Spring Cloud Function Example – 2022**

**pom.xml**

<modelVersion>4.0.0</modelVersion>

Graphical user interface, application

Description automatically generated

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.6.12</version>

<relativePath />

</parent>

<properties>

<java.version>11</java.version>

<spring-cloud.version>2021.0.4</spring-cloud.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-function-context</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-function-web</artifactId>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>${spring-cloud.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

**Entity Layer**

@Data

**public** **class** Employee {

**private** **int** id;

**private** String name;

}

**Spring Boot Main Application**

@SpringBootApplication

**public** **class** MyserverlessApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(MyserverlessApplication.**class**, args);

}

// Here also you can define all @Beans as you have done in CoreFunctions class

}

**Basic Core Functions**

@Component

**public** **class** CoreFunctions {

@Bean //GET http://localhost:8080/supply

**public** Function<String, String> function() {

**return** input -> input;

}

@Bean

**public** Consumer<String> consume() {

**return** input -> System.***out***.println("Input: " + input);

}

@Bean

**public** Supplier<String> supply() {

**return** () -> "Hello Guys";

}

}

**Set of functions for Employee**

@Component

**public** **class** EmployeeFunction {

@Autowired

**private** EmployeeService empService;

@Bean // POST http://localhost:8080/createEmp , Http Response code: 202 - Accepted

**public** Consumer<Employee> createEmp() {

**return** emp -> empService.createEmployee(emp);

}

@Bean // GET http://localhost:8080/getNameById/23

**public** Function<Integer, String> getNameById() {

**return** idValue -> empService.getEmpNameById(idValue);

}

@Bean // GET http://localhost:8080/getEmployeeById/23

**public** Function<Integer, Employee> getEmployeeById() {

**return** idValue -> empService.getEmployeeById(idValue);

}

@Bean // GET http://localhost:8080/getTotalEmp

**public** Supplier<String> getTotalEmp() {

**return** () -> "Total: 50";

}

@Bean // POST http://localhost:8080/updateEmpName

**public** Function<Employee, String> updateEmpName() {

**return** emp -> empService.updateEmp(emp);

}

}

**Service layer**

@Service

**public** **class** EmployeeService {

**public** String createEmployee(Employee emp) {

System.***out***.println("Emp: " + emp);

**return** "Employee created successfully ...";

}

**public** String getEmpNameById(**int** id) {

**return** "John Abraham";

}

**public** Employee getEmployeeById(**int** id) {

Employee emp = **new** Employee();

emp.setId(id);

emp.setName("Jyotika Sadana");

**return** emp;

}

**public** String updateEmp(Employee emp) {

Employee emp1 = **new** Employee();

emp1.setId(emp.getId());

emp1.setName("Viday Balan");

return "Name changed successfully ...";

}

}

| **Method** | **Path** | **Request** | **Response** | **Status** |
| --- | --- | --- | --- | --- |
| GET | /{supplier} | - | Items from the named supplier | 200 OK |
| POST | /{consumer} | JSON object or text | Mirrors input and pushes request body into consumer | 202 Accepted |
| POST | /{consumer} | JSON array or text with new lines | Mirrors input and pushes body into consumer one by one | 202 Accepted |
| POST | /{function} | JSON object or text | The result of applying the named function | 200 OK |
| POST | /{function} | JSON array or text with new lines | The result of applying the named function | 200 OK |
| GET | /{function}/{item} | - | Convert the item into an object and return the result of applying the function | 200 OK |

Note: There is no PUT, DELETE Mapping in case of Spring cloud function for Serverless technologies.