

# Serverless Architecture

*Software Architecture*

Richard Thomas

May 6, 2024

### *Oxymoron 1.* Serverless

Logic running on someone else's server.

Developers can focus on logic, not infrastructure to deliver it.

*Definition 1.* Backend as a Service (BaaS)

Cloud-hosted applications or services that deliver functionality used by an application front-end.

- Front-end may be a SPA or mobile app.
- Back-end provides sophisticated functionality (e.g. database, machine learning, location services, authentication, ...).
- Front-end ties back-end services together to deliver the application's functionality.

# BaaS Iceberg *[Brunko, 2019]*



## BaaS Example



- Example of simple system with back-end functionality delivered entirely via BaaS.
- Feature-rich front-ends coordinate behaviour delivered by BaaS.
- Consequence: Front-ends are tightly coupled to BaaS.
- Consequence: Front-ends are have both UI and functional behaviour logic.
- Front-end could have a layered design, though many SPAs don't.

*Definition 2.* Functions as a Service (FaaS)

Application logic that is triggered by an event and runs in a *transient*, *stateless* compute node.

- Node may only exist for duration of function call.
- Server infrastructure (e.g. type of node, lifespan, scaling, ...) are managed by hosting provider.
- e.g. AWS Lambda, Google App Engine, Azure Automation, ....

# FaaS Iceberg *[Brunko, 2019]*



# FaaS Example



- Example of simple system with back-end functionality delivered entirely by FaaS.
- Feature-rich front-ends coordinate behaviour delivered by FaaS.
- Front-ends invoke functions via an API.
- API Gateway provides some separation between front-end and functions.
- May allow a bit more separation between UI and logic.

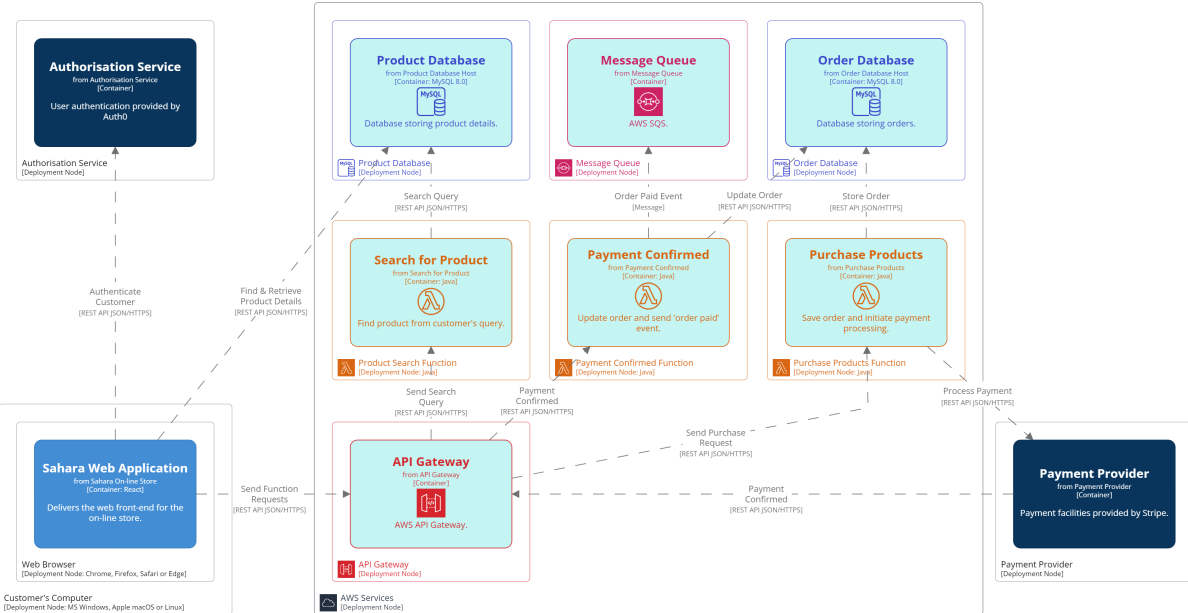


### *Definition 3. Serverless Architecture*

Software system delivering functionality through BaaS or FaaS.

- Many people focus on FaaS when considering Serverless.
- Some simple Single Page Web Apps (SPA) coordinate services.
- Front-end ties back-end services together to deliver the application's functionality.

## Sahara Browse & Order



- Sahara eCommerce example as a serverless app.
- Only browse, search and purchase are shown.
- Point out that it uses both BaaS & FaaS.
- Shopping cart is implemented within the web and mobile app for this architecture.
- Order Scenario 1: Customer checks out their shopping cart in the web or mobile app.
- Order Scenario 2: App calls Purchase Products function via API Gateway.
- Order Scenario 3: Purchase Products stores order in DB and sends a payment request to Payment Provider.
- Order Scenario 4: We provide Payment Provider with API end point to call to report payment result.
- Order Scenario 5: Payment success causes Payment

The diagram illustrates the architecture of an e-commerce system using AWS services. The components and their interactions are as follows:

- API Gateway** (from API Gateway [Container]) and **API Gateway** (Deployment Node) are at the top left. The API Gateway (Deployment Node) sends a **Query Order Status** (REST API JSON/HTTPS) to the **Order Status** service.
- Order Status** (from Order Status [Container: Java]) and **Order Status Query Function** (Deployment Node: Java) are in the middle left. The **Order Status Query Function** sends a **Query Order Status** (REST API JSON/HTTPS) to the **Order Database**.
- Order Database** (from Order Database Host [Container: MySQL 8.0]) and **Order Database** (Deployment Node) are at the bottom left. The **Order Database** sends an **Update Order Details** (REST API JSON/HTTPS) to the **Order Status** service.
- Order Shipped** (from Order Shipped [Container: Java]) and **Order Shipped Function** (Deployment Node: Java) are in the middle. The **Order Shipped Function** sends an **Order Shipped** (REST API JSON/HTTPS) to the **API Gateway** and an **Order Shipped Event** (Message) to the **Message Queue**.
- Email Service** (from Email Service [Container]) and **Email Service** (Deployment Node) are at the top right. The **Order Shipped Function** sends an **Order Shipped** (REST API JSON/HTTPS) to the **Email Service**.
- Lambda Service** (from Lambda Service [Container]) and **Lambda Service** (Deployment Node) are in the middle right. The **Order Shipped Event** (Message) is sent to the **Message Queue**, which then sends a **Poll for Messages** to the **Lambda Service**.
- Fulfillment Mobile Device** (from Sahara On-line Store [Container: React Native]) and **Fulfillment Mobile Device** (Deployment Node: Android or iOS) are at the far right. The device sends a **Send Pick List** (HTTPS) to the **Fulfill Order** service.
- Fulfill Order** (from Fulfill Order [Container: Java]) and **Fulfill Order Function** (Deployment Node: Java) are at the bottom right. The **Fulfillment Mobile Device** sends a **Send Pick List** (HTTPS) to the **Fulfill Order** service, which then sends a **Ship Order Message Batch** to the **Message Queue**.
- Message Queue** (from Message Queue [Container]) and **Message Queue** (Deployment Node) are in the bottom middle. The **Message Queue** sends an **Order Picking Event** (Message) to the **Fulfill Order** service.
- AWS Services** (Deployment Node) are at the bottom left, connected to the **Order Database** and **Message Queue**.

- Sahara eCommerce example as a serverless app.
- Only fulfilment functions are shown.
- Shows Lambda Service polling Queue, demonstrating how Lambda Functions are invoked via events in a message queue.
- Fulfilment Scenario 1: Lambda Service monitors Queue for 'ship order' messages.
- Fulfilment Scenario 2: Lambda Service batches groups of 'ship order' messages and sends them to Fulfill Order function.
- Fulfilment Scenario 3: Fulfil Order gets order details from DB and sends pick list to Fulfilment App.
- Fulfilment Scenario 4: When order is shipped, Fulfilment App calls Order Shipped function via API Gateway.

## Serverless Benefits

- Automatic scaling
  - Multiple instances of function

## Serverless Benefits

- Automatic scaling
  - Multiple instances of function
- Reduced cost for dynamic loads
  - No server idle time

## Serverless Benefits

- Automatic scaling
  - Multiple instances of function
- Reduced cost for dynamic loads
  - No server idle time
- Reduced server management

## Serverless Benefits

- Automatic scaling
  - Multiple instances of function
- Reduced cost for dynamic loads
  - No server idle time
- Reduced server management
- Easier to run closer to client
  - Launch in same zone as client

## BaaS Tradeoffs

- Front-end accesses database directly
  - Front-end needs to sanitise inputs
  - Easy to spoof messages from front-end
    - Hope DB provider is secure

Spoofing messages is an issue for all BaaS services.



## BaaS Tradeoffs

- Front-end accesses database directly
  - Front-end needs to sanitise inputs
  - Easy to spoof messages from front-end
    - Hope DB provider is secure
- Application logic is in front-end
  - Less modularisation
  - Duplication of logic with multiple front-ends
    - Web, mobile, ...
- Modern expectations are that almost all systems will have multiple front-ends.
- Duplication of front-end logic is a smaller, but still partial, concern for FaaS.

## BaaS Tradeoffs

- Front-end accesses database directly
  - Front-end needs to sanitise inputs
  - Easy to spoof messages from front-end
    - Hope DB provider is secure
- Application logic is in front-end
  - Less modularisation
  - Duplication of logic with multiple front-ends
    - Web, mobile, ...
- No control over server optimisation

## FaaS Tradeoffs

- No server state
  - All state needs to be saved (e.g. Redis, S3, ...)
    - Not just persistent state
- Server running function can be killed when function is not running.
- Can occasionally send messages to functions to keep them alive.

## FaaS Tradeoffs

- No server state
  - All state needs to be saved (e.g. Redis, S3, ...)
    - Not just persistent state
- Execution duration
  - Can't be long running process
    - AWS Lambda – up to 15 minutes

## FaaS Tradeoffs

- No server state
  - All state needs to be saved (e.g. Redis, S3, ...)
    - Not just persistent state
- Execution duration
  - Can't be long running process
    - AWS Lambda – up to 15 minutes
- Startup latency
  - Functions take time to start
    - Some languages worse than others (e.g. Java)

Java has concurrency benefits over other languages.

## FaaS Tradeoffs

- No server state
  - All state needs to be saved (e.g. Redis, S3, ...)
    - Not just persistent state
- Execution duration
  - Can't be long running process
    - AWS Lambda – up to 15 minutes
- Startup latency
  - Functions take time to start
    - Some languages worse than others (e.g. Java)
- Proliferation of functions
  - Loss of encapsulation

*Question*

When is serverless appropriate?

*Question*

When is serverless appropriate?

*Answer*

- Rich client apps with common backend
  - BaaS



### *Question*

When is serverless appropriate?

### *Answer*

- Rich client apps with common backend
  - BaaS
- High latency processing
  - Within function duration constraints

### *Question*

When is serverless appropriate?

### *Answer*

- Rich client apps with common backend
  - BaaS
- High latency processing
  - Within function duration constraints
- Apps with variable load
  - Take advantage of auto-scaling

*Question*

When is serverless *not* appropriate?

*Question*

When is serverless *not* appropriate?

*Answer*

- Quick response required
  - Can't wait for FaaS to start

*Question*

When is serverless *not* appropriate?

*Answer*

- Quick response required
  - Can't wait for FaaS to start
- Compute intensive processing

### *Question*

When is serverless *not* appropriate?

### *Answer*

- Quick response required
  - Can't wait for FaaS to start
- Compute intensive processing
- Apps with steady load
  - Server-based approaches are cheaper

*Self-Study Exercise*

- Redesign your scalability assignment to be serverless.
  - What parts of your design would benefit from being serverless?
- Implement your revised design.

## Pros & Cons

Extensibility



Reliability



Interoperability



Scalability



Deployability



Modularity



Testability



Maintainability



Security



Simplicity



- Modularity: Deployed functions are naturally modular.
- Modularity: Higher-level abstractions to group deployed functions is difficult.
- Testability: Unit testing FaaS functions is easy.
- Testability: Integration testing is harder.
- Maintainability: Backend modularity and independence should facilitate its maintenance.
- Maintainability: Frontend contains UI and application logic.
- Security BaaS: Front-end access database directly. No server-side protection of db.
- Security FaaS: Every function needs its own security policy (e.g. IAM), which is easy to get wrong.



## References

- [Brunko, 2019] Brunko, P. (2019).  
Serverless architecture: When to use this approach and what benefits it gives.  
<https://apiko.com/blog/serverless-architecture-benefits/>.