

Serverless Architecture

CSSE6400

Richard Thomas

May 23, 2022

Oxymoron 1. Serverless

Logic running on someone else's server.

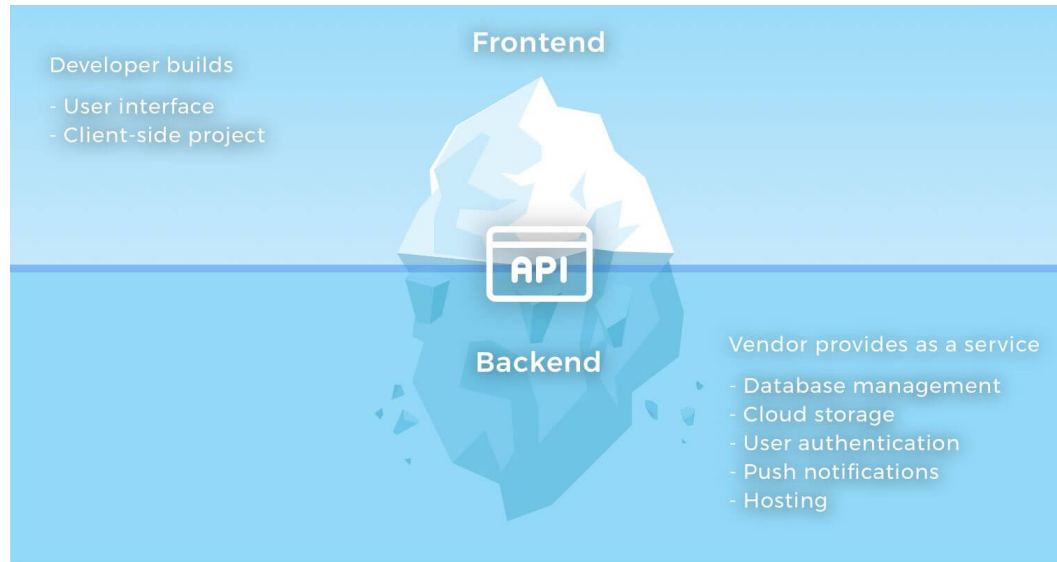
Developers can focus on logic, not infrastructure to deliver it.

Definition 1. Backend as a Service (BaaS)

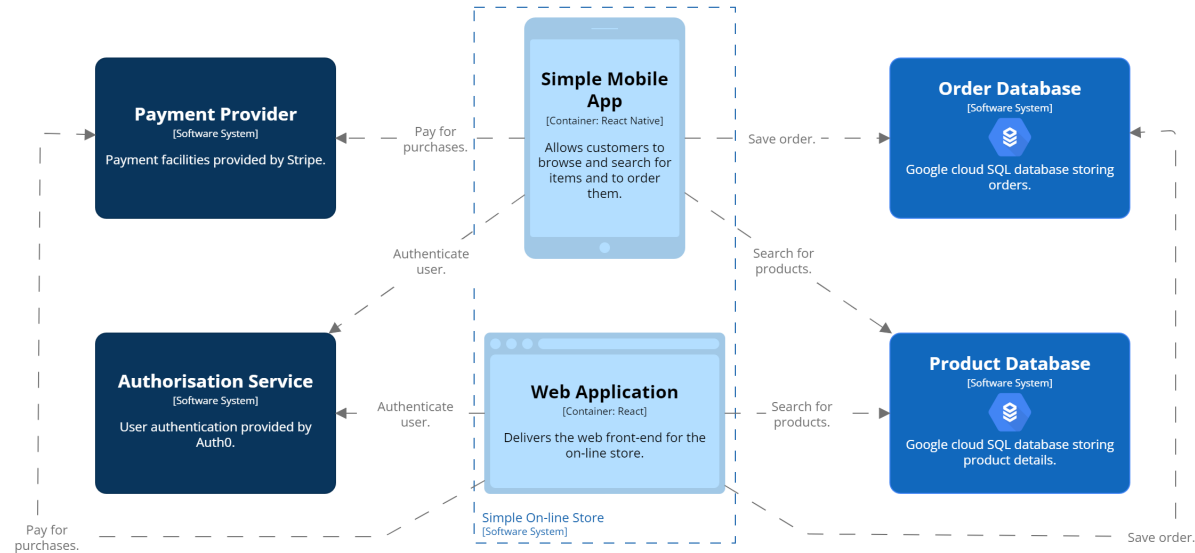
Cloud-hosted applications or services that deliver functionality used by an application front-end.

- 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 [1]



BaaS Example



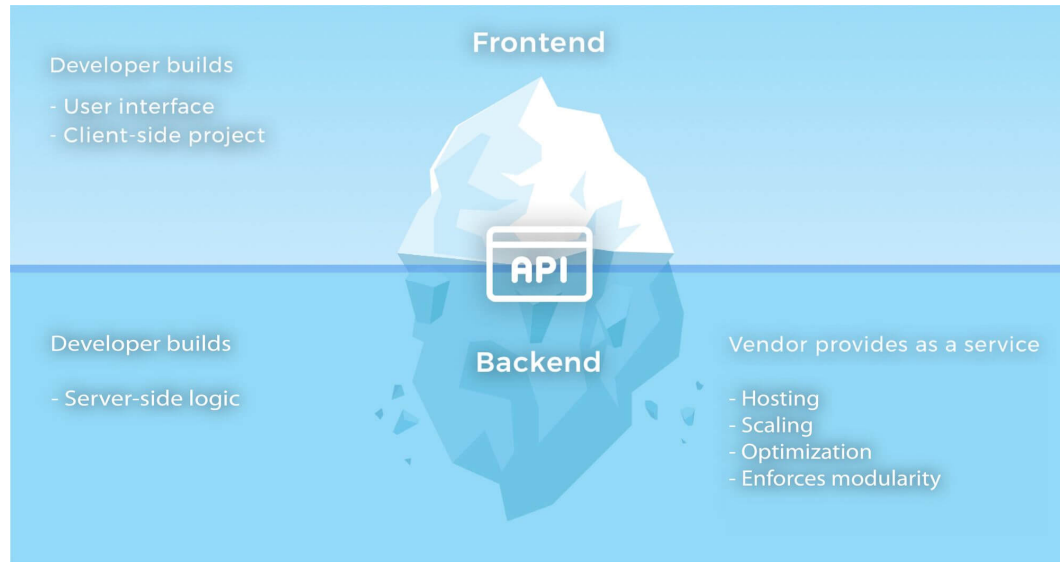
- Example of simple system with back-end functionality delivered entirely via BaaS.
- Feature-rich front-ends coordinate behaviour delivered by BaaS.
- Consequence: Front-ends are tightly coupled to BaaS.
- Consequence: Front-ends are have both UI and functional behaviour logic.
- Front-end could have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

Definition 2. Functions as a Service (FaaS)

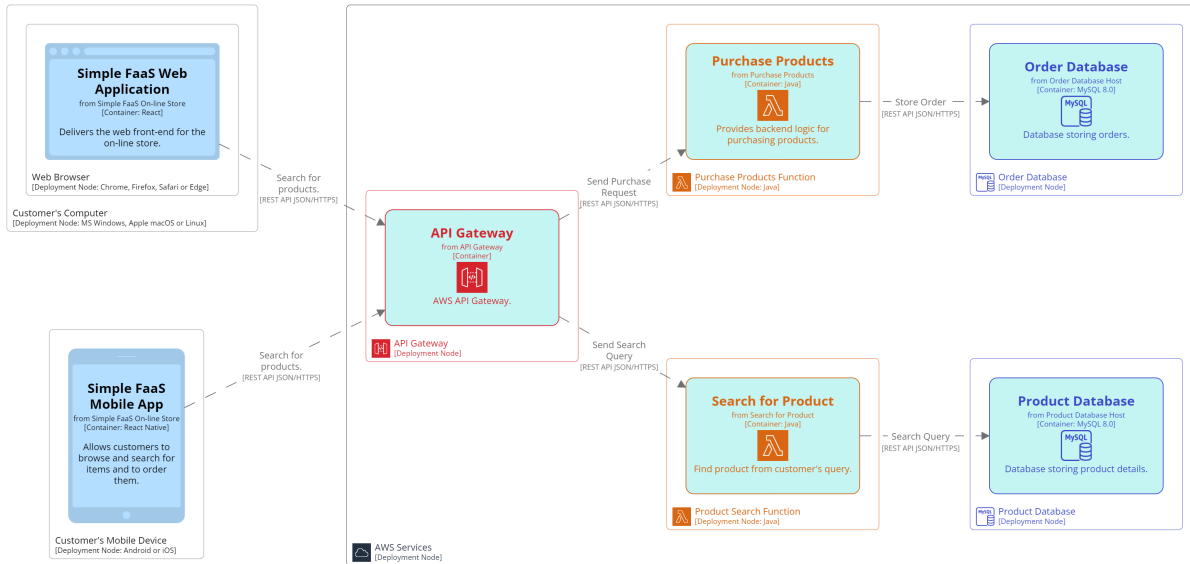
Application logic that is triggered by an event and runs in a transient, stateless compute node.

- 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 [1]



FaaS Example



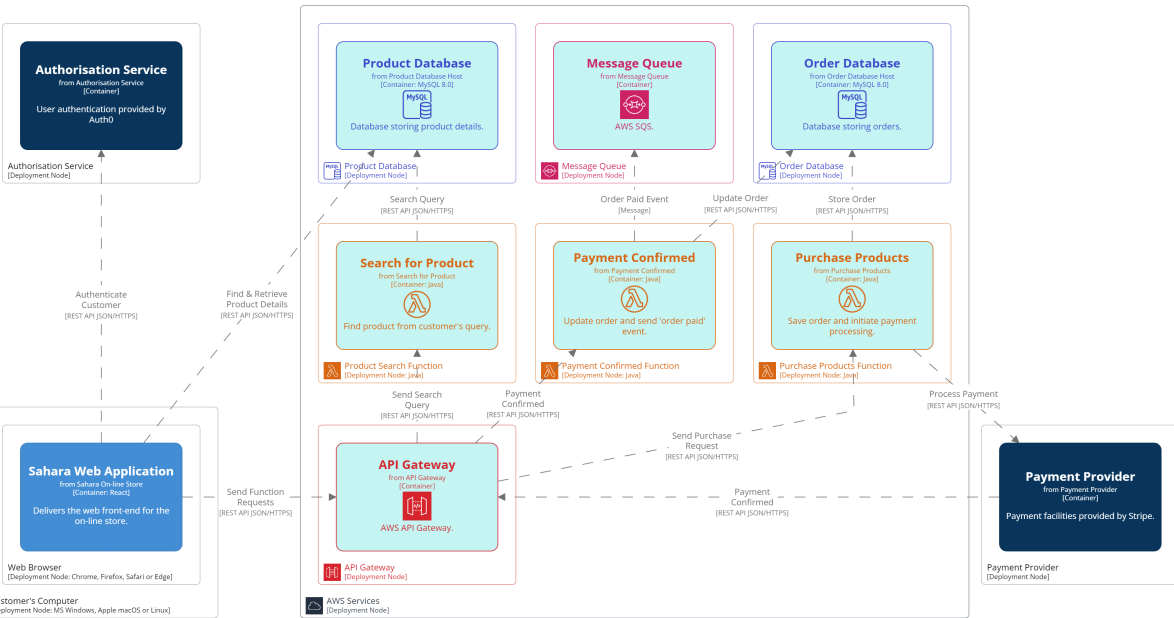
- Example of simple system with back-end functionality delivered entirely by FaaS.
- Feature-rich front-ends coordinate behaviour delivered by FaaS.
- Front-ends invoke functions via an API.
- API Gateway provides some separation between front-end and functions.
- May allow a bit more separation between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

```

graph TD
    subgraph AWS_Services [AWS Services]
        API_GW[API Gateway  
from API Gateway  
Container  
AWS API Gateway.  
API Gateway  
Deployment Node]
        Order_Status[Order Status  
from Order Status  
Container: Java  
Respond to customer query with  
current status of order.  
Order Status Query Function  
Deployment Node: Java]
        Order_Shipped[Order Shipped  
from Order Shipped  
Container: Java  
Send email to customer with  
message that order has shipped.  
Order Shipped Function  
Deployment Node: Java]
        Email_Service[Email Service  
from Email Service  
Container  
AWS SES.  
Email Service  
Deployment Node]
        Lambda_Service[Lambda Service  
from Lambda Service  
Container  
AWS Lambda Service polls Message  
Queue and batches messages to  
send to Lambda Functions.  
Lambda Service  
Deployment Node]
        Order_DB[Order Database  
from Order Database Host  
Container: MySQL 8.0  
Database storing orders.  
Order Database  
Deployment Node]
        Message_Queue[Message Queue  
from Message Queue  
Container  
AWS SQS.  
Message Queue  
Deployment Node]
        Fulfill_Order[Fulfill Order  
from Fulfill Order  
Container: Java  
Send pick list for order to warehouse.  
Fulfill Order Function  
Deployment Node: Java]
    end

    subgraph Mobile_App [Fulfillment App]
        Fulfillment_App[Fulfillment App  
from Sahara On-line Store  
Container: React Native  
Coordinates fulfillment  
of orders.  
Fulfillment Mobile Device  
Deployment Node: Android or iOS]
    end

    API_GW -- "Query Order Status" --> Order_Status
    Order_Status -- "Update Order Details [REST API JSON/HTTPS]" --> Order_DB
    Order_Status -- "Order Shipped [REST API JSON/HTTPS]" --> Order_Shipped
    Order_Shipped -- "Send Order Shipped Message" --> Email_Service
    Order_Shipped -- "Send Order Shipped Event [Message]" --> Message_Queue
    Message_Queue -- "Poll for Messages" --> Lambda_Service
    Lambda_Service -- "Ship Order Message Batch" --> Fulfill_Order
    Fulfill_Order -- "Order Picking Event [Message]" --> Message_Queue
    Fulfill_Order -- "Send Pick List [HTTPS]" --> Fulfillment_App
    Fulfillment_App -- "Fulfillment Mobile Device" --> Fulfillment_App

```

- and updates order status in DB

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

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, ...

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
- Spoofing messages is an issue for all BaaS services.
 - 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.

FaaS Tradeoffs

- No server state
 - All state needs to be saved (e.g. Redis, S3, ...)
 - Not just persistent state

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 is 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 is up to 15 minutes
- Startup latency
 - Functions take time to start
 - Some languages worse than others (e.g. Java)

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 is 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
- Server running function can be killed when function is not running.
 - Can occasionally send messages to functions to keep them alive.
 - Java has concurrency benefits over other languages.

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



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 hard.
- 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

[1] Pavlo Brunko.

Serverless architecture: When to use this approach and what benefits it gives.

<https://apiko.com/blog/serverless-architecture-benefits//>, March 2019.