Serverless Architecture

Software Architecture

Richard Thomas

April 20, 2026

Oxymoron 1. Serverless

Logic running on someone else's server.
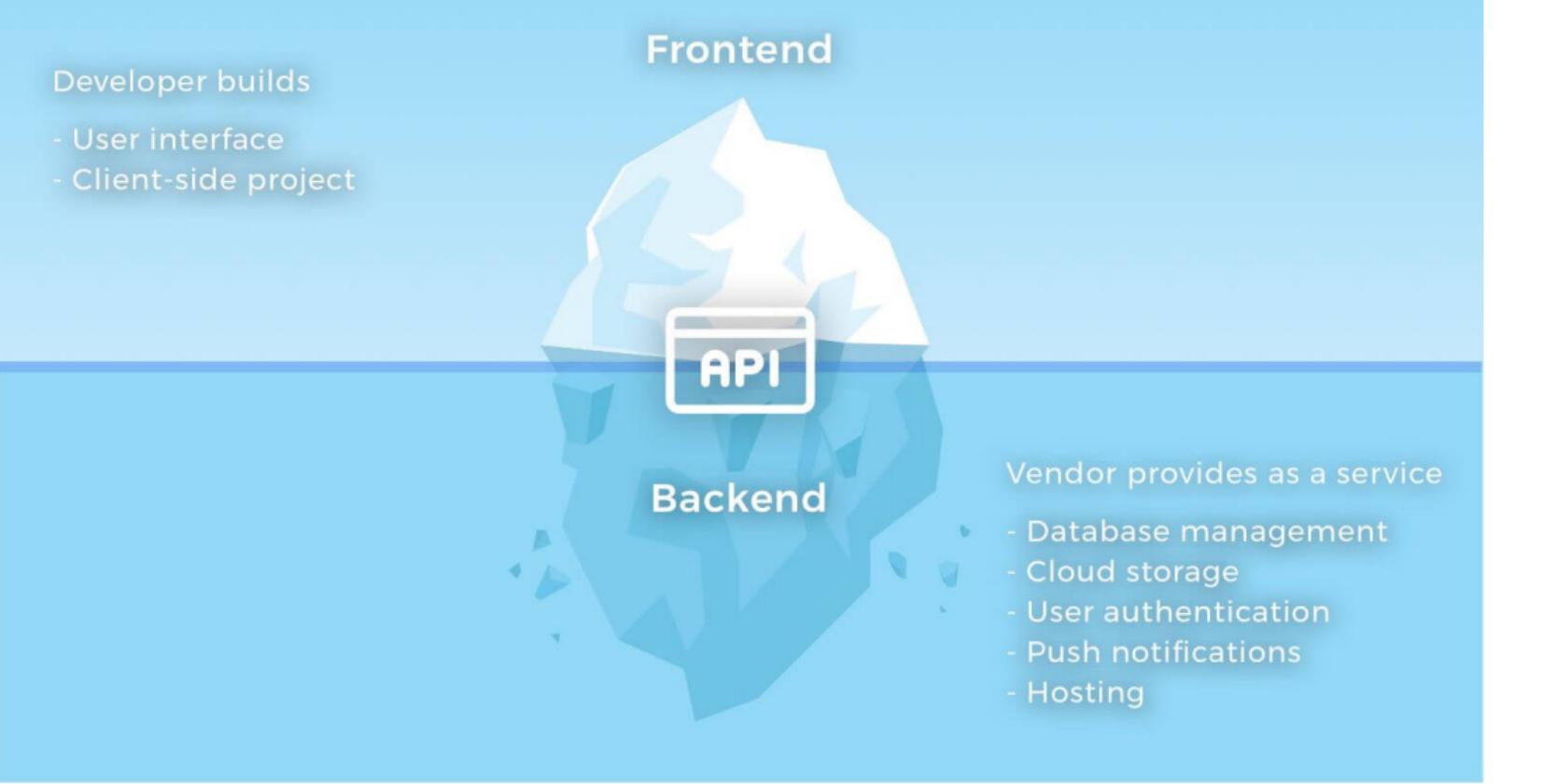
Developers can focus on logic, not infrastructure to deliver it.

> *Definition 0.* Backend as a Service (BaaS)
>
> Cloud-hosted applications or services that deliver functionality used by an application front-end.
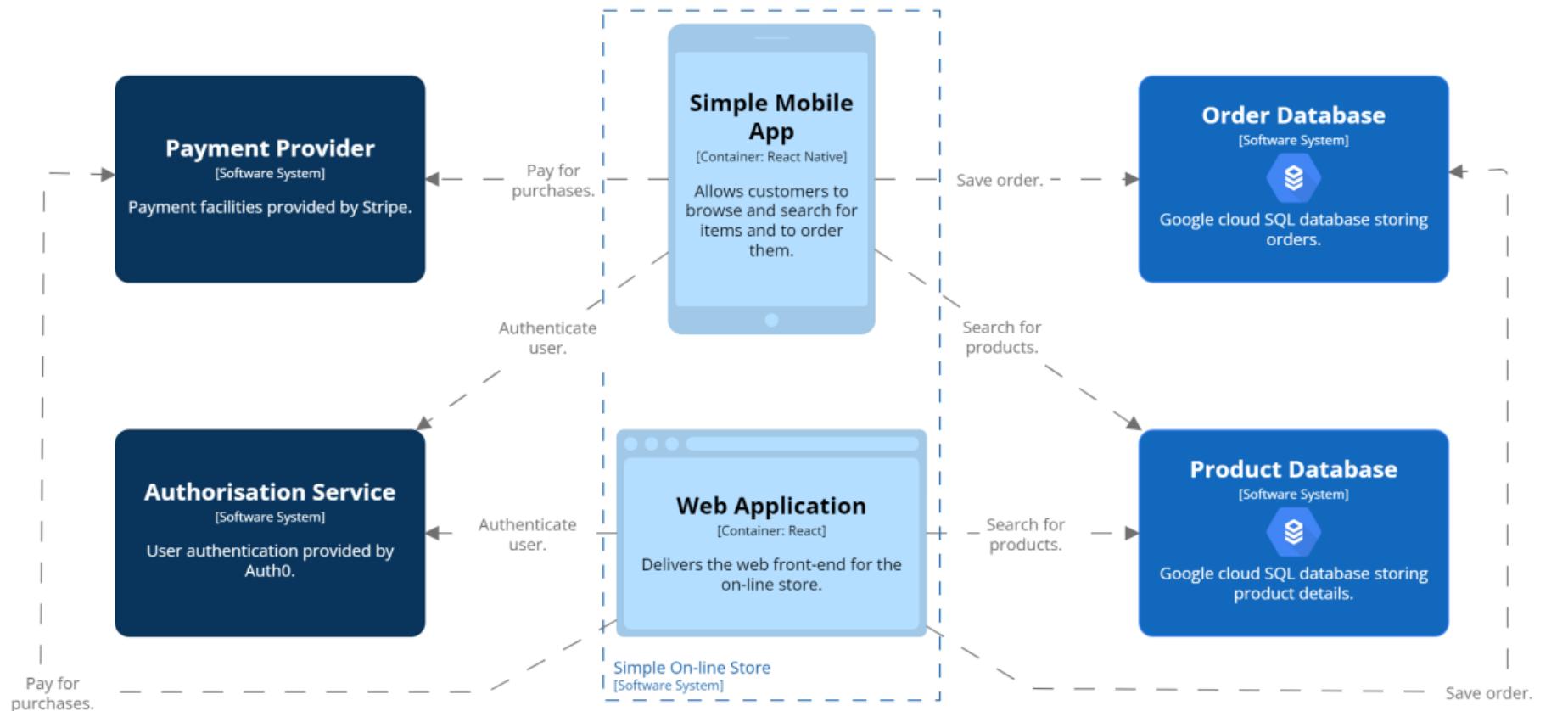
- Front-end may be a SPA or mobile app.
- Back-end provides sophisticated functionality (e.g. database, machine learning, location services, authentication, …).
- Front-end ties back-end services together to deliver the application's functionality.

BaaS Iceberg [Brunko, 2019]

Frontend

Developer builds

- User interface
- Client-side project

API

Backend

Vendor provides as a service

- Database management
- Cloud storage
- User authentication
- Push notifications
- Hosting
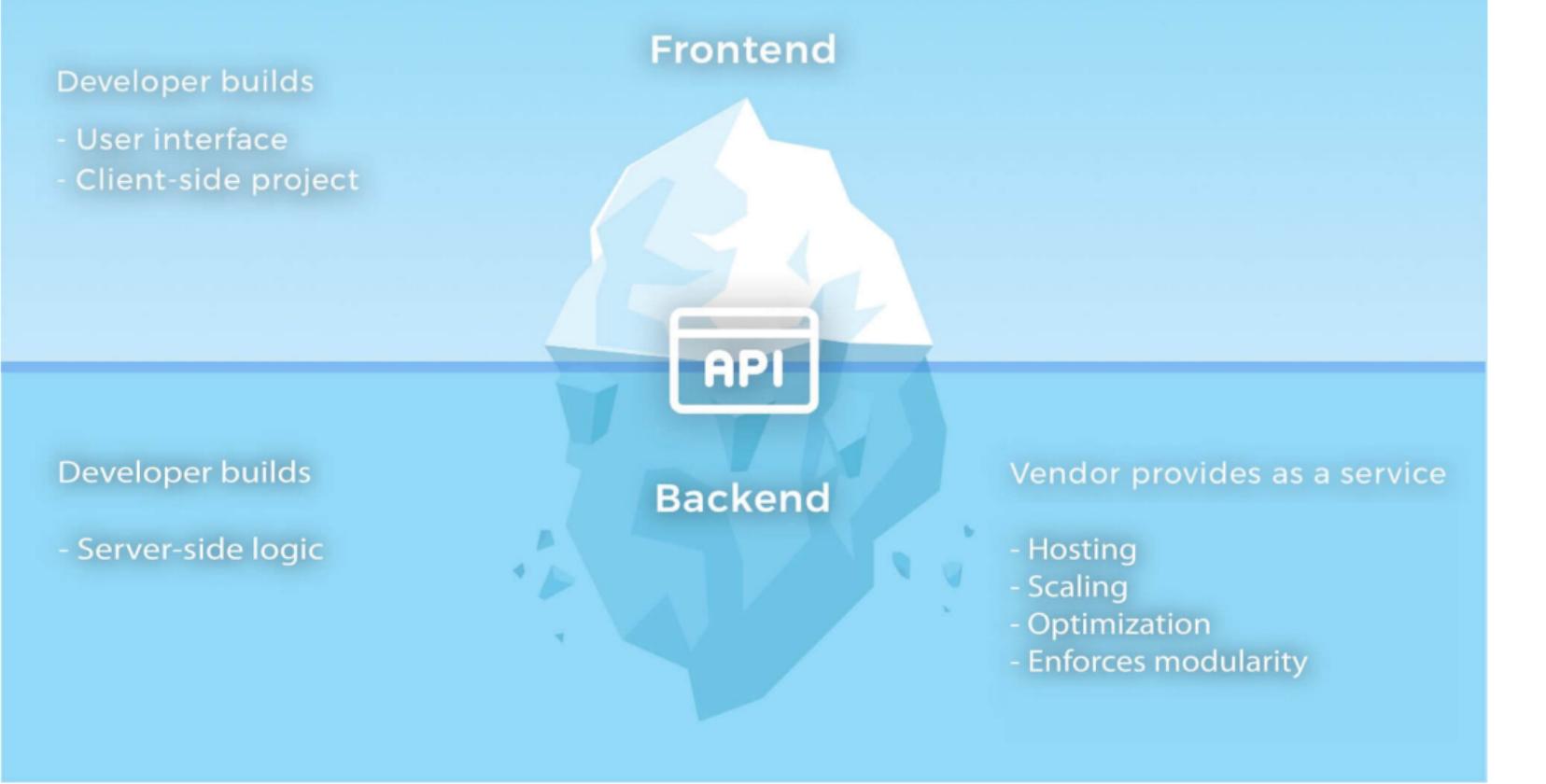
BaaS Example

- Example of simple system with back-end functionality delivered *entirely* via BaaS.
- Feature-rich front-ends coordinate behaviour delivered by BaaS.
- Consequence for Front-ends
  - Tightly coupled to BAAS
  - Contains both UI and functional behaviour logic – Poor Cohesion
- Front-end could have a layered design – many SPAs don't.

> **Definition 0.** Functions as a Service (FaaS)
>
> Application logic that is triggered by an event and runs in a *transient*, *stateless* compute node.
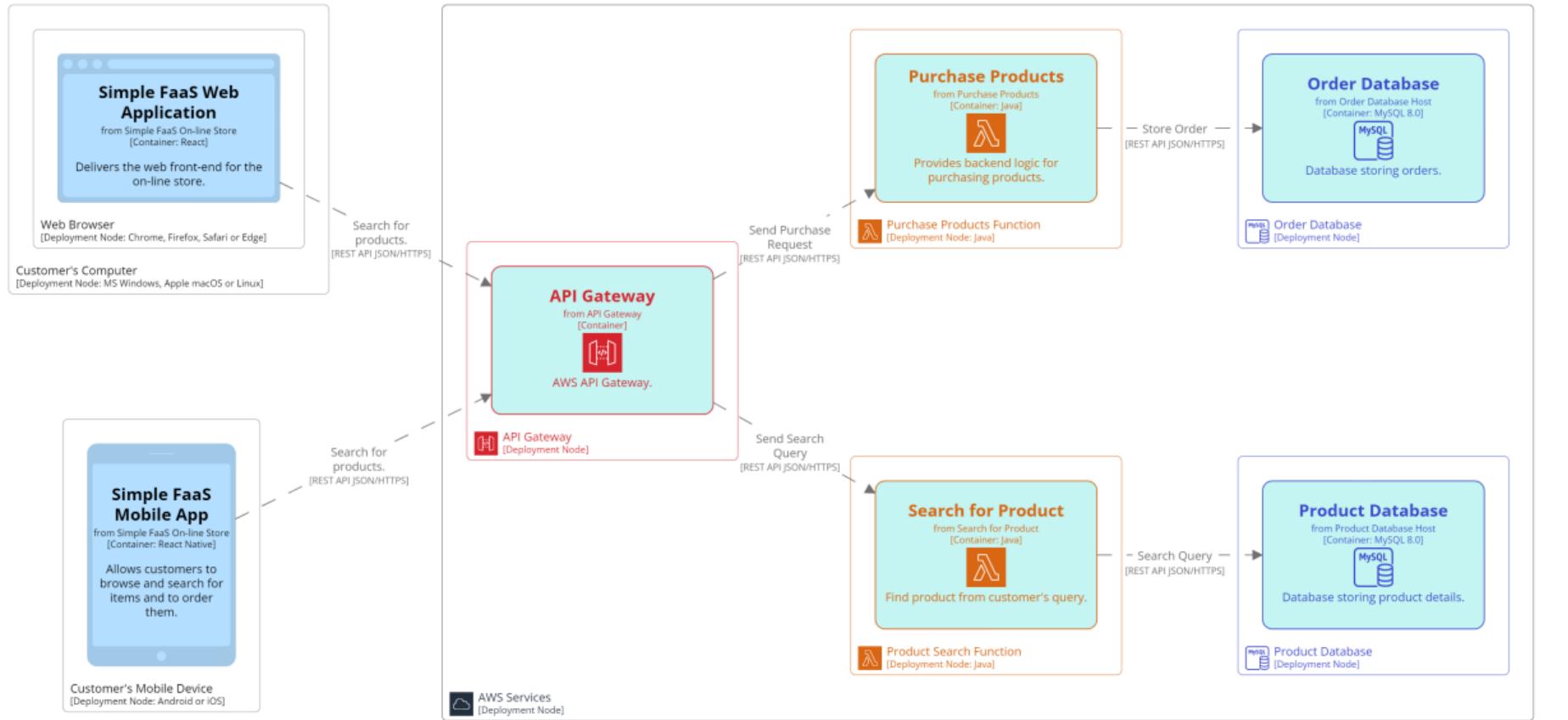
- Node may only exist for duration of function call.
- Server infrastructure (e.g. type of node, lifespan, scaling, ...) are managed by hosting provider.
- e.g. AWS Lambda, Google App Engine, Azure Automation, ...

FaaS Iceberg [Brunko, 2019]

Frontend

Developer builds

- User interface
- Client-side project

API

Backend

Developer builds

- Server-side logic

Vendor provides as a service

- Hosting
- Scaling
- Optimization
- Enforces modularity
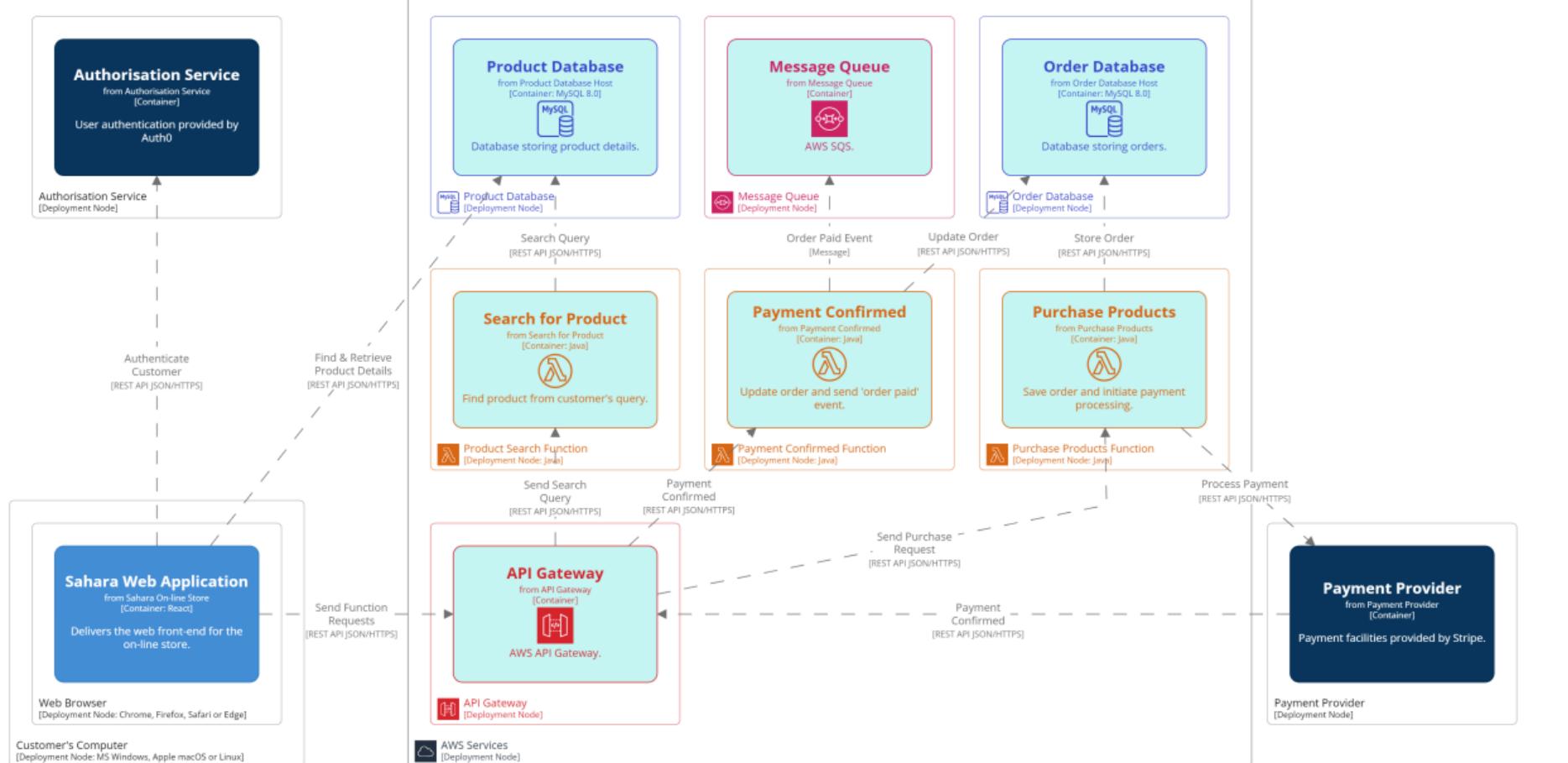
FaaS Example

- Example of simple system with back-end functionality delivered by FaaS.
  - Some services delivered via BaaS.
    (e.g. Authentication – not shown on diagram for simplicity.)
- 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.
- Allows a bit more separation between UI and logic.

> **Definition 0.** Serverless Architecture
>
> Software system delivering functionality through BaaS or FaaS.
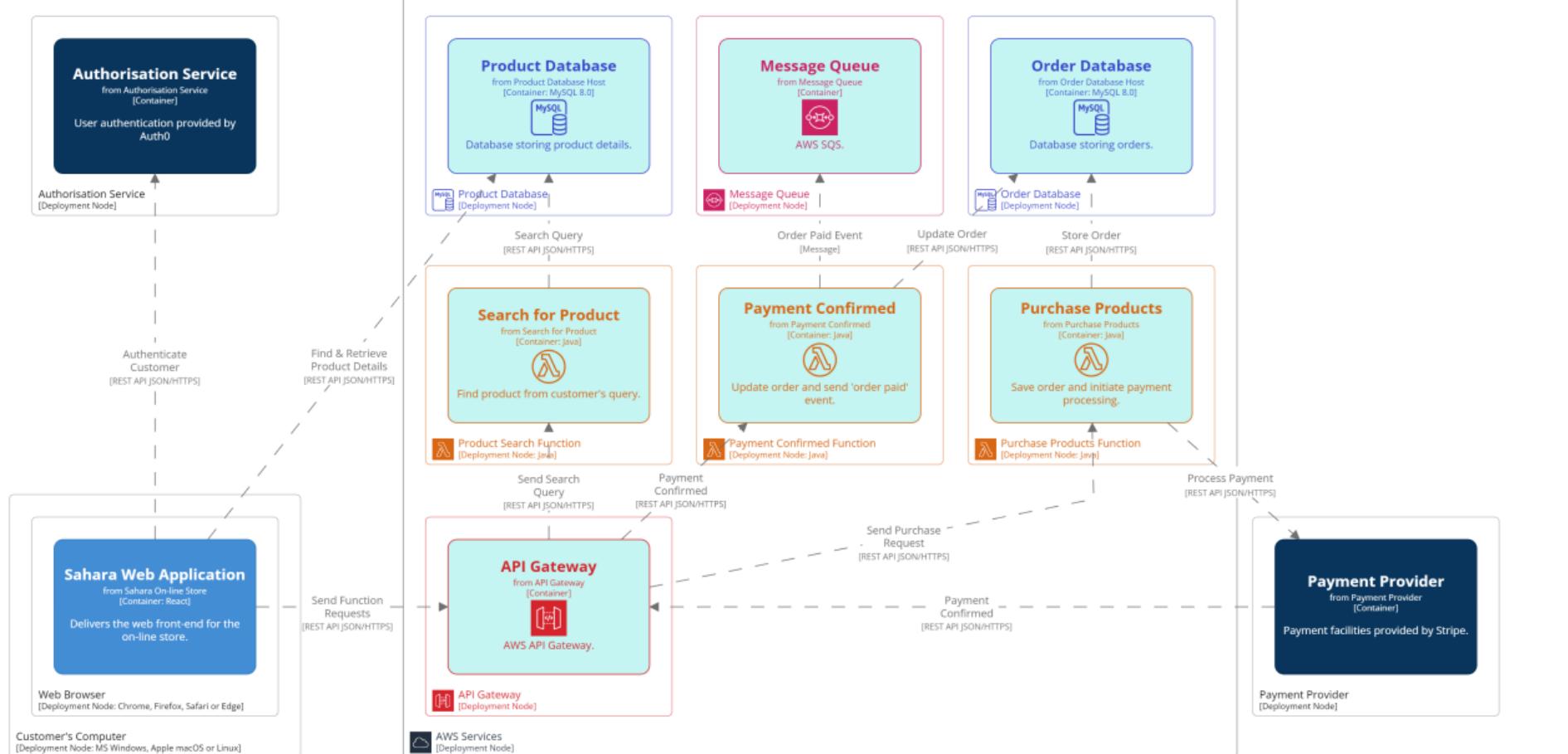
- Many people focus on FaaS when considering Serverless.
- Mobile App or Single Page Web App (SPA) coordinate services.
- Front-end ties back-end services together to deliver application's functionality.

Sahara Browse & Order — Serverless
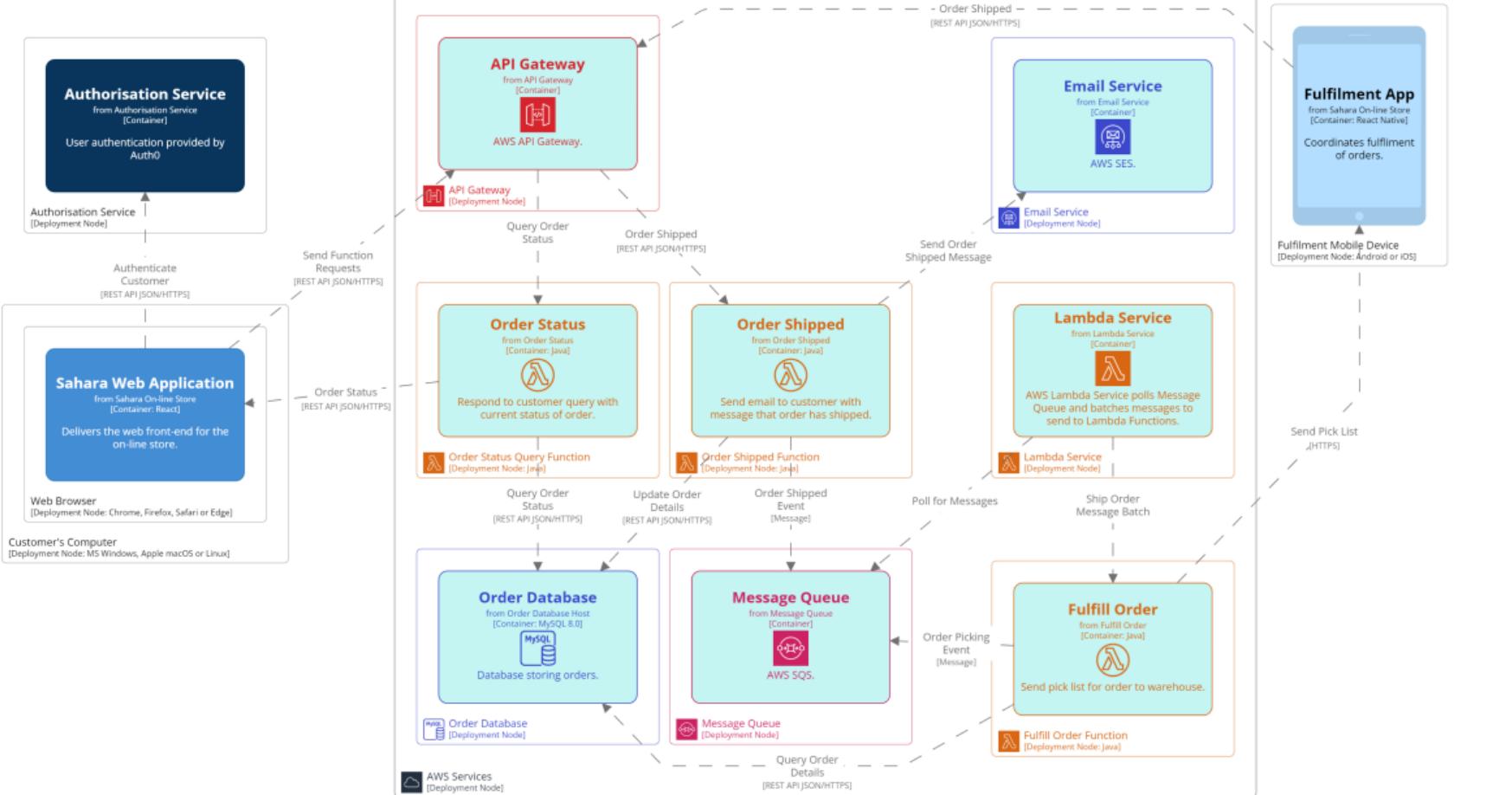
- Only browse, search and purchase are shown.
- Uses both BaaS & FaaS.
- Shopping cart is implemented within the web and mobile app for this architecture.
- Order Scenario 1: Customer checks out their shopping cart in the web or mobile app.
- Order Scenario 2: App calls Purchase Products function via 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 5-9: Notes continue on *next slide*.

Sahara Browse & Order — Serverless
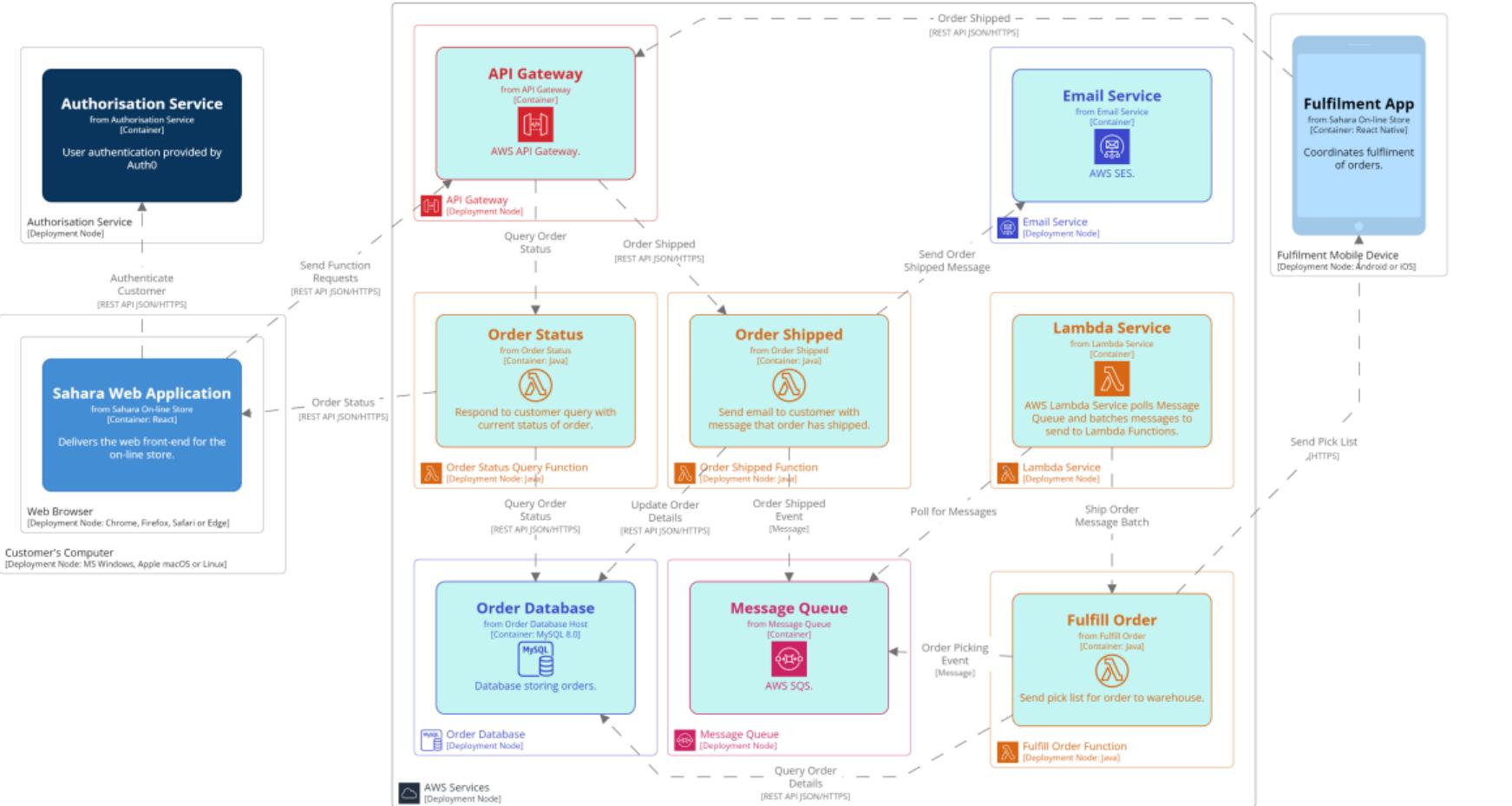
- Order Scenario 5: Payment success causes Payment Confirmed function to be invoked.
- Order Scenario 6: Payment Confirmed updates order in DB with payment status.
- Order Scenario 7: Payment Confirmed adds Payment Confirmation message to the Queue.
- Order Scenario 8: Payment Confirmation message is picked up by a fulfilment function to pack & send order.
- Order Scenario 9: Once order is shipped, another message would trigger an 'order sent' function.

Sahara Fulfilment — Serverless

- 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.
- Fulfilment Scenario 5-8: Notes continue on *next slide*.

Sahara Fulfilment — Serverless

- Fulfilment Scenario 5: Order Shipped sends email to customer and updates order status in DB.
- Fulfilment Scenario 5a: Simplification of order picking, packing and shipping steps.
- Fulfilment Scenario 5b: Each step sends messages to Queue to trigger other functions.
- Fulfilment Scenario 6: Customer queries order status in the web or mobile app.
- Fulfilment Scenario 7: App calls Order Status function via API Gateway.
- Fulfilment Scenario 8: Order Status queries Order DB and sends status back to customer via App.

- Automatic scaling
  - Multiple instances of function

- 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

- 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 — Not ideal.

- 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

- 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.

- 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

When is serverless appropriate?

When is serverless appropriate?

- Rich client apps with common backend
  - BaaS

When is serverless appropriate?

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

When is serverless appropriate?

- Rich client apps with common backend
  - BaaS

- High latency processing
  - Within function duration constraints

- Apps with variable load
  - Take advantage of auto-scaling

When is serverless *not* appropriate?

When is serverless *not* appropriate?

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

When is serverless *not* appropriate?

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

When is serverless *not* appropriate?
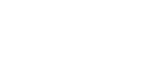
- Quick response required
  - Can't wait for FaaS to start

- Compute intensive processing

- Apps with steady load
  - Server-based approaches are cheaper

- 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 😄👍
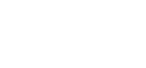
Interoperability 😄👍

Scalability 😄👍

Deployability 😄👍

Modularity 😐
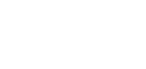
Testability 😐

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 back-end directly. No server-side protection of db or other resources.
- 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/.