IoT: Cloud Services

Serverless

What is serverless?

- Serverless architectures refer to applications that significantly depend on
  - third-party services (knows as Backend as a Service or "BaaS"), or
  - custom code that's run in ephemeral containers (Function as a Service or "FaaS")

Mobile Backend as a Service (MBaaS)

- Serverless was first used to describe applications that significantly or fully depend on 3rd party applications / services (‘in the cloud’) to manage server-side logic and state.
- Typically ‘*rich client*’ applications (think single page web apps, or mobile apps) that use the vast ecosystem of cloud accessible services
  - Think of single page web apps, or mobile apps that use the vast ecosystem of cloud accessible databases (like Parse, Firebase), authentication services (Auth0, AWS Cognito), etc

Functions as a Service (FaaS)

- Apps where some amount of server-side logic is still written by the application developer but unlike traditional architectures is run in stateless compute containers that are
  - event-triggered,
  - ephemeral (may only last for one invocation),
  - and fully managed by a 3rd party.
- Building and supporting a ‘Serverless’ application is not looking after the hardware or the processes
  - they are outsourced to a vendor.

Serverless Vendors

- Amazon Lambda - Run code without thinking about servers. Pay for only the compute time you consume.
- Google Cloud Functions - Lightweight, event-based, asynchronous compute solution that allows you to create small, single-purpose functions that respond to cloud events without the need to manage a server or a runtime environment.
- Azure Functions - Listen and react to events across your stack.
- IBM OpenWhisk - Distributed compute service to execute application logic in response to events.
- And many others…
  - A good list can be found: https://github.com/anaibol/awesome-serverless

IoT: Cloud Services

Deeper on Serverless

Serverless main concept

- The phrase “*serverless*” doesn’t mean servers are no longer involved.
- It simply means that developers no longer have to think "that much" about them.
- Computing resources get used as services without having to manage around physical capacities or limits.
- Serverless allows you to NOT think about servers.
  - Which means you no longer have to deal with over/under capacity, deployments, scaling and fault tolerance, OS or language updates, metrics, and logging.

Serverless vs Containers

- Take containers as an example. When you deploy a Docker container to Amazon ECS you still need to think about the hosting Cluster that your container will run on. You need to consider such questions as:
- Which Cluster would be best placed to run this container?
- Does the Cluster have capacity for my container's resource needs (CPU, memory)? If not how should I expand it?
- What is my strategy for deploying multiple instances of the container across multiple machines in the Cluster?
- If the Cluster has multiple types of machine within it, do I need to be concerned about that when I choose my deployment strategy?
- What are the security constraints of the Cluster, and do they need to be changed in order to properly host my container?

Benefits of Serverless

- The hosting provider
  - figures out the all the allocate questions for you dynamically
  - Guarantees it will have sufficient capacity for your needs
- Do not have to spend money upfront over-provisioning your host environment
- Not being constrained down the road by under-provisioning the environment.
- Not paying for idle
- Reliability and Availability are build in.

The Idea

- By composing and combining different services together in a loose orchestration developers can now build complex systems very quickly and spend most of their time focusing on their core business problem.
- These serverless systems can scale, grow and evolve without developers or solution architects having to worry about remembering to patch that web server yet again.
- A good serverless architecture can speed up development time and help to produce a more robust product.

IoT: Cloud Services

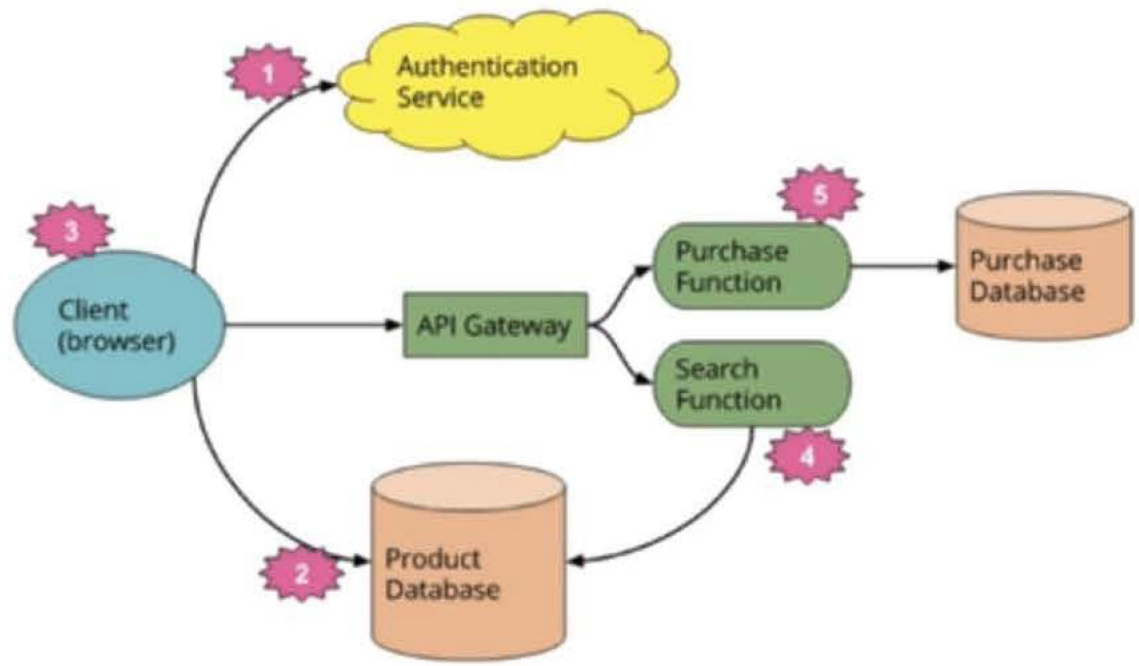
Serverless Examples

UI Driven app: The classic design

- A traditional 3-tier client-oriented system with server-side logic. A good example is a typical ecommerce app (dare I say an online pet store?).
- For example implemented in Java on the server side, with a HTML / Javascript component as the client
- With this architecture the client can be relatively unintelligent, with much of the logic in the system - authentication, page navigation, searching, transactions - implemented by the server application.

Serverless redesign of the Pet Store

- Deleted the authentication logic replaced it with a third party BaaS service.
- Client can have direct access to a subset of our database (for product listings), which itself is fully 3rd party hosted (e.g. AWS Dynamo.)
- Likely to have a different security profile for the client accessing the database in this way from any server resources that may access the database.

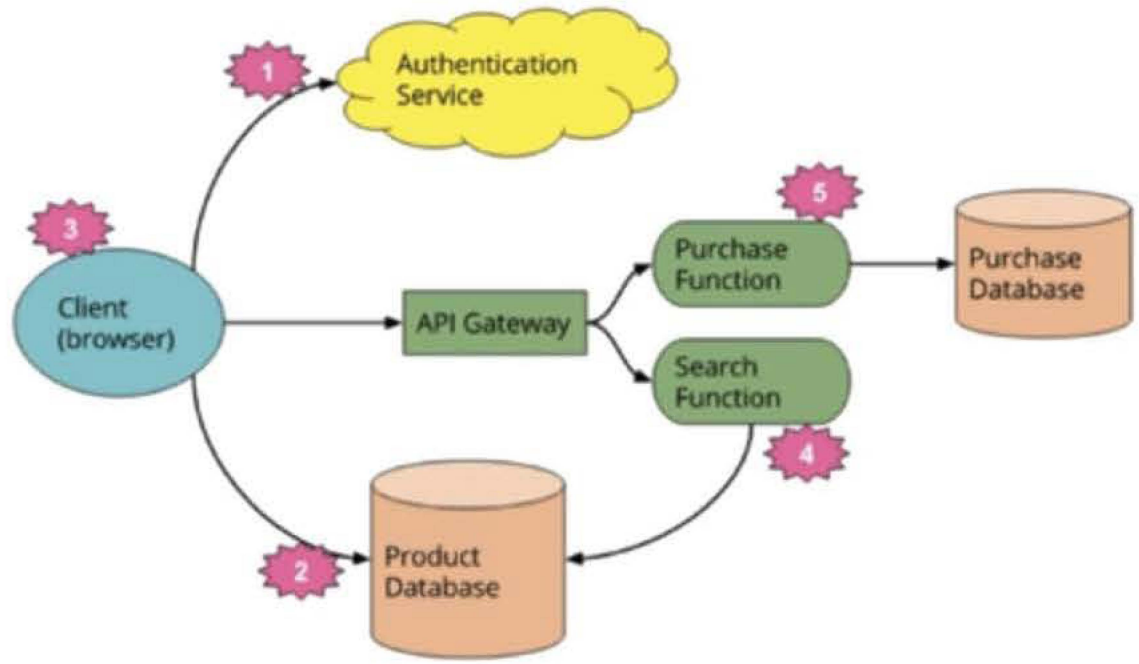


UI Driven app: The classic design

- A traditional 3-tier client-oriented system with server-side logic. A good example is a typical ecommerce app (dare I say an online pet store?).
- For example implemented in Java on the server side, with a HTML / Javascript component as the client
- With this architecture the client can be relatively unintelligent, with much of the logic in the system - authentication, page navigation, searching, transactions - implemented by the server application.

Serverless redesign of the Pet Store

- Deleted the authentication logic replaced it with a third party BaaS service.
- Client can have direct access to a subset of our database (for product listings), which itself is fully 3rd party hosted (e.g. AWS Dynamo.)
  - Likely to have a different security profile for the client accessing the database in this way from any server resources that may access the database.



A serverless view

- These previous two points imply a very important third - some logic that was in the Pet Store server is now within the client,
  - E.g.: keeping track of a user session, understanding the UX structure of the application (e.g. page navigation), reading from a database and translating that into a usable view, etc. The client is in fact well on its way to becoming a Single Page Application.
- Some UX related functionality can be kept in the server,
  - if it's compute intensive or requires access to significant amounts of data. An example here is 'search'.
  - For the search feature instead of having an always-running server we can implement a FaaS function that responds to http requests via an API Gateway (described later.) We can have both the client, and the server function, read from the same database for product data.

Another view

- Since the original server was implemented in Java, and AWS Lambda (our FaaS vendor of choice in this instance) supports functions implemented in Java, we can port the search code from the Pet Store server to the Pet Store Search function without a complete re-write.
- Finally we may replace our ‘purchase’ functionality with another FaaS function, choosing to keep it on the the server-side for security reasons, rather than re-implement it in the client. It too is fronted by API Gateway.