LinearBase

The 16 Configurable Topologies of Cloud*LESS* Data Processing

Contents

Summary

Cloud*LESS* is new, built from the ground up with operational data, analytics, availability, simplicity, and the environment in mind.

The practice of installing always-on paid-for third-party database products on a server is outdated, as is the concept of a backend. Instead, at just 4Mb, the CloudLESS portable code library incorporates data processing and interactive data streaming capabilities into any .NET or Java app or program for use internally within an application or as part of a collaborative micro-service network.

Distributing processing to on-demand devices helps reduce the dependency on datacentres and the cloud which in turn directly impacts carbon emissions.

The ability to model new or transform existing rigid-schema data assets into standardized relational databank files provides a consistent data storage and process almost any-data practically anywhere architecture.

There is no pay-to-be-on subscription or per-core licensing. Start and stop apps or programs in response to demand and pay only for the services used.

The key to CloudLESS is design-time, run-time, or on-the-fly dynamic profiles producing a configurable just-in-time mesh of data-processing micro-service hosts across mobile, desktop, datacentre, Cloud, etc.

Profiles join-the-dots and enable incremental simultaneous mix and match of 16 data topologies.

As an example, the CDN and Offset topologies considerably reduce resources required within a business by moving the majority of data processing to consumer devices. GDPR and Peer-to-Peer topologies ensure private, sensitive, or regulatory data remain under the consumers' control.

Instead of API's, businesses expose query processing, analytical SQL, and search capabilities to third parties using a simple reference that links apps or programs; there's little or no development cost.

Integrating outsourced development projects, or third-party products from the Cloud*LESS* app marketplace, just works due to a standardized approach to implementation and discovery.

Profiles

Follow the links, for example PROFILE1, to review Profile configuration, location, and function settings relating to each topology.

Introduction

Before discussing topologies let's investigate the difference between "traditional" databases and CloudLESS from LinearBase.

The term "traditional", in this context, is all-encompassing and covers many product-specific features and implementations, or should that be limitations: SQL[1] vs NoSQL vs NewSQL, Centralized vs Decentralized vs Distributed, Structured vs Semi-Structured vs Unstructured, and so on.

Regardless of the features, the common denominator across businesses is that the backend data models and data processing products become fixed assets, a constant, with data the variable, whether it's transactional relational models, analytical dimensional models, document stores, or

[1] Structured Query Language

data lakes. It's sometimes easier to introduce a new system and duplicate some data rather than change an existing one.

The result is custom APIs, transformation services, and message queues to provide data interoperability and synchronization between systems. Systems often feed into one or more critical aggregators to provide business-wide analytical resources.

Supporting and dependent systems become tightly-coupled with their associated schemas making change or migration difficult or near impossible. Schema changes must flow through each layer, and step, often involving multiple teams with varied skillsets.

In contrast, Cloud*LESS* inverts this design, making data and data processor the constants and data models the variable; data and processor are now separate, but consistent, entities. This means data can exist anywhere and enabled apps or programs can process almost any-data, practically anywhere.

An enabled app or program is one that integrates CloudLESS functionality into its codebase creating a micro-service host that's capable of processing relational data internally. This removes the need for a separate external database product.

Whether modelling new domains or digitally transforming existing data assets, CloudLESS persists data as incremental databank files in dedicated file-system directories. So, regardless of the original database vendor, data shape, format, or location, Cloud*LESS* provides a uniform data storage and processing platform.

The key to it all is design-time, run-time, or on-the-fly dynamic configuration profiles that marry databank data files with enabled apps or programs and register associated service URIs and features. This produces a configurable just-in-time mesh of micro-service hosts that collectively distribute relational data processing, but appear as one.

Hyper-Connected services automatically know how to communicate with each other and what their role is, there is no need for intermediary steps or transformations.

Schema changes propagate automatically, immediately, and are backward compatible. In Cloud*LESS* the schema is an instruction or set of instructions, the same as any other insert, update, or delete transaction. Enabled apps or programs interpret these instructions at runtime.

Whilst only .NET and Java implementations provide internal data processing capabilities, other gRPC enabled programming languages benefit from data streaming and the same low-code profile driven development syntax and inline data availability; data just exists somewhere as defined by profiles.

Migrating critical aggregators and their dependent, often monolithic or out-of-support, systems is now possible, practicable, and a lot easier.

It's time to forget almost everything you know about databases. From now on, all you need to know is how to model domains and ensure databank files are available, plus, how to create and manage profiles to specify how and where data processing occurs; it's all in the profile. Data only surfaces in the app or program.

CloudLESS = Databank Files + Data Processors + Profiles. At last, an actual digital transformation not just legacy processes and data moved to the Cloud.

The 16 Configurable Topologies of Cloud*LESS*

To explain the 16 topologies of data we'll follow the evolution of a fictitious ticket resale business, we'll call TicketyBox, that may have begun trading around 2000.

When this non-existent business began, the initial setup centred around a single RDBMS[2] hosted on a server housed in a rack based in the office, generously referred to as 'the On-Premise Datacentre'. Customers used a website to search and book tickets. Internally, employees use a mix of web applications and Windows™ desktop programs to manage customers, content, and analytics.

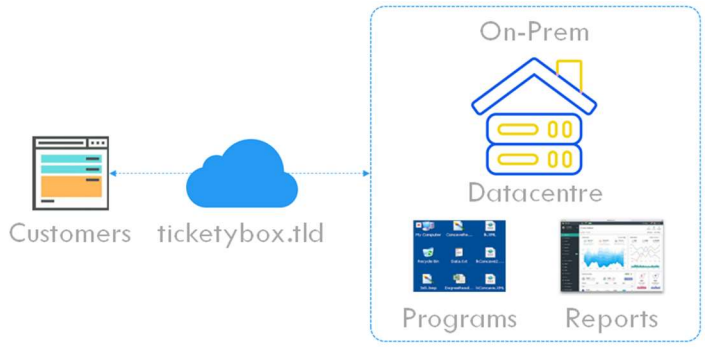

Figure 1 - TicketyBox Circa 2000

Over time, mobile became as important as web for customers resulting in additional development across Android and iOS devices. The mobile developers introduced a document store database to simplify development, this had the effect of creating a one-to-one relationship between app and data storage.

As the business grew, and amid concerns over scale and resilience, data processing moved to an Off-Premise datacentre. Data warehousing became necessary to separate analytical processing from the, now numerous, operational transactional databases.

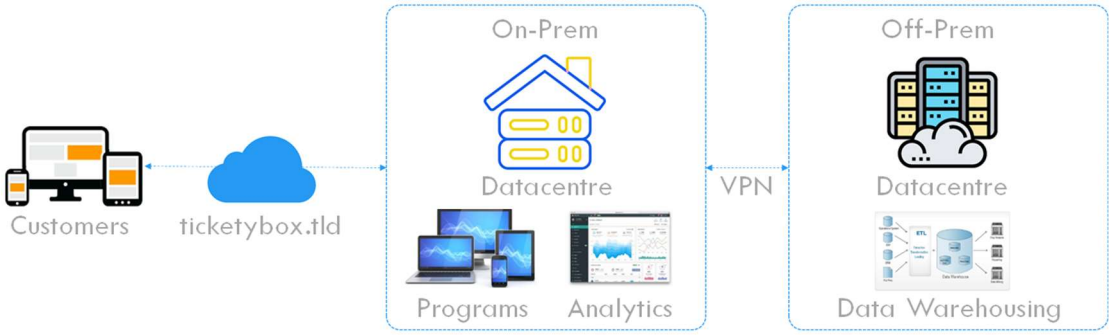

Figure 2 - The Arrival of Mobile, Off-Premise Datacentre, and Data Warehousing

More recently, the RDBMS databases are nearly out of support and need migrating to the latest version, again! Some systems now run in the Cloud so there is pressure to follow suit with the operational databases.

Updates to apps, programs, and web projects increasingly take longer and are more complex to complete. Even simple changes must propagate across front-ends, back-end databases, data warehouses, critical aggregators, and analytical reports.

The transition of data through the various vendors and products, from transactional to analytical, requires many development steps that usually transform data back and forth between binary, JSON[3], XML[4], and CSV[5] data formats; binary to human readable text format and back essentially.

[2] Relational Database management Service
[3] JavaScript Object Notation
[4] Extensible markup Language
[5] Comma Separated Values

Due to historic development decisions the business has GDPR[6] issues and duplicate versions of the truth dispersed across many sites.

There are now multiple programming languages, storage mechanisms, and development techniques resulting in difficulty hiring personnel and increased technical debt.

The business often suffers from customer overload when too many try to book tickets at the same time. It is difficult, with the technology stack in use, to manage peak load without paying for anticipated peak load even when it's not needed; a situation often overlooked when migrating to the cloud.

Data processing is currently managed by third party COTS[7] products that typically use a pay-to-be-on subscription or per-core licensing models.

So, how can Cloud*LESS* simplify the data processing landscape, reduce costs, consolidate development, increase customer satisfaction, and help the business be greener?

The primary data model is relatively simple, we have Performance, Venue, Artist, Ticket, and Customer databases; so, let's go ahead and digitally transform them into standardized databank files.

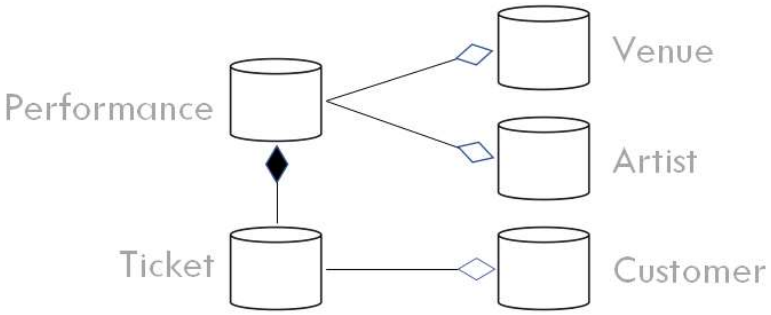

Figure 3 - TicketyBox Data Model

Please read the document on '[How to digitally transform an existing database into a Databank program]'.

We take this opportunity to introduce a domain driven design and model each database separately. Each resulting databank receives a context unique alpha-numeric identity. For this demonstration we'll assign:

- Performance = WM1WALQ0
- Venue = TRQTVXQZ
- Artist = 4FPM496M
- Ticket = TBJVPJBV
- Customer = AEJXA6EY

[6] General Data Protection Regulation
[7] Commercial Off the Shelf

The resulting directory structure looks like this:

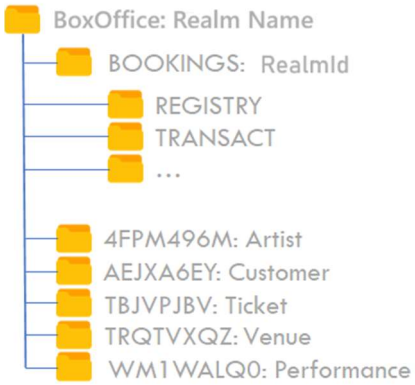

Figure 4 - Directory Structure

Next, we introduce relationships between the resulting databanks so they appear as one but data processors are able to manage reads and transactional writes independently, practically anywhere.

To connect databank to data processor we create profiles. Profiles are key, they tell the data processor the location of databank data to load into memory, the URL of other micro-services and the data they hold, and which functions to load. Functions provide serverless-function capability to encapsulate custom logic, similar to an API or database procedures, functions, and packages.

Cloud*LESS* uses profiles to describe topologies, their features, and to specify how and where data or serverless-function processing occurs. Each app, program, databank, or function can belong to multiple topologies simultaneously.

Next, we auto-generate code that represents the databank structures and any functions we created. This eliminates the need for Object Role Modelling scaffolding such as Hibernate or Entity Framework. Data appears inline, there are no dependencies, databases, or connection strings.

With modern development environments, it is possible to have one solution covering mobile, desktop, and web development. The ideal situation is to create new apps or programs to adopt modern development practices and procedures, like CI/CD[8] and DevOps[9].

Alternatively, for upgrading existing apps, programs, and web projects, we add our auto-generate code and replace the legacy data access logic, substituting it with Cloud*LESS* Read(…), Write(…), and serverless-function method calls that replicate database stored procedures, functions, and packages.

The expected outcome is cross-platform apps or programs implementing Cloud*LESS* initialised through design-time configuration. This means, every time we start the app or program or open the web page, the built-in Publish(…) mechanism calls the system Registry service to read and apply the specified profile or profiles. So where does this Registry service come from? Introducing our first of 16 topologies, Standalone.

1 – Standalone

Standalone is the out-of-the-box setting of most databases, that is, they're running system data, and possibly some business data, but there are no connections to it, yet. Cloud*LESS* is no different, it has a number of system databanks that represent and support the Realm.

[8] Continuous Integration and Continuous Development
[9] Development and Operations

A Realm defines the transaction boundary. A business may have multiple internal Realms or interact with any number of external Realms.

Okay, let's get this digital transformation started. We'll begin by creating our first service providing access to the system databanks and, through a boot profile, instruct the new service where to load the databank files from. We then need to establish a BootStrap profile to inform other services, within the Realm, how to invisibly interact with our new system databank service to access system services including profiles, transactions, etc.

The REGISTRY system databank stores all the Profiles that collectively define the structure of and associations within the Realm. The transformation process automatically creates and publishes default profiles with the relevant usage options, covering command line, configuration file, dockerfile, etc., for each newly created databank based on the information provided.

To initialize and serve the system databanks for our new Realm, we're going to use the Off-Premise datacentre and LinearBase Docker image containing a single Cloud*LESS* enabled console program. We ensure all the databank directories, created as part of the earlier database transformations, are in their predefined accessible location at the datacentre.

Each profile has a unique, automatically generated, uppercase alpha-numeric identity. For this demonstration we'll call this AUTOBOOT profile PROFILE1.

To initialize the new contained service, we need to load and initiate the boot profile PROFILE1. To do this we inform the service of the physical location of the REGISTRY system databank directory using an attached volume path (**in grey**) with the Realm name (**in green**) and Realm identity (**in orange**), along with the port the service should listen on, for example **/TicketyBox/BoxOffice/BOOKINGS 60001**. We do this via the container dockerfile script or run command. The service loads the system databanks into memory and we're up and running. We refer to this as the Primary Active Realm Service or PARS for short.

Each transformation generates four 'Initializer' files containing examples of each startup option: CommandLines.txt, Initializers.txt, LaunchOptions.txt, and docker-compose.txt. Each file contains options for Standalone, Centralized, Distributed, and Decentralized congiurations.

Now the container is running we have an internal, behind the firewall, IP address for PARS and the port it's listening on, let's say it is 10.1.1.100:60001. There's another automatically created boot configuration called the BootStrap profile, this time for use by any business apps or programs that bind to this Realm. For simplicity we'll say the BootStrap identity is PROFILE2.

The BootStrap profile contains the URI location of each system databank service, for example the registry service would be https://10.1.1.100/REGISTRY and the transaction service https://10.1.1.100/TRANSACT.

In this scenario, all the system databanks are hosted within a single PARS host, so the IP address is the same for each. Alternatively, we could have a service-per-system PARS setup resulting in individual IP addresses.

It is a simple configuration change to switch from single to multiple services on-the-fly. This is the primary advantage of the dynamic configuration-based approach; migrate between topologies as circumstances dictate or trial new parallel topologies in real-time.

We follow a similar process to add our operational databanks.

There is no definitive approach here, it all depends on how we want to distribute processing. Let's start simple and have all operational databanks defined in one profile; we'll explore alternatives later when developing profiles for other topologies.

So, similar to the system databanks, the profile specifies individual physical read-write paths to the relevant operational databank directories, for example **/TicketyBox/BoxOffice/TRQTVXQZ** for Venue, plus the BootStrap URI https://10.1.1.100/PROFILE2. For demonstration purposes let's say this profile has the identity of PROFILE3.

Each profile specifies variables that control every aspect of behaviour, such as data collation order, read-write access, latency options for consistency, plus lots more, allowing the 4Mb data processor logic to deliver multiple diverse processing scenarios.

We spin up another container and use https://10.1.1.100/PROFILE3 to initialize the service. This first retrieves PROFILE3 from the REGISTRY service at https://10.1.1.100 then uses the BootStrap URI specified in the profile's Boot section to identify and retrieve PROFILE2. We now have a new service with a notional IP of https://10.1.1.101, the operational data loaded into memory as defined in PROFILE3, and the URI locations of PARS as defined in PROFILE2.

2 – Router

The Router topology is a combination of a gPRC capable reverse-proxy forwarding requests to one or more CloudLESS services.

The primary purpose is to route external requests to the corresponding internal Hyper-Connected services based on the unique databank identity. Address-off locations specify the databank's host service URL and unique identity, for example, Venue would be https://10.1.1.101/TRQTVXQZ. No data processing takes place within the router service, it just redirects.

We'll save the router service profile as PROFILE4.

In this situation the router service can use the Agent version of Cloud*LESS*. This has no internal data processing capabilities but shares everything else and will be available across all gRPC enabled programming languages.

To access internal services, we need to create a one-off DNS entry to provide an IP gateway to the reverse-proxy. We'll use the subdomain approach and name it to reflect the Realm, let's call it Booking, so our external address is booking.ticketybox.tld.

We are still standalone as we have no services binding to our operational databank service at https://10.1.1.101. Introducing the next, and most widely implemented traditional database topology, Centralized.

3 – Centralized

In a central topology, most or all data processes are located at one site. Clients at remote sites communicate with a centralized service or services. Clients may be internal or external.

All requests automatically route to the relevant service, streaming as bi-directional collections of hierarchical binary object graphs, there is no restructuring as with rectangular data sets.

As with the Router topology, the Centralized topology client does not perform any data processing itself, it is just address-off locations. Here we can use the Agent version of Cloud*LESS* in our Centralized apps or programs.

The Centralized topology is popular with traditional database developments as Distributed and Decentralized are often complex or expensive to implement or require a different product line.

It is important to note that when a host cannot access the file system directly it must use a proxy host that can. This occurs when the host is outside the firewall and/or mobile devices such as Android and iOS.

Internal

Internal clients require profiles that reference internal IP addresses, unless the internal app or program consumes an external service, that is. For now, we will concentrate on connecting to our internal operational databank service at https://10.1.1.101. In this situation we can re-use the router service profile PROFILE4.

In our internal app or program, we add the profile URI to the start-up process, that is, the IP address of the REGISTRY service followed by the Realm and profile identity: https://10.1.1.100/BOOKINGS/PROFILE4. The profile informs the app or program of the URI location of each operational databank service.

External

External clients require profiles that reference public static IP addresses. We reference the router to forward external requests to the correct internal CloudLESS service.

For this, we create a new profile, PROFILE5, the same as PROFILE4 but using https://booking.ticketybox.tld in place of https://10.1.1.101, for example, Venue becomes https://booking.ticketybox.tld/TRQTVXQZ.

The only difference between internal and external apps or programs is the identity of the profile specifying the IP addresses and the need to specify a file system proxy URL. In the external app or program start-up we reference https://booking.ticketybox.tld/PROFILE5.

4 – Load Balanced

Load balancing provides real-time configurable distribution options for failover support, redundancy, performance, etc.

Cloud*LESS* optionally employs load-balancing when multiple services reference the same databank or serverless-function. Optionally, as there may be a valid configuration setting reasons to not load-balance, for example when one target service is read-write and another read-only, etc.

Each profile configuration address-off location is specific to the unique identity of a databank or function but, in load-balanced mode, has multiple hosting services URL rather than the usual single URL. Built-in algorithms specify how to distribute requests between the hosting services, such as round-robin, sequence, availability, weighted, etc.

The 16 topologies listed here require no internal app or program code-logic changes, code-logic remains unchanged.

Restarting an app or program, or invoking the Publish(…) mechanism, is all that's needed to implement a new or modified configuration.

The easiest to achieve is our next topology, Distributed.

5 – Distributed

Distributed topology shares processing of individual databanks across multiple services and locations. A databank is the single-version-of-the-truth. A service can process multiple databanks each with different configuration settings.

Distributed infers a form of load-balancing.

In-Process

Let's create a new containerized service in the Off-Premise datacentre with IP address 10.1.1.102. To initialize the new service either create a new profile or re-use an existing one, let's re-use https://10.1.1.100/PROFILE3. The result is two IP addresses, hosting the same databanks or serverless-functions.

To access the additional service, add load-balancing to the corresponding client profiles. In our case PROFILE4 is the router and internal client profile currently in use. Rather than modify PROFILE4 we'll replicate and call it PROFILE6 and reference the newly created service's URL denoting the operational databank it hosts, making Venue https://10.1.1.101/TRQTVXQZ and https://10.1.1.102/TRQTVXQZ, save, and the profile is live.

Finally, we change the services referencing PROFILE4 at start-up to PROFILE6 and redeploy. Clients implementing PROFILE6 attempt to distribute processing depending on the load-balance settings. Having separate profiles means we can use them across different scenarios simultaneously.

Elastic

Containerization and orchestration provide automated elastic scale-out capabilities; to demonstrate this we'll use Kubernetes within the Off-Premise datacentre.

Without going into too much detail, we configure Kubernetes Services and define fixed IP addresses that correspond to individual databanks starting with Performance at 10.1.2.10, venue at 10.1.2.11, etc. Internally, Kubernetes provides the elasticity, distributing requests via a gRPC capable load balancer in response to load.

To switch from in-process to elastic, replicate PROFILE4 as PROFILE7 and replace the URLs to address the individual Kubernetes Service IP address of each databank identity.

Now the business can leverage all the resources and SLA[10]s the Off-Premise datacentre has to offer depending on demand.

What happens when we want to use one or more Cloud providers with, or to replace, our Off-Premise datacentre. Enter the Decentralized topology.

6 - Decentralized

If we want to augment our existing Off-Premise datacentre services or multi-cloud options, we need to employ a global file-system[11] across datacentre and Cloud providers to enforce the single-version-of-the-truth blockchain databank requirement.

Each decentralized blockchain databank directory must be accessible to each platform in real-time with low-latency.

Making distributed into decentralized is then an easy process of replicating the installation across Cloud platforms by copying the container orchestration setup and configurations. All that remains is to replicate PROFILE7 as PROFILE8 and load-balance each databank URL per platform possibly using DNS entries to mask the IP address.

Setting the Router to PROFILE8 automatically enables external clients, implementing PROFILE5, along with any internal clients, also set to PROFILE8, to read from and transactionally write to any platform simultaneously.

[10] Service Level Aggreement
[11] https://www.ctera.com/technology/global-file-system/

But what about all that spare processing capacity we have at the office or offices such as PCs, tablets, or servers? Introducing the On-Premise topology.

7 – On-Premise

This topology requires we extend the global file-system to the office, or offices, we are going to leverage processing at.

In addition to the shared file-system, the On-Premise topology uses a multi-platform app, in development, that wraps the functionality required to register and de-register the app's host device so the system knows it is available to perform processing internally within the business.

The idea with the On-Premise topology is to not use a paid for alternative unless absolutely necessary; like having your own wind farm but with a backup electricity supply.

The On-Premise distributor is a dynamic load-balancing system service that redirects request to a relevant participating device in the same way it would in our earlier distributed topology.

Naturally, devices will toggle availability. It is the responsibility of the system to maintain a pool of available resources to switch between as required.

Having the capability to distribute processing using configuration alone is powerful. Let's extend our reach to expose internal data and functionality to the outside world. Introducing the Relational Aggregator topology.

8 - Relational Aggregator

The Relational Aggregator topology exposes read-only data processing and streaming capabilities to external business apps or programs or to third parties. Think of it as the ability to query any business information in real-time using the same Cloud*LESS* Read(…) and Invoke(…) syntax with any gRPC capable programming language.

Typically, a Relational Aggregator service employs a snapshot of data reflecting what the business wants the outside world to know; in essence, contextual SEO[12] advertising using consolidated analytical data.

This is the information search engines and data aggregation providers are interested in, that is, what's the data model, what data does it contain, how does the data relate, and how can they add value.

This is not the same as exposing a RESTful API, the business is providing, relational, analytical SQL, and search capabilities to external clients or third-parties, to use as though their own; a relational, controlled alternative to screen-scraping.

Whether it's creating aggregate relationships to extend their own domains or some other re-use, third parties consuming the service use TicketyBox's processing resources as the services are within the business.

It may be that we want to restrict access to this topology to only apps or programs belonging to the business. In this scenario, we are moving the processing to the edge of the business. This may be preferable to routing requests to live, operational, transactional, temporal data services.

Consolidating periodically changing data such as Performances, Venues, and Artists into analytical snapshots allows us to move processing to a Relational Aggregator service. If we create another DNS

[12] Search Engine Optimization

entry, let's call it Availability, we have an external address of availability.ticketybox.tld that we point to our availability Relational Aggregator service.

To take effect, it is a simple as replicating our external profile PROFILE5 as PROFILE9 and modifying the URL. Requests for Performances, Venues, and Artists go via availability.ticketybox.tld leaving Customer and Ticket to use booking.ticketybox.tld.

As the external app or program is initialized by a design time configuration, we need to reset it to reference https://booking.ticketybox.tld/PROFILE9 and redeploy for this change to take effect.

We could modify the internal profiles, PROFILE4 or PROFILE7, but internal apps or programs most likely need to interact with the live databank services to add or modify data.

Collectively, the topologies described up-to-now fall into our next topology, Client-Server.

9 - Client-Server

The topologies outlined so far are all classed as Client-Server, meaning the processing occurs in a hub-and-spoke arrangement leaving the client primarily for data entry or display purposes. The cost of data processing is borne by the business.

So how to change this, enter the CDN topology.

10 – CDN

The Content Delivery Network topology moves processing out of the business. It is still the businesses data but we're not providing processing resources, external apps or programs or third-parties provide this.

In our demonstration, we're going to use the Performance, Venue, and Artist analytical snapshot databanks created for the Relational Aggregator topology. Instead of fronting the data with a service, we expose the data raw on a URL, for example, https://ticketybox.tld/cdn/performance.cdn.

To implement our cdn files we'll replicate PROFILE9 as PROFILEA and change the Performance, Venue, and Artist locations to configurations. This will load the data on start-up into the app or program. Databank file sizes are typically a few megabytes, the size of a picture image.

Once again, we need to reset our external app or program to reference https://booking.ticketybox.tld/PROFILEA and redeploy for this change to take effect. In reality this topology is a design choice from the outset, updating the reference and redeploying is only necessary when switching topologies.

Now, all searches relating to Performance, Venue, and Artist take place on the device exactly the same as if hosted by the business; there are no code logic changes.

Similarly, transforming any data into standardized Cloud*LESS* data and exposing it via CDN makes it available to process by any permitted consumer, practically anywhere.

The CDN topology has saved the consumer a round-trip to the business and the business the cost of the resource required to perform consumer searches. As important, it has made data available for third-parties to utilize, possibly for a fee?

But we still have the cost of writing data back to business, when booking tickets for example. How can we reduce the burden on the business even more? Enter the Offset topology.

11 – Offset

With the Offset topology, the client app or program performs most of the transactional preparation work by creating the transactional databank files before submitting to the business, an optimistic approach.

All that is left to do is for the business to check concurrency and approve the transaction. The processing split is about 75% client to 25% business.

The combination of CDN and Offset topologies reduce the businesses processing overhead and cost whilst increasing reliability.

For the consumer the tangible effect is satisfaction. Previously, the business suffered from melt-down due to the tsunami of bookings when announcing high-profile tour availability. The Offset topology means there is a lot less for the business to do and can dedicate resources at the most critical point negating outages.

Once again, replicate PROFILEA as PROFILEB and change the Ticket location processing directive from Remote to Local, simple.

For consumers with older device, or who may not want to use the CDN or Offset topology approaches, it is the difference between profiles at start-up. Within the apps or programs, the functionality is the same but we now provide individual, profile specific, app or program versions.

The elephant in the room is the Customer databank. Why is the business holding personal data? TicketyBox historically used this data for analytical purposes, now they're in contravention of GDPR or CCPA. Regulatory topology to the rescue.

12 – Regulatory GDPR CCPA

The Regulatory GDPR CCPA topology implements the Standalone topology in combination with any other Client-Server, CDN, or Offset topology and has the effect of fire-walling personal customer data from business data.

The result is two customer data models, the business version containing publicly available or non-personal data and the consumers version for everything else. The business version has a relationship to the consumer version, so for development purposes they appear in the same hierarchical object graph.

So, let's create the consumer and lightweight internal versions. This time we don't transform as they don't exist, instead we model them using the in-development Domain Modeller App either manually or using a JSON Schema file. The new consumer Customer-Private databank has the unique identity of DFSUWKTS and the lightweight Customer-Internal is VRVX9Q42.

To the developer the only difference is a small change to the hierarchical object graph. When saving the object graph, the process automatically splits write actions into Realm and Domain specific bundles. This way, it is normal to create an entirely new object graph, such as an order, that spans multiple related databanks or Realms and have them saved in a single Write(objectGraph) step, probably across multiple Cloud*LESS* services.

Internally, the business now only has access to a consumers' unique identity and other GDPR compliant data. The app or program provides the union between data models and databanks.

As ever, the implementation is in the profile, this time we are creating two individual profiles. Replicate PROFILEB as PROFILEC to represent the business services. Modify the address-of location to reflect the Customer-Internal service.

Next, add a new profile section for our Customer-Private databank and call if PROFILED. PROFILED defines Customer-Private as a sandboxed Realm, albeit a very small one, in its own configuration containing its own system databanks and transaction boundary. The question is where to store the personal databank files. For this demonstration we'll use the device, but it could be any private low latency file-store.

Some may argue mobile devices come with SQLite pre-installed, and that it is easy to install on desktops, etc., so why not use this. They're missing the point. The SQLite approach introduces a dependency and ties the design into SQL based development with its associated requirements and limitations. With Cloud*LESS*, there's no dependencies and only one development syntax. Close the app or program and the OS[13] releases all related resources.

As it stands, our external apps or programs are connected to two entry-points within the business that load-balance requests across a mix of On-Premise, Off-Premise, and Cloud processors: availability.ticketybox.tld and booking.ticketybox.tld. This is a point of failure, time for the Load-Balanced Client topology.

13 – Load-Balanced Client

To load-balance on an external client we first need to expose our Cloud provisioned services as externally accessible gateways.

In PROFILE8 we created AWS and Azure exposure for all the operational databank services. Since then, we have gradually moved processing to the edge; to the extent the only two services of interest are Ticket and Customer-Internal.

Time to create PROFILEE from PROFILEC and modify our Ticket and Customer-Internal locations to include our Cloud providers. We already have our external DNS entry for internal services at booking.ticketybox.tld.

The Balance is set to Preference so we target internal resources in preference to paid-for alternatives unless the internal becomes unavailable; ideally internally we're using spare On-Premise resources?

Up to now the emphasis is on distributing data processing across datacentres and spare internal or compatible consumer devices through configuration. Introducing the Reward topology.

14 - Reward

The Reward topology pays consumers for allowing businesses to utilize their spare processing capability. The consumer may be an individual with a smart-phone or PC, or another business with their own spare processing capacity.

The Reward Client app is an extension of the On-Premise app and the Offset technique, running on compatible consumer devices creating a Community Cloud.

There are natural concerns around data sovereignty, security, and bad actors although it is possible to mitigate these concerns through Authentication Providers, data anonymity, at-rest and in-transit security, and other advanced techniques.

The intention is that businesses use the Community Cloud to process operational and analytical read requests; it is not intended for transactional write processing. The reward for the business is reduced processing costs when compared to other Cloud alternatives.

[13] Operating System

Statista.com estimates there will be 10 billion non-IoT devices such as smart-phones, laptops, and computers by 2025. That's rather a lot of unused data processing capacity; remember SETI@home anyone?

The penultimate topology is Peer-to-Peer.

15 - Peer-to-Peer

The Peer-to-Peer topology provides a ring-fenced environment where an actor has their own data but provides access to one or more other actors running the same app.

Suppose a business develops a messaging app. When consumers download and install the app from the relevant app marketplaces it generates a private Realm on the device. The Realm contains all the messages the actor generates. The app allows other actors with the same app to collaborate.

Normally, messaging apps hold messages centrally. With Cloud*LESS*, messages are stored only on the device of the message producer. Peers only see a rendered version of a message; they cannot download messages. That's not to say bad actors can't screen-capture a message and redistribute it outside of the group, there's no way to avoid this.

The message producer has complete control over the message and can delete or archive at any time.

Publishers can require Peers supply credentials provided by recognised Authentication Authorities avoiding rogue actors. As with the other 14 topologies, the emphasis is on participant transparency, visibility, traceability, and accountability.

16 – Offline

The Off-Line topology provides the ability to switch between online and offline invisibly.

Conclusion

Cloud*LESS* reduces the barrier to entry and time to value when developing data driven web, mobile, desktop, and embedded applications.

Combining all the desirable features of SQL and NoSQL databases into a configuration driven interactive micro-service process.

Simultaneously implement 16 topologies from Centralized, Decentralized, and Distributed to Regulatory and Offset via the Internet and Intranet across cloud, data centre, on-premise, and mobile app deployments.

Table of Figures