nucoris

Architecture

Jaume Canals

Spring 2019

Contents

1 Introduction

The architecture of a system describes its components and the interactions among them. **Architecture is important because it provides the foundation to achieve the desired system qualities**.

In this document we start by examining the qualities we expect from a healthcare information system (HIS). Then we review some common architectural patterns and Azure services, and how they could help us to achieve the desired quality objectives.

We end with an overview of **nucoris** architecture, the exploratory prototype of a cloud-native HIS that I have developed and used as a learning platform during a sabbatical leave.

2 HIS Quality Attributes

2.1 Availability

Availability is a very important quality in an HIS: clinicians need access to the patient record to make effective care decisions, and decisions may be needed at any time.

So, achieving high availability is a key objective of **nucoris**.

2.2 Performance

Good performance is important in an HIS since users often make important clinical decisions under stress and are less tolerant of slow systems.

2.3 Scalability

Scalability is a desired property in any system. In applications such as e-commerce sites, ticket-selling systems or media portals, high scalability is a must because their workloads may experience significant peaks, sometimes unpredictably.

In an HIS, however, the workload is quite stable. Even in epidemic outbreaks the demands on the system are implicitly throttled by the number of staff members and computers available in a hospital. So, achieving high scalability is less of a concern in **nucoris**.

2.4 Security

Securing patient data is a must in an HIS:

- Data must be secured at rest and when moving between servers and client applications.
- Data may need to be stored in the same region/country as the customer.

- Application must log every access to a patient record.
- User identification must be able to support two-factor authentication.

2.5 Usability

An HIS has a variety of users, which can be grouped into 3 main groups:

- *Administrative* staff from ancillary departments (reception, finance, HR, etc.)
- *IT*
- *Clinicians*

Some clinicians see the tasks they perform in the HIS as an administrative burden that distracts them from patient care. It's therefore important that an HIS is as user-friendly as possible.

That said, this is a quality that is out of the scope of **nucoris**, which focuses on exploring the cloud backend of the HIS.

2.6 Customizability

Because of differences in regulations and local practices, customers may need to collect different data types, or may have different workflows, some of them automatically activated in response to system events. **Nucoris** shall make it easy to support high degrees of customization.

2.7 Interoperability

In a health institution many different specialized systems may coexist: laboratory systems, radiology systems, critical care systems, inventory systems, etc.

It's therefore important to provide ways to interact with such systems.

2.8 Maintainability

Healthcare systems have long life cycles, because of a combination of factors:

- Slow innovation rate in the business side of healthcare
- Risks incurred by a major software upgrades in such highly available and interconnected systems
- Financial constraints in most health institutions

Whatever the reasons, it's important that an HIS has an architecture that can be tuned, extended and evolved over time.

2.9 Summary of the Quality Attributes

In the table below we rate the importance of each quality for **nucoris** (excluding usability, which we'll not be explored in this project). There are three levels, expressed with plus (+) signs.

Note that we are interested to achieve <u>all</u> the listed qualities: a low number of (+) signs does not make a quality irrelevant. It only means that when a trade-off between two qualities has to be made, we will prioritize those with more (+) signs. If both have the same importance, we'll do our best to find a reasonable trade-off.

Quality Attribute	Importance
Availability	+++
Performance	++
Scalability	+
Security	+++
Customizability	+++
Interoperability	++
Maintainability	++

3 Go to the Cloud or Stay on Premises?

Before delving into the architecture, it may be worth reflecting a bit on whether it makes sense for an HIS to be in the cloud.

3.1 General Considerations

Generally speaking, **the cloud is best when you are looking for**:

- Availability
- Scalability, especially when workload is very high or has unpredictable peaks
- Very large volumes of data, especially if they grow quickly
- Low up-front costs
- Advanced services such as machine learning or natural language processing.

The cloud may be <u>inappropriate</u> when:

- All your users are in the same local network
- Your company prefers *capex* over *opex*, and it has the financial muscle to set up the required server capacity
- You already have a datacenter and a properly staffed IT team
- You don't have a need for high availability or high scalability

3.2 Healthcare Applications

Healthcare systems fall somehow in between the two categories outlined above:

a) You need availability but not high scalability
b) Data volumes are large, but not very large (unless you also need to store radiology images or IoT sensor data)
c) Most of the users will connect from the same center, but some companies may own several hospitals in a region and share a single HIS for all. Occasionally specialists may connect remotely.
d) Size and skills of the IT team varies significantly from one center to another.

3.3 Security

Are cloud-based applications less secure than on-premises applications? Let's see:

1) **Connectivity**:
   First of all, nowadays most companies' datacenters are actually a set of machines hosted by external providers outside of companies' facilities, a setup that isn't much different from what you would have if your applications were hosted in Azure. You can secure your connection to Azure cloud in two ways:

   a. By setting up an Azure Virtual Network, where you can configure private IP addresses and define subnets, access control policies, etc.
   b. By using Azure ExpressRoute service to create private connections between Azure datacenters and infrastructure on your premises. These connections don't go over the public Internet.
2) **Data at rest.**
   Both relational and NoSQL databases can be encrypted at rest in Azure, and they are encrypted when replicated among Azure datacenters.
3) **Datacenter security.**
   There is no reason to believe that an Azure datacenter is less secure than your own datacenter, both in terms of physical security and quality of the infrastructure and staff. The opposite is more likely to be true.

3.4 Conclusion

In summary, we see some advantages in moving the HIS to the cloud, and no relevant disadvantages, so we think **it makes sense to explore the feasibility of a cloud-native HIS**.

4 Microsoft Azure Services

Microsoft Azure is a huge cloud platform with over a hundred different services.

In this section we list the ones that may be useful in a cloud HIS, together with the main quality attributes they would contribute to:

Azure Service	Description	Contributes To
App Service	Hosts web apps, mobile back ends, and RESTful APIs written in .NET, node.js, Java, etc., without managing infrastructure. It offers auto-scaling and high availability, supports both Windows and Linux, and enables automated deployments from GitHub, Azure DevOps, or any Git repo.	Availability Maintainability Scalability
Functions	Serverless compute service that can run a script or piece of code in response to a variety of events (http, timer, event bus…) without having to explicitly provision or manage infrastructure	Customizability
Logic Apps	Designer-based automation logic to build scalable workflows that implement business processes and integrate apps and data across cloud services and/or on-premises systems.	Customizability Interoperability
Service Bus	Messaging service you can use to send information between applications and services. The asynchronous operations give you flexible, brokered messaging, along with structured FIFO messaging, and publish/subscribe capabilities	Interoperability
API Management	Allows to publish APIs to external developers securely and at scale. You can create and manage API gateways for existing back-end services hosted anywhere	Availability Interoperability Security
Azure SignalR	Facilitates real-time web communication to applications, at a scale. Can be combined with web apps, functions, service bus, etc.	Customizability Interoperability
SQL Database	A relational database-as-a-service (DBaaS) based on the latest stable version of Microsoft SQL Server Database Engine. Fully managed, easy to scale up or out, automated tuning.	Availability Scalability
Cosmos DB	A globally distributed, multi-model database service that supports document, key-value, wide-column, and graph databases. Designed to provide low latency, elastic scalability and throughput, and high availability.	Availability Performance Scalability
SQL Data Warehouse	A data warehouse that combines SQL relational databases with massively parallel processing. Appropriate for warehouses larger than 1 TB.	Scalability

Data Factory	An integration service that can compose data storage, movement, and processing services into automated data pipelines.	Customizability Interoperability
Azure Active Directory	Directory and identity management service	Security
Key Vault	Safeguards cryptographic keys and other secrets used by your apps	Security
Virtual Network	Private network in the cloud, via VPN or a dedicated ExpressRoute connection.	Security

Note: in most cases I've taken the service descriptions from Azure website.

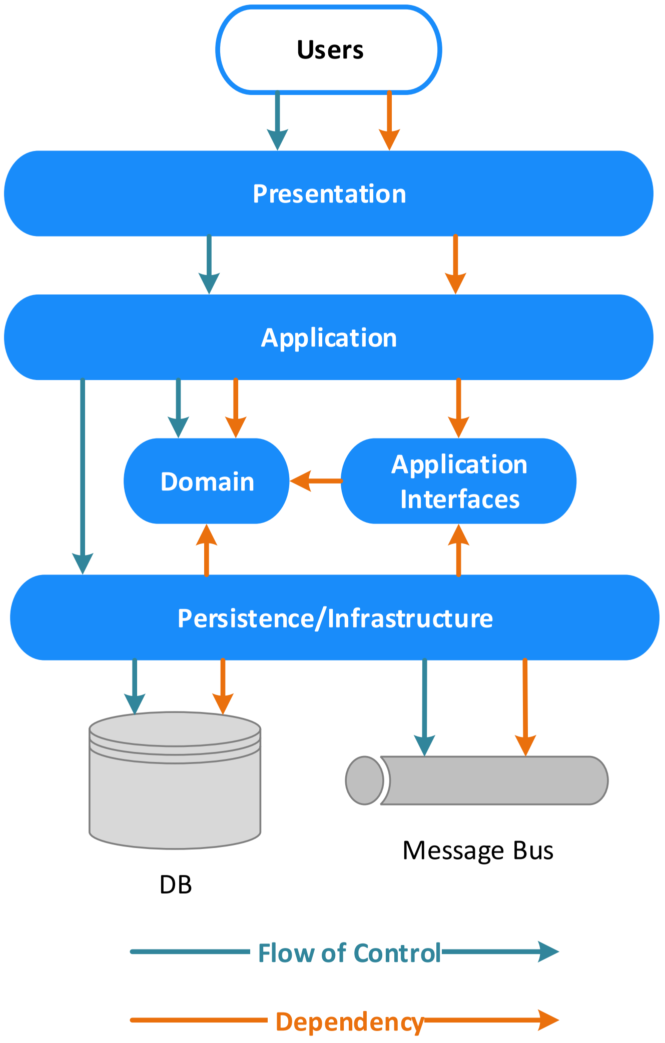
5 Architectural Decisions

5.1 Four-Layer Clean Architecture

A **clean architecture** is a domain-centric approach organized around the following principles:

1) The domain lies at the center and it has no dependencies.
2) Inner layers define interfaces; outer layers implement these interfaces.
3) The application layer implements use cases and depends on the domain and the interfaces.
4) Persistence, infrastructure and UI are pushed to the edges.

nucoris follows this architectural style, which is consistent with **SOLID** principles, facilitates testing and promotes loose coupling among layers, increasing flexibility and maintainability.



5.2 CQRS

CQRS is an architectural pattern that **separates the models for reading and writing**, under the assumption that system operations can be divided into:

- **Commands** that modify the system state.
- **Queries** that answer a question by returning some data without modifying state.

The separation can be applied at different levels: at the application layer, the persistence layer or the database, and you may even create a service for commands and a different service for queries in a microservices architecture. The more levels at which you apply CQRS, the more flexible but the more complex the resulting system becomes. So, there is a tradeoff that in **nucoris** I have resolved as follows:

1) There is a separation at the application layer between commands and queries.
2) There is a single database but there may be a different data model for certain queries. This is explained in detail in *nucoris Persistence* document, but in a nutshell:
    a. Queries on data of a *single patient* may use the same domain repositories and model as command handlers.
    b. Queries on *multiple patients* (used by the UI in searches or to populate certain worklists) will interact with specialized repositories and do not depend at all on the domain.
3) Command handlers interact with the domain entities to perform system state changes.
4) Query handlers interact directly with repositories. As said above, these may be domain repositories for single-patient queries or specialized repositories for multiple-patient queries.

The rationale for having a single database is discussed in the next section on event sourcing.

The rationale for the distinction between single-patient and multiple-patient queries is explained in the separate *nucoris Persistence* document.

5.3 Event Sourcing

CQRS is sometimes used with the **Event Sourcing** pattern. In event sourcing, actions on entities are modeled as a *sequence of events*. The application stores the events in an *event store*, which becomes the authoritative data source: to know the current state of an entity, you replay all its events.

The events are published to some kind of message bus where they can be consumed by interested parties. A typical usage is to maintain the current state in a *materialized view* where generic queries can be easily run to update application's UI.

Event sourcing is interesting because:

- It reduces contention and concurrency in write operations.
- It maintains a faithful audit record of changes done to an entity.
- It's easier to track entity state across time, and revert and apply back changes.

However, we have discarded event sourcing in **nucoris** because:

- There is some delay between the application adding events and the changes being reflected in the query model, hence in UI. This short delay is unlikely to become an issue in my opinion, but if a clinical mistake is ever made it could provide the basis for litigation: the customer could accuse us of purposefully choosing a design that cannot guarantee up-to-date UI data that drive clinical decisions. So, event sourcing in a HIS increases the clinical and business risk.
- An HIS is mostly an OLTP system. Even though the event store is in theory the authoritative data source, all relevant business and clinical decisions would likely be based on data read from the secondary query database (patient care, billing, payroll, etc.). To fulfill them you will end up implementing the query model as a normalized relational database, as complete and complex as if it had been the primary and only data repository. So, event sourcing in a HIS increases complexity.
- HIS are long-lived applications that will go through several major upgrades during their lifespan. Upgrading the schema or contents of a relational database to fulfill new requirements is usually quite straightforward with a SQL script. In contrast, changing the format of an event in an event store may require iterating through all the persisted events to make changes so they're compliant with the new format.

On the other hand, we would regret missing some of the value that event sourcing provides, such as faithfully tracking changes to entities, so in **nucoris** we have instead applied the related **domain event logging** pattern, which is a type of event sourcing where the current state is stored first, and the event last:

1) Command implementation in the application layer invokes domain methods on entities, which will trigger domain events.
2) The application layer gathers the events generated and synchronously persists both the modified entities and the events to the primary data source (a relational or document DB), which is also the source for queries.
3) Events are additionally sent to an application service bus, where optional Azure Functions or Logic Apps can asynchronously inspect them and perform custom business logic.

5.4 Relational or NoSQL Database?

5.4.1 Azure Cosmos DB

NoSQL databases are popular nowadays because they help to resolve a type of problems that have become more common and that are difficult to handle with a relational database (RDBMS). It's **a type of problems characterized by the 3Vs**:

- ***V**olume*: huge volume of data
- ***V**ariety*: data comes in different shapes
- ***V**elocity*: data grows very fast

NoSQL databases are better equipped to tackle such problems because they are designed to be **distributed, schema-free, and scalable**.

Microsoft's cloud NoSQL database is **Cosmos DB**. Its main characteristics are:

a) **Multi-model**: supports JSON documents, tables, graphs and columnar data.
b) **High scalability and volume**: you scale storage with horizontal partitioning and configurable guaranteed throughput.
c) **Global availability and performance**: for enhanced performance and availability you can configure a database to be available from different regions around the world, and Cosmos DB will take care of replicating the data. If you also deploy your web app to those regions, customers around the world are automatically connected to their nearest replica and therefore get equivalent performance levels.

5.4.2 Cosmos DB for a HIS?

Some of the qualities of a document DB such as Cosmos DB make it very attractive to a HIS provider:

- A HIS stores a high **variety of data**: physiological measures from clinical devices, lab test results, billing information, reports written by doctors, patient allergies, visit scheduling, clinician's profiles, etc.
  Moreover, quite often clients require the development of custom forms, with different data items.
  The schema-free characteristics of a document DB could help address this problem.
- A HIS needs **high availability**. Cosmos DB offers it by design, without the need to deploy complex solutions such as *SQL Server Always-On Availability Groups*.

On the other hand, there is a quality of RDBMS that document DBs lack: **the ability to efficiently perform ad-hoc queries**. Data in Cosmos DB is stored in *collections* of *JSON documents*. Cosmos DB indexes all properties of a JSON document by default (it can be configured) so you can efficiently query a collection by a document property, but there are a couple of significant constraints:

a) You can't perform a query join across documents, only among nested properties of a document.
b) A collection in a document DB is physically organized in horizontal partitions (that's why they can easily scale out). So, you have to designate a property of your JSON documents as the **partition key** (in our case it could be, for example, patientId). Documents with different partition keys are stored in different partitions. Running a query that spans several partitions is feasible but not recommended because the documents may be stored in different disks, and therefore it's not efficient.

These query restrictions make **document DBs inappropriate for OLAP systems and OLTP systems that don't need to scale**.

In conclusion: there is, oh surprise, a **trade-off between the capabilities of document DBs and RDBMs**. An HIS is a patient-centric system, and most of the queries could be efficiently served from a document DB, but queries to populate worklists with data from different patients would suffer if reading the data from a Cosmos DB collection partitioned on patientId. With this in mind, in this prototype we are willing to explore the following design:

1) We'll use Cosmos DB for day-to-day operations.
2) In Cosmos DB there will be *a collection for patient data* and a separate *collection for application queries*.
    a. The collection on queries will be used to feed the well-known application worklists. It will contain an almost flat document for every candidate item in the worklist, with properties needed for filtering and display.
    b. The persistence layer of the application will take care to update both the patient data collection and the query affected by the event.
    (since they will be in different partitions it can be done asynchronously)
3) For non-real time business analysis, ad-hoc queries, invoicing, etc. we will deploy a relational data warehouse. At specified periods an ETL process (implemented with *Azure Data Factory*) will copy data of interest to the data warehouse.

We are not sure that this design will work out, but we think in this prototyping phase it's worth testing this approach to see if we can achieve the availability, scalability and flexibility of NoSQL databases while maintaining the querying functionality required by the HIS.

5.5 Multi-tenancy Model

A multi-tenant application is an online shared resource that allows different users/companies (called the "tenants") to access it as if it were their own. An example of a large multi-tenant application is Office365.

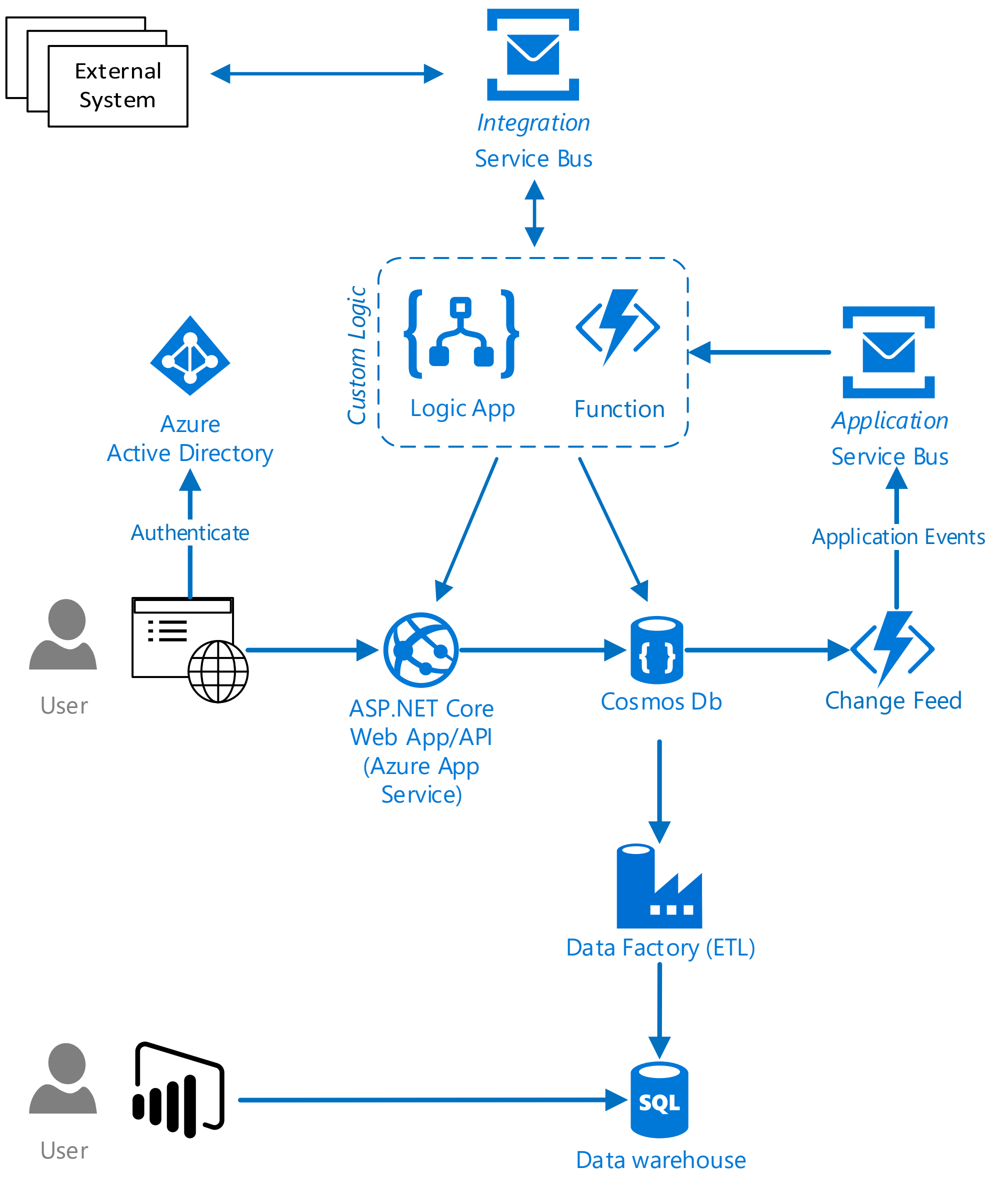
The following Microsoft article discusses the different ways to implement a multi-tenant application in Azure: https://docs.microsoft.com/en-us/azure/sql-database/saas-tenancy-app-design-patterns

The model that we've selected for **nucoris** is the **standalone single-tenant app with a single-tenant database**. In practice it almost amounts to saying that **nucoris** is not a multi-tenant application. This is more expensive in Azure than other options and has some management overhead, but it provides the *highest isolation* possible among tenants. Isolation is valuable for Nucoris because:

- **nucoris** deals with sensitive patient data. Maximizing the isolation among tenants may help to address customers' concerns about data security in the cloud.
- Customers are likely to request that the data is stored in the Azure geography closest to their premises, so at least a different deployment per geography would be needed anyway.
- Each tenant will need to interoperate with different on-premises systems through a different Virtual Private Network. This is easier to set up if customers don't share the same application/database.
- Each tenant may need specific customizations. This is also easier to set up if they don't share the application/database.
- The isolation makes it easy to schedule upgrades at different times, to minimize risks and possibly ask individual customers to pay a fee for the upgrade.

6 Architecture

This is an overview of the architecture we propose for **nucoris**:



In the frontend, an **ASP.NET Core** web app hosted by **Azure App Service** renders *Razor Pages* styled with *Bootstrap*. The applications exposes also a REST API that some pages use to post simple commands and do partial view updates.

In the backend, data is stored in **Azure Cosmos DB** document database. Domain events are also persisted. An **Azure Function** is bound to Cosmos DB **Change Feed** and picks and publishes events to an **Azure Service Bus** which we call the "application" bus.

Azure Functions and Logic Apps subscribe to those domain event messages in the bus. They implement custom logic as needed to extend the core functionality, and may publish integration events into an “integration” bus to which other systems can subscribe. They may also access other resources, including the REST API exposed by the application.

For business analysis, **Azure Data Factory** periodically transfers data from nucoris Cosmos DB into a data warehouse supported by an **Azure SQL Database**.