# Document

# Development Team Selection Guidance

## Development Team Lead Selection Guidance

## Developer Selection Guidance

### Conceptual Understandings Checklist

#### Background

The objective of Development Framework Suppliers (eg: .NET ASP Core, Django, etc.) is to develop a simple framework to facilitate adoption, by removing complexity – and options – not needed by most systems, especially if they are internal enterprise services.

This objective is understandable from a marketing perspective – but is demonstrably at odds with the objective of organisations to develop complex, powerful, evolvable, large, externally facing public capable services.

Developers who have learnt development on a commercial framework are capable of using the framework within its narrow abilities – but often do not have the computer science background or experience to understand the trade-offs and long term impacts of using the framework’s default approaches.

### Roles and Permissions

Most modern frameworks have stabilised on offering only Role Based Access Control (RBAC).

It is important that developers demonstrate an understanding that Roles only Logical Containers for a collection of System Module defined Permissions.

RBAC is demonstrably inadequate for flexible systems intended to be used for a long duration.

* Although most systems will start with a simple set of Roles (eg: SystemAdmin, Manager, Users) it quickly becomes evident under pressure from consumers that more and more are required to handle all scenarios (AccountantManager (ie, the Group Manager, so they are not in charge of anybody else’s Group), HeadAccountant (can do anything), SubAccountant (same as accountant but can’t sign cheques, AccountantGuest (external 3rd party auditors), etc).
* The above could have been solved more elegantly with permissions (CanViewPaymentRequests, CanDevelopPaymentRequests, CanSignPayments, CanCancelPayments, etc.) Associated to Roles. This way, if an guest auditor role becomes needed later all you have to do is give that role CanViewPaymentRequests without any worry of compromising the service (they do not have CanDevelopPaymentRequests). Same for if you want to develop a SubACcountantRole – they have the same Permissions as Accountant, excluding the CanSignPayments Permission.

### Roles and Groups

The above point touched on the fact that Roles are not universal. Although a simple system may get away with a single system wide Role (“Manager”) for a while, it should be apparent that any system quickly runs into the need for Roles for different groups (SystemManager, UserManager, CaseManager, etc, at the very least – not even discussing whether the system is intended to be used by multiple business groups.

Therefore RBAC based authorisation is more often than not insufficient for Enterprise development.

A more valuable system requires a design which is based on developing User Groups and associating GroupRoles to them (in turn +/- associated to one or more [System Logical Module] Permissions).

### Users and Groups

Note that in Enterprise systems it is common to think of Users belonging to Groups as no Enterprise Employee belongs to more than one (for payroll reasons).

This internal architecture is potentially appropriate for most traditional staffing and payment requirements – but little else, as it ignores the very real reality of external partners, consultants, users, potentially invited to participate in multiple internal groups, with different roles in each.

For example, an employee could be a manager in one group, while being an informed board member in another group. Another employee could be the team lead of their department’s peers, but a consultant member on another project.

External partners and consultants could be being hired in different capacities (ie roles) within multiple groups.

Etc.

The solution is therefore the opposite of Users belonging to Groups.

Users instead have Temporal (ie, have a defined Start/Stop Date) Roles (linked to Permissions) in multiple Groups.

### Users, Groups, Identities, Aliases

The computer industry is seeded by self-taught developers who use system development nomenclature interchangeably without an understanding of key differences.

User, Person, Identities are all distinct concepts.

**Persons** are*external* to a system, acting in the wider world, typing away at a keyboard, consuming the Service’s APIs or UI, as System Users.

Note that Persons can be Natural (dead or alive persons) or Logical (a Business Entity), but in most cases we’re talking about Natural Persons.

**Service Clients** are *external* systems consuming the APIs of the Service, as a System User.

Persons and Service Client Systems are the two primary forms of a System Users

**Identities**: A Person (and even Service Clients) always has multiple Identities. In the physical/real world, a person can be known as “Mum” as well as Mrs. Smith, or “Honey”. A company can be known as “MyCompany”, or if you are external to it, as “International Business Machines, Inc.” (or an Alias of it: “IBM”).

Although it’s commonly described that Person is a System User, it’s actually incorrect. It is not the Person that is the System User, but a Person (or Service Client’s) Identity that is a Service User.

**Groups:** A Person/Service Client’s multiple Identities are specific to Groups. In a technology environment, the system itself is the largest *logical* Group -- within which one can nest child groups (Tenancies, Organisations, Business Groups, etc).

**Identity Brokers:** A Person/Service Client is introduced to a Group (ie, the Service itself) via a Group-trusted broker. In the physical world, that’s a friend, and in the online world, that’s an independent Identity Provider (IdP) Service.

**[System Users]**: Once an Identity is introduced, internally to the system, an *internal* User record is created as an Internal model of an *external* Person’s Identity. Outside, it’s a person typing away at a keyboard, or a remote System acting as a Client of the API services made available by a service. Internally it’s just a User Record that other records refer to.

A Person, or Service Client, will be two different Users in two different Systems.   
It would be incorrect to describe a single User (as opposed to a Person) being the same in two different Systems.

**[System] UserID:** within the system – a Group in its own right -- a Person/Service Client using the System is a [System] User, referenced via the clustering primary key (eg: {some time sequential guid} or long such as 10482723) of the internal User record created to represent the external Person/Service Client.

It is important to note that the more correct term would have been UserIdentityID.

**Aliases & DisplayNames:** User Identities have one or more Aliases. If the system only provides for one, it’s commonly simply referred to as the User’s DisplayName -- but in a current, multi-cultural environment, having a single DisplayName will be inadequate. For example, a Chinese person will have a real Chinese name in Chinese characters, a Chinese name, in westernized characters, an easy for westerners to understand handle “Ben Wu”, and potentially a nickname among peers “Benny” and so forth.   
All are *correct* -- and at least the first 3 are legal names. Displaying which Alias when is handled by system rules (eg: System or nested Group-specific User Preferences).

All to say, A Group, User and Role table -- although the default for most current simple to learn development framework stacks -- is quickly to prove itself as inadequate after a couple of months of real world public use by users and service clients, hence a very expensive development red-herring to avoid.

### Credentials and Tokens

A Person’s Identity has Credentials to prove who they are to the service that is accepting the connection. In the physical world one might do it by voice (“Is that you, mum?”, “Yes Dear”), or by a accepting un-tamperable evidence from a third party (eg: a Driver’s License)

Identity Providers is a misnomer – they don’t provide Identities. They provide trustable Tokens in exchange for your proving your Credentials to it. Hence they should maybe have been called *Identity Confirmation and Token Providers* to more correctly describe what they do.

### Security First

Applying Authorisation to System API endpoints is not well conceptually well understood by developers who have only relied on following framework norms.

Common errors that incur costly bugs and/or rework include the following

#### Where to Enforce Authorisation

Not understanding intuitively where to apply authorisation.

Just like on a house, one applies locks on the outside, stopping them at the front door (the externally accessible rust API operations).

Just like in a house, it is almost always an error to install key locks on interior doors. In services, all operations are invoked by exterior operations, so it is on the API operations that all of the Permissions required to perform the operation and nested operations are checked first.

One *can* check a user’s permissions internally, but just like in a nightclub it a) requires more expense (due to more time consumed checking, affecting responsiveness and free flow, more places to ensure everybody is updated as to new regulations/rules, more bouncers to handle exceptions), b) causes a lot of unnecessary flap throwing people out from nightclubs when it could have been solved so much more elegantly at the front door, not letting them in in the first place.

#### When to Enforce Authorisation

Feeling under pressure to produce something for business stakeholders and project sponsors, it is not uncommon for developers to misunderstand when is the latest time to apply security.

The answer is – you never ever release software that is insecure, unaudited. So you better get cracking on it first.

You wouldn’t walk into a store, no matter how fantastic the sign outside is, if you had any chance that you would be robbed of your identity, right? Same for end user identities. You should never every provide a service they is potentially unsafe to *anyone*, no matter how antsy property developers are. The first responsibility of a developer is to develop a secure service, even if all it says is “Hello Customer!” The good news is that having a firmer understanding of security elements (People, Identities, Users, Duties, Responsibilities, and Permissions) and knowing where to apply them, the work is actually not as difficult to do quickly as it appears at first to intermediate developers.

### Cache First

A side effect of specialisation and the use of cloud services is latency. Although cloud providers have concentrated common services in the same buildings, just the fact of crossing between tiers and device boundaries adds latencies in the thousands of times higher (or multiples therefore) than accessing memory in the same device.

It’s not a good user experience to wait.

Yet Frameworks concentrate on reducing development complexity by concentrating on presentation, leaving as an exercise for the reader to add caching later.

When a PoC is being developed, expected to run only temporarily, used by only a few stakeholders, this makes a lot of sense.

This makes no sense when developing services intended for significant volume user bases. If a server can handle 400 requests per second, and a poorly designed presentation page consumes 40 requests (I’m looking at you, XXX) this translates to 10 simultaneous users per second before there is impact, leading to discussions about horizontal scaling. Since each PaaS service leads adds approximately 250$/month (Q3/2020), along with the initial upfront costs of developing automated or many deployments of multiple servers, it is evident that the use of caching leads to significantly less effort (ie, cost and delays) being needed up front.

It is best practice to apply caching as close to use (ie, nearer the presentation layer as opposed to the source retrieval layer), in the format as close to use as possible (ie, the formatted language specific text, rather than the source record).

It is not uncommon for cache to contain at any one time several different language fragments representing the same record.

It is not uncommon to use only need a few different relatively short time delay amounts to release pressure from database servers (eg: 1 second, 5 seconds, 30 seconds). It is rare for it to be correct to cache data for long durations (eg: 5 minutes or even hours) without it in turn causing secondary operation issues.

### Asynchronous First

When a service thread makes a request across tier boundaries to another device (eg: a database service, a cache service, a externalized workflow process management service), it has to wait for the response. As discussed above, these cross-boundary operations are1000/10,000x slower than in-memory operations. In other words, there’s 1000/10,000 other things that the service could be doing.

Not necessarily for you-- but at least for other customers who are in the request queue to be processed. The solution is to get a ticket and move out of the queue and let others up to the counter. In other words – use Asynchronous development techniques.

Another way of looking at it is that with better architecture, with the same number of devices, one can handle many more customers without paying more rent for space and other services to cloud providers. By itself it won’t be that you can handle 10,000 more customers on a single device – but it is amazing how much can be done (research how few servers, at low cost, StackOverflow uses to maintain their global service).

Frameworks are meant to be easy to be quickly understood by as many developers as possible. In is therefore common for frameworks to not make Asynchronous development patterns mandatory, and leave it up to developers to use the pattern when they are ready.

This is understandable – but it is more important that all team members understand and apply and assist each other with developing asynchronously, to extract the 1000/10,000x benefits.

### Queue First

Queues are a natural outcome of developing useful services.

If you don’t want to deal with learning to process queued operations, you have to work around it by either offering lots of devices (costly), make everybody wait (reputation damaging), or make a service that nobody wants (easy enough).

Again, as Frameworks are meant to be easy on-ramps to developing simple software, it is common that frameworks don’t add Queuing from the start.

Adding Queueing later is practically impossible to be added later without serious unnecessary risk to the service’s integrity due to introducing new bugs, etc.

Not everything takes so long as to need being processed later. Adding queuing doesn’t mean that. Adding Queuing means that you’ve simply added the plumbing to record who got into the line first, recording their request, and then seeing how it plays out.

By recording their position, and request, as they came in, if someone calls Fire! (ie, a server shutdown) and everybody leaves the building and loses their position in the queue, when the building is opened up again, the workers can go through the queued requests, process them in the order received, have the answer waiting for when the doors open again, or simply dispatched to their home address. If there had not been an evacuation, it is probably that the queue’s log of requests would not have been used. If any of the requests would have taken too long (*estimated* wait time >500ms) to complete, then the request would not have been completed right there and then, the person sent home, and they would be Notified later that the task is completed, the response ready (e.g., via a Toast notification, or other).

Queue-First implies that developers have an understanding of the Command Pattern and why it should be used (it’s no use to anyone understanding something conceptually to pass an examination or interview, but then balk at applying it).

### UUID First

Most developers have had the experience of finding out how using random autogenerated UUIDs as PRIMARY KEYs can cause serious impact on performance.

The cause is that the GUIDs are random. The impact from having to insert records all over the table – rather than limiting them to only the tail end of the table – is serious fragmentation of the table, which in turn needs regular lengthly unfragmentation processes that in turn impact responsiveness.

Most developers quickly retreat from this discovery in haste, assuming that the solution is to return to using traditional INT or LONG indexes.

Both INT and LONG type based primary keys have serious flaws. They’re slow in that the table has to be temporarily locked, while the the last key value used is retrieved, incremented, the value used record as the new key, inserted, before the table can be unlocked for the next insert operation.

Secondly, INTs and LONGs cannot be effectively used in distributed scenarios (e.g.: as is the case with most mobile apps that allow records to be added locally, or when using more than one database server, or using one variation or another of geographically dispersed databases).

That’s exactly what Universally Unique Identifiers (UUIDs) were developed for.

But for them to not impact performance, one has to avoid scenarios where a) the table needs to be temporarily locked to get the latest count, increment it, insert it into a record attribute, save it, unlock the table, and b) ensure records are placed at the tail end of the table, in order to not have cause fragmentation.

The solution is therefore a) do not use Sql Server’s random based UUIDs, b) avoid using Sql Server’s UNIQUEIDENTIFIER UUIDs (they’re only slightly better), and instead use a timestamp-based+random suffix, client-side, developed UUID, type 6 or 7 of this RFC (<https://datatracker.ietf.org/doc/html/draft-peabody-dispatch-new-uuid-format>).

### ORM First

It is relatively still common to have to engage in arguments for or against ORMs.

The primary argument for the use of SQL is performance, due to better execution Optimisation Planning.

The argument is only valid when all other maintainability, security, cost considerations are removed from the table.

The lack of market availability of full-stack developers who are not just acceptable, but specialists, in SQL, who can extract the performance, puts projects and organisations at risk.

The cost of maintaining a dedicated SQL specialist, as opposed to full-stack developers, is high, and unwarranted for all projects.

The impediment to speed that ORMs introduce is negligeable – the cause is always the same (poor normalisation, indexing and poor value greedy queries) which needs to be addressed in either case, ORM or no ORM.

The reputational damage that an inadvertent introduction of Sql Injection attack is not worth the speed optimisation that can be obtained.

The advantages that modern ORMs bring are also important. The most important being that modern ORMs seamlessly manage the key pain points of database development over the past 30 years: DB Schemas as Code (CodeFirst in SqlServer) and subsequent schema updates, that are a prerequisite for Automated Deployment.

A second advantage of using ORMs is that they are designed to handle 95% of workloads – so if it’s hard with an ORM it *should* give one time to pause and reconsider if the approach being attempted is exotic for no particular reason, or can be achieved (ie, remain maintainable) by alternate, more run of the mill, approaches.

For example, ORMs help make it evident that one should not be using Natural Keys as Primary Keys (even if this was common practice in the 90’s, and literature still exist on the net proposing that approach).

### Analytics & Reporting First

The development of reports is often left to the end for multiple reasons, including:

* Generating PDF, etc. based reports involve a different technology set than the more often used full-stack technologies
* The results are non-interactive Print based reports, so it’s less engaging a development experience than front-end development.
* Reports are often thought of by many to be external appendages to the service.

Leaving Reporting to the end is an scheduling error that unnecessarily impacts risk and rework later:

* Reports are not external to the service – they are as valid a View of information as any user interface – if not more so -- and are developed in much the same way, with Controllers, Models, Validators.
* They happen to be the *only* Views that Sponsor stakeholders are actually interested in to direct their decision making (Users use the other Views to Browse, Read, Edit, Add & Deprecate records). It makes a lot of political sense to develop first for the person who pays for a project.
* The development of Analytics and Reports dictate the type of data Models required from the system, in turn strongly influencing the Data Schema – so it’s best to know those requirements earlier than later in order to not have to rebuild large swarths of code, introducing new bugs, later.
* With the number of cloud services available, and the fact that modern Browsers include PDF rendering capabilities, it’s not as difficult a technical challenge to solve as it used to be.

## API First

TODO

## Business Last

With so many Firsts defined (Security, Caching, Asynchronous methods, Queuing, ORMs, Reports, APIs, etc.), it’s hard to imagine ever getting around to stakeholder functionality requirements.

One would be right to wonder.

Although there is *always* pressure to show sponsor stakeholders progress on the functionality they desire -- there’s actually always quite a lot of time to meet their expectations.

More importantly – after putting down stable and flexible technical services (as opposed to business services) underneath everything – the work to meet their requirements will go faster AND if there are any changes in the requirements, you’ll be able to pivot faster with less damage to the system.

Another real advantage is that a little more time gives stakeholders time to cogitate on what they’re *actually* asking for, which is time to recognise any logical errors *before* money is spent on building it rather than after. Everybody wins.

The good news is that the logic required to fulfil Business Functionality is usually only a small part of a custom software service. For example, an average mid-size bank app requires services, operations and pretty straightforward models to manage Customers, Suppliers, Resources, Accounts, Invoices, Payments, Refunds (8 key models) -- whereas Systems *-- no matter what business function they meet --* need at least to handle Diagnostics, Exceptions, Configuration, Settings, Requests, Sessions, Operations, Users, Profiles, Terms, Conditions, Permissions, Roles, SearchCriteria, SearchSummaryItem(s), Commands, Queues, Reports, Workflows, Reports, Controllers, Views, etc. (22 and counting).

If a project focuses too early on the discovery, defining, developing testing, delivering, user testing, correcting, rewriting of the business aspects (eg: 8 of them), the project will more often than not run out of budget, resources and time to complete the underlying system aspects (eg: 22+), resulting in the delivery of incomplete, unmaintainable solution.

#### SOLID Development

TODO

#### Command Pattern

TODO

#### Create, Retrieve, Update, ~~Delete~~ Deprecate

### System Design

System Developers are not System Designers, so it is not understood that it is not essential that they know how to lay out system components to improve the chances of delivery – but it is helpful when they understand the objectives of System Designers so that they don’t work at cross-purposes to their Objectives and delivery Responsibilities.

#### Domain Driven Design (DDD)

In general, over 2/3rds of projects with an IT service delivery component fail to deliver on expectations of functionality, cost, resources or schedule.

Simply put, there is always an unacceptably high risk of IT projects failing and therefore there is a duty of care to use any approach which diminishes this ever-present risk.

There is an anecdote somewhere out on the net of an IT consultant at a presentation asking all attendees to raise their hands, then lower them if they would stay on an airplane if told their computer department wrote the plane’s navigation and flying systems. Everybody dropped their hand except one person. When queried as to why he was so confident as to not lower his hand, he answered that if that were the case he could be confident his plane wouldn’t be able to taxi down the runway without failure, let alone take off and fly.

Over the past decade Domain Driven Design (DDD) system design guidance has proven itself at being the way to designing systems with the highest chance of meeting functionality and quality expectations within budget, resources and schedules.

Without going into DDD in great detail there are a couple of key development specific points to know.

* Objects have methods, but as per OO principles, they only manipulate their *own* state. You use Services to provide methods to invoke the methods of Objects.
* Services are stateless singleton instances of closely related functionality.
* There are 3 key groupings of Services: Application Services, Business Services and Infrastructure Services.
* Presentation/API Controllers are \*thin\* and in turn invoke \*thin\* Application Services which orchestrate calls to Infrastructure and Business Services.
* Infrastructure Services provide functionality to manage infrastructure (storage, caching, etc.).
* Business Services never ever invoke Infrastructure Services directly, and vice versa. It’s up to the Application Service to know how to orchestrate calls between them so that this cross-talk/cross-contamination never occurs.
* The 3 separate Services do share common internal Models (ie Objects).
* There are internal Models (with Methods as necessary) and external Models (DTOs, for communication via APIs, and without Methods).

The above short list should be enough to understand why a system designer will generally prepare the assemblies/dlls/packages used in a service into 5 basic groups, and expect developers to know where to put new code.

#### Event Driven Design

EDA and PubSub based architecture are examples of uncoupled Event driven design.

PubSub is appropriate to develop low coupling within a single system.

MicroServices are an appropriate to communicate between independent services managed by different deployment teams/projects/groups.

TODO

#### MicroServices (Dead Last)

Avoid using in a single system (it’s a solution for bridging the gap left by disparate department delivery objectives – it is not a solution for unnecessarily breaking a single department’s service into microservices).

TODO

### Interface Design

#### Context, Status, Navigation, Current

TODO

#### Browse, Read, Edit, Add, Deprecate (BREAD) Interface flows

TODO

# Appendices

## Terms

### Relational Database Index Terms

A Key and an Index are synonyms.

An Key is used to quickly find a record. Note that an indexed column (a Key) is not necessarily unique (eg: “DOB”).

A UNIQUE KEY is a special case of an Index column/key that is guaranteed to be unique (eg: “NationalStudentNumber”).

A PRIMARY KEY is a special case of UNIQUE KEYs that does not allow a NULL.

A FOREIGN KEY is a column of group of columns in a relational database table that contains the value(s) of another table’s Primary Keys, in order to provide a unique link between two table’s records.

Natural Keys are real world identifiers, such as National Health or Student Numbers, used as Primary Keys.

Under no circumstances is it best practice to use Natural Keys as Primary Keys. Only use them as UNIQUE KEYs.

A Clustered Key is a PRIMARY KEY used to tell Sql Server how to organise the records in a database.