Selected Architecture Problems:

#1 #SO 77615623

Architecture Problem:

I'm currently developing 2 different multiplayer games on Unity where the user has to register (and subsequently log in) to be able to play.

The way I would like it to work would be to only have a single account, thus being able to register only once and then play both games with the same account (think Battle.net for WoW, HS, ...).

Of course, each of the 2 games has its own specific items, coins, etc. that would be stored in their database. What I need is a good way to lay down the DB architecture (I'm using SQL).

I figured that could use a single database in the back-end with a table for authentication and then multiple tables for each game, but I see some problems with this:

The single DB would look already confusing with 2 games, let alone if I want to add more games in the future.
If one game was to gain popularity, I couldn't scale its part of the DB on its own, but would need to scale the whole DB
Another idea would be to have a single DB dedicated to authentication (with the users' accounts) and then one DB per game with objects, coins, etc. This would of course make the architecture more complicated.

Do my concerns make sense? Is one of those 2 ideas the way things are made in the industry? Are there better ways? What are those and what would be a way to implement them correctly?

Architecture Decision:

If each game requires registration, and uses the same registration, then the registration should be separate from the games.

Design Rationale (Human Expert):

This prevents the two games from duplicating registration functionality. Since registration is logically separate, the data should be separate as well, meaning that you have an authentication service that stores player identities and an ID. Then in the games, store this

ID for each player. Players authenticate against your service with their credentials and receive a JWT encoding their player ID.
This is so separate from the games that you can just buy it off the shelf. You can maximize your chances of making a successful game by allowing your game devs to concentrate on game dev.

Design Rationale (Llama3 LLM-based Agent):

Based on the analysis from various aspects, I would argue that the chosen architectural decision to separate the registration process from individual games is a sound one.

From the perspective of Data Isolation, the design decision ensures that each game's data is isolated and contained within its own domain, preventing data corruption or interference. This approach also allows for better scalability, as each game's database can be scaled independently without affecting the overall system.

From the perspective of Authentication Centralization, the design decision enables centralized authentication management across all games, reducing complexity and making it easier to manage. The registration process becomes a standalone service that handles user account creation and login functionality, independent of individual games.

From the perspective of Game-Specific Data Management, the design decision allows for individual management of game-specific data without affecting the overall system. Each game has its own dedicated space in the system to manage its specific data, which is essential for maintaining data integrity and security.

From the perspective of Complexity Management, the design decision minimizes complexity by decoupling concerns, reducing coupling between games and the registration system, improving scalability, and enhancing maintainability.

Finally, from the perspective of Maintenance and Updates, the design decision enables independent maintenance and updates for each game's database, making it easier to troubleshoot and fix issues. This modular approach also allows for deploying updates or patches for one game without affecting others, reducing the risk of downtime or data corruption.

In weighing the tradeoffs, I would argue that the benefits of separating the registration process from individual games outweigh the potential drawbacks. The design decision:

- Ensures data isolation and prevents data corruption or interference
- Enables centralized authentication management across all games
- Allows for individual management of game-specific data without affecting the overall system
- Minimizes complexity by decoupling concerns, reducing coupling, improving scalability, and enhancing maintainability
- Enables independent maintenance and updates for each game's database

Overall, I believe that the chosen architectural decision to separate the registration process from individual games is a sound one that balances the needs of data isolation, authentication centralization, game-specific data management, complexity management, and maintenance and updates.

#2 #SO 77476674

Architecture Problem:

I am trying to implement clean architecture in my blazor server project. In my application layer I have this service.

```
public class GetOrganizationDetailQuery : IGetOrganizationDetailQuery
{
   private readonly IDatabaseService _database;

   public GetOrganizationDetailQuery(IDatabaseService database)
   {
      _database = database;
   }

   public OrganizationDetailModel Execute(Guid id)
   {
      var organization = _database.Organizations
                  .Where(x => x.Id == id)
                  .Select(x => new OrganizationDetailModel()
                  {
                     Id = x.Id,
                     Name = x.Name
                  })
                  .Single();
      return organization;
   }
}
```

The interface is also defined in the application project.

```
public interface IDatabaseService
{
   DbSet<Domain.Organization> Organizations { get; }

   void Save();
}
```

My data layer relies on the application layer, and it defines an implementation of the above interface.

```
public class DatabaseService : DbContext, IDatabaseService
{
   ...

   public DbSet<Organization> Organizations { get; set; }
```

```
    public void Save()
    {
        this.SaveChanges();
    }
}
```

Finally in my UI layer I try to inject the query so I can use it on a page.

Program.cs

builder.Services.AddScoped<IGetOrganizationDetailQuery, GetOrganizationDetailQuery>();
This throws an exception; the service cannot be constructed.

Is there any way to fix this issue without adding a reference to the data layer from my UI layer?

Architecture Decision:

Ultimately you need to reference all projects where interfaces and implementations are defined to be able to use dependency injection. My preference is to move all dependency registration into a dedicated project.

Design Rationale (Human Expert):

This (in my opinion) keeps the app cleaner and means that I can avoid referencing implementation projects (e.g. Entity Framework implementations).
The downside with this approach is that you need to be able to abstract the dependency container itself, but this (again, my opinion) is a worthwhile trade-off.

Design Rationale (Gemini CoT):

Step 1: Understand the Architecture Design Decision

The architecture design decision is to move all dependency registration into a dedicated project. This means that the UI layer will not have a reference to the data layer, and the data layer will not have a reference to the application layer.

Step 2: Advantages of Choosing this Architecture Design Decision

Decoupling: This architecture design decision helps to decouple the different layers of the application. The UI layer will not be dependent on the data layer, and the data layer will not be dependent on the application layer. This makes it easier to maintain and change the application in the future.

Testability: This architecture design decision makes it easier to test the different layers of the application. The UI layer can be tested without the need to mock the data layer, and the data layer can be tested without the need to mock the application layer.
Reusability: This architecture design decision makes it easier to reuse the different layers of the application. The UI layer can be used in other applications, and the data layer can be used in other applications.

Step 3: Disadvantages of Choosing this Architecture Design Decision

Complexity: This architecture design decision can add some complexity to the application. It requires a separate project to be created for dependency registration, and it can be more difficult to track down dependencies when they are spread across multiple projects.
Performance: This architecture design decision can introduce some performance overhead. When a dependency is injected, the application will need to load the assembly that contains the dependency. This can add some latency to the application.

Step 4: Rationale for Choosing this Architecture Design Decision

The advantages of choosing this architecture design decision outweigh the disadvantages. Decoupling, testability, and reusability are all important factors to consider when designing an application. The performance overhead is minimal, and the complexity can be managed with proper planning.

Overall, this architecture design decision is a good choice for applications that are complex, have multiple layers, and are expected to be reused.

#3 #SO 77246412

Architecture Problem:

I'm developing an Android application using Jetpack Compose, and I'm facing a design decision regarding handling different user types, such as admin and user, within the app.

Architecture Decision:

Currently, my application follows the single activity architecture.create a new activity, let's call it "ClientActivity," to separate the logic and UI components for different user types. This would mean that when a user logs in, they would be directed to either the main activity or the client activity based on their user type.

Design Rationale (Human Expert):

While it is possible to solve all those problems, handling it in a single activity where you have a clear admin state as a LiveData that allows you to quickly update the UI as needed, sounds like a much simpler approach to me.

Design Rationale (Mistral LLM-based Agent):

Based on the analysis provided, the choice of using a single activity architecture in an Android application developed using Jetpack Compose comes with several trade-offs and rationales.

Firstly, from the perspective of Modularity and Simplicity, centralizing user authentication and authorization logic in a single activity architecture allows for a more modular approach to handling different user types while simplifying the application structure. This approach eliminates the need to update the Manifest file every time and reduces boilerplate code by removing the requirement for declaring startActivityForResult methods. Additionally, using fragments within the single activity can further increase modularity by dividing the UI into discrete chunks and improving reusability.

Secondly, from the perspective of Performance, having a single activity architecture reduces the overhead of creating and destroying multiple activities, resulting in improved performance and an enhanced user experience. This improvement comes from eliminating the need to start new activities, which can save time and resources.

Thirdly, from the Security standpoint, centralizing user authentication and authorization in one place enhances security by ensuring that all access control checks are consistent across the entire application. In a single activity architecture, the entry point of the application is the main activity, making it a logical place to implement security measures such as user authentication and authorization.

Fourthly, from the Scalability perspective, a single activity architecture allows for easier scaling as new features or user types are added to the application. All components are already integrated into a unified structure within the same activity, making it simpler to manage and modify as compared to having multiple activities or fragments. Additionally, since all components share the same lifecycle, managing state and data flow becomes more streamlined, further contributing to scalability.

Lastly, from the Testing perspective, having a single activity architecture simplifies testing by eliminating the need to test multiple activities and their interactions. This makes it easier to focus on individual components and their functionality. Additionally, popular tools and frameworks like Android Studio, JUnit, Espresso, and Debug are designed with this architecture in mind, making testing more effective and efficient.

In conclusion, based on the trade-offs and rationales discussed above, the choice of using a single activity architecture in an Android application developed using Jetpack Compose is a reasonable decision that offers several advantages such as modularity, simplicity, performance, security, scalability, and testing efficiency. However, it's important to note that every project is unique, and this architectural decision may not be suitable for all use cases. Therefore, careful consideration should be given to the specific requirements of each project before making a final decision.

#4 #Discussion 2090

Architecture Problem:

We are currently evaluating prisma and are discussing possible architectures for a solution.

The current approach is based von Next.JS, Prisma and MSSQL Server and involves the need that users can connect to different databases by configuration at runtime. So a user might have access to multiple databases by permission provided from a different backend and can choose at runtime to which db she wants to connect to. For now I'd assume that schema is structure-wise identical for all. So we wouldn't need to generate and maintain multiple prisma schemas - at least not for this. Potential user base is some 100s up to 1,000 of users, for concurrent users I wouldn't assume more than 50 (rather less).

So we are wondering if there is a way to implement this using prisma.

Architecture Decision:

Approach b): maintain a map or comparable structure of prisma client instances in the nextjs app / node server instance in memory, one for each db / schema instance requested? That can obviously result in numerous prisma client instances based on the number of dbs/schemas requested. And would infer some complexity because we'd have to take care of purging the list when clients are not used for a certain amout of time. etc. The risks which come to mind are of course memory issues when there are too many prisma client instances etc., and also connection pooling issues on db side

Design Rationale (Human Expert):

You definitely want to go with b) here. Creating a database connection is costly, and you can not easily switch between databases on runtime. So I would suggest you keep a list of Prisma Clients and make sure to only use the correct one for the correct user. Prisma Client is not optimized for this use case, so it will use a bunch of memory for each instance - so you might need to manage the clients you have open. Usually that goes fine though and turns out not to be that big of a problem.

Design Rationale (GPT-4 LLM-based Agent):

The architectural decision to maintain a map or comparable structure of Prisma client instances in the Next.js app / node server instance in memory, one for each db / schema instance requested, presents several trade-offs that need to be considered.

From a scalability perspective, this approach is beneficial as it allows the system to handle an increasing number of databases and users without performance degradation. It provides a scalable solution to the problem of connecting to multiple databases at runtime. However, it also introduces complexity in terms of managing and purging the list of client instances when

they are not in use, which could potentially lead to memory and connection pooling issues if not handled properly.

From a memory management perspective, this approach ensures efficient use of memory resources while maintaining the flexibility of allowing users to connect to different databases at runtime. However, it introduces the complexity of managing potentially numerous Prisma client instances, which could lead to memory issues if not properly handled.

From a connection pooling perspective, this approach aligns with the concept of connection pooling, which aims to minimize the overhead associated with establishing new connections for each database interaction. However, having too many Prisma client instances can lead to memory issues and can exhaust the database connection limit.

From a runtime configuration perspective, this approach provides a flexible and dynamic system that can adapt to the varying needs of the users. However, it also acknowledges the potential risks and complexities associated with this approach, such as managing numerous Prisma client instances which can lead to memory issues and connection pooling issues on the database side.

From a security perspective, this approach allows for a robust and flexible control of user access to multiple databases. However, it also introduces potential risks, such as memory issues due to too many Prisma client instances and connection pooling issues on the database side.

From a maintenance complexity perspective, this approach could potentially increase the complexity of the system due to the need to manage and purge unused client instances. However, it could also potentially improve the efficiency of the system by allowing users to connect to different databases by configuration at runtime.

In conclusion, while this architectural decision presents several benefits such as scalability, efficient memory management, connection pooling, runtime configuration, and security, it also introduces potential risks and complexities such as memory issues, connection pooling issues, and increased maintenance complexity. Therefore, the decision to adopt this approach should be based on a careful evaluation of these trade-offs, taking into consideration the specific requirements and constraints of the system, as well as the capacity of the team to manage the associated complexities and risks.

#5 #Issue 270

Architecture Problem:

openbao/openbao
I feel that Vault's design has become trapped in a cocoon of its own making.
This includes investing significantly in features like Raft/replication and transactions, which, in my opinion, add an unnecessary burden.

Architecture Decision:

I suggest pushing Vault towards functioning mainly as a controller layer, where each node can handle reads and all nodes can write, assuming the database supports transactions.

Design Rationale (Human Expert):

It's clear that there are numerous databases on the market that already support these features, like spanner, foundationDB. I wish it would rely more on databases for durability, reliability, and replication, rather than on Vault itself.

Design Rationale (GPT-4 CoT):

Step1: Understanding the Architecture Design Decision based on the provided Architecture Problem.
The architecture problem suggests that the current design of Vault is overly complex due to the inclusion of features like Raft/replication and transactions. The proposed design decision is to simplify Vault's role to mainly function as a controller layer, where each node can handle reads and all nodes can write, assuming the database supports transactions.

Step2: Advantages of choosing this Architecture Design Decision.
The main advantage of this design decision is simplicity. By reducing Vault's role to a controller layer, the architecture becomes easier to understand, maintain, and scale. This approach also allows for better distribution of workload among nodes, as each node can handle reads and all nodes can write. This can potentially improve the system's performance and reliability. Furthermore, by assuming that the database supports transactions, the system can leverage the database's built-in mechanisms for ensuring data consistency and integrity, reducing the need for custom solutions.

Step3: Disadvantages of choosing this Architecture Design Decision.
The main disadvantage of this design decision is that it relies heavily on the database's support for transactions. If the database does not support transactions, or if its support is limited or unreliable, this could lead to data inconsistencies and other issues. Additionally, by simplifying Vault's role, some features or capabilities may be lost. For example, the use of Raft/replication can provide fault tolerance and high availability, which may not be as robust if Vault is only used as a controller layer.

Step4: Rationale for choosing this Architecture Design Decision.
Considering the advantages and disadvantages, the rationale for choosing this design decision is based on the need for a simpler, more scalable, and more maintainable architecture. While there are potential risks associated with relying on the database's support for transactions, these can be mitigated by choosing a reliable database system and implementing appropriate error handling and recovery mechanisms. The potential loss of features or capabilities can also be addressed by carefully evaluating the system's requirements and ensuring that the simplified architecture can still meet these requirements. Overall, the benefits of a simpler and more scalable architecture outweigh the potential drawbacks, making this a sound design decision.