Architectural

Monolith

A monolithic architecture is an architectural pattern where all components of an application are combined into a single codebase and deployed as a single unit. In a monolithic design pattern:

1. **Codebase**: All the application's functionalities, including the user interface, business logic, and data access code, are packaged together into a single codebase.
2. **Deployment**: The entire application is deployed as a single unit, typically on a single server or a set of servers.
3. **Communication**: Communication between different components of the application typically occurs through function calls or method invocations within the same process.
4. **Scalability**: Scaling a monolithic application involves scaling the entire application, which can be challenging as it requires replicating the entire application stack.

Monolithic architectures are often simpler to develop and deploy initially, especially for small to medium-sized applications. They offer advantages such as ease of development, debugging, and testing, as well as simpler deployment and management.

However, as applications grow in complexity and scale, monolithic architectures can become harder to maintain and scale. They may suffer from issues such as tight coupling between components, limited scalability, and difficulty in adopting new technologies or frameworks.

In contrast, microservices architecture, where an application is decomposed into smaller, independently deployable services, offers benefits such as improved scalability, flexibility, and maintainability. However, microservices come with their own set of challenges, such as increased complexity in managing distributed systems and communication overhead between services.

Ultimately, the choice between monolithic and microservices architectures depends on factors such as the size and complexity of the application, scalability requirements, development team expertise, and organizational constraints.

Model View Controller

The MVC architecture is very broad and can change depending on the programming language and type of application you are doing, so in this case, yes your approach can be accepted as correct.

What I have learned from static typed languages is that you define the model and views as complete separate entities, and the controller takes an instance of both model and views as parameters.

What you need to ask yourself to define if your app is MVC is the following:

- If I change something in the view do I break anything in the model?
- If I change something in the model do I break anything in the view?
- Is the controller communicating everything in both view and model so that they don't have to communicate with each other?

If nothing breaks and the controller does all of the communication then yes, your application is MVC.

You might want to look into design patterns such as Singleton, Factory and others that all use the MVC architecture and define ways to implement it.

Microservices

Others

Event-Driven Architecture (EDA) and Service-Oriented Architecture (SOA) are both architectural patterns used in software design, but they have different approaches and focus areas. Here's a comparison between the two:

1. **Communication Paradigm**:
   - **EDA**: In Event-Driven Architecture, communication between components is based on events. Components publish events when certain actions or changes occur, and other components subscribe to these events and react accordingly. This asynchronous communication model allows for loose coupling between components and enables real-time responsiveness.
   - **SOA**: In Service-Oriented Architecture, communication between components is typically based on service interfaces. Services expose their functionality through well-defined interfaces (APIs) that other services or clients can invoke. Communication in SOA is often synchronous, although asynchronous messaging can also be used.
2. **Granularity**:
   - **EDA**: EDA tends to be more fine-grained, with events representing specific actions or changes within the system. Components can react to individual events and perform specific actions accordingly.
   - **SOA**: SOA can vary in granularity, but services tend to encapsulate larger units of functionality. Services are typically designed to represent business capabilities or domain entities, providing coarse-grained operations.

3. **Flexibility**:
   - **EDA**: EDA offers greater flexibility and agility, as components can react to events in a dynamic and decentralized manner. New components can be added or existing components modified without affecting the overall system.
   - **SOA**: SOA also provides flexibility, but to a lesser extent compared to EDA. Changes to services may require coordination between service providers and consumers, and service contracts need to be carefully managed to ensure compatibility.
4. **Scalability**:
   - **EDA**: EDA inherently supports scalability, as components can be scaled independently based on event processing requirements. Event-driven systems can handle bursts of traffic more gracefully by distributing processing across multiple instances.
   - **SOA**: SOA can be scalable, but scaling individual services may not always be straightforward, especially if services have dependencies or shared resources.
5. **Complexity**:
   - **EDA**: EDA can introduce complexity, particularly in managing event propagation, event schemas, and ensuring event consistency across components. Event-driven systems may also require additional infrastructure for event processing and management.
   - **SOA**: SOA tends to be less complex compared to EDA, as services have well-defined interfaces and interactions. However, managing service contracts, versioning, and service discovery can still introduce complexity, especially in large-scale deployments.

In summary, Event-Driven Architecture is well-suited for scenarios where real-time responsiveness, loose coupling, and flexibility are paramount, while Service-Oriented Architecture provides a more structured approach to building distributed systems with reusable and interoperable services. The choice between EDA and SOA depends on the specific requirements, constraints, and trade-offs of the application or system being designed.

Revision #8
Created 14 February 2024 22:57:28 by victor
Updated 26 February 2024 01:06:04 by victor