Building Microservices Introduction

In software development, we often start with a monolithic approach - one large application that handles everything. However, as applications grow, we can face challenges with scaling, maintenance, and deployment. This is where microservices come in.

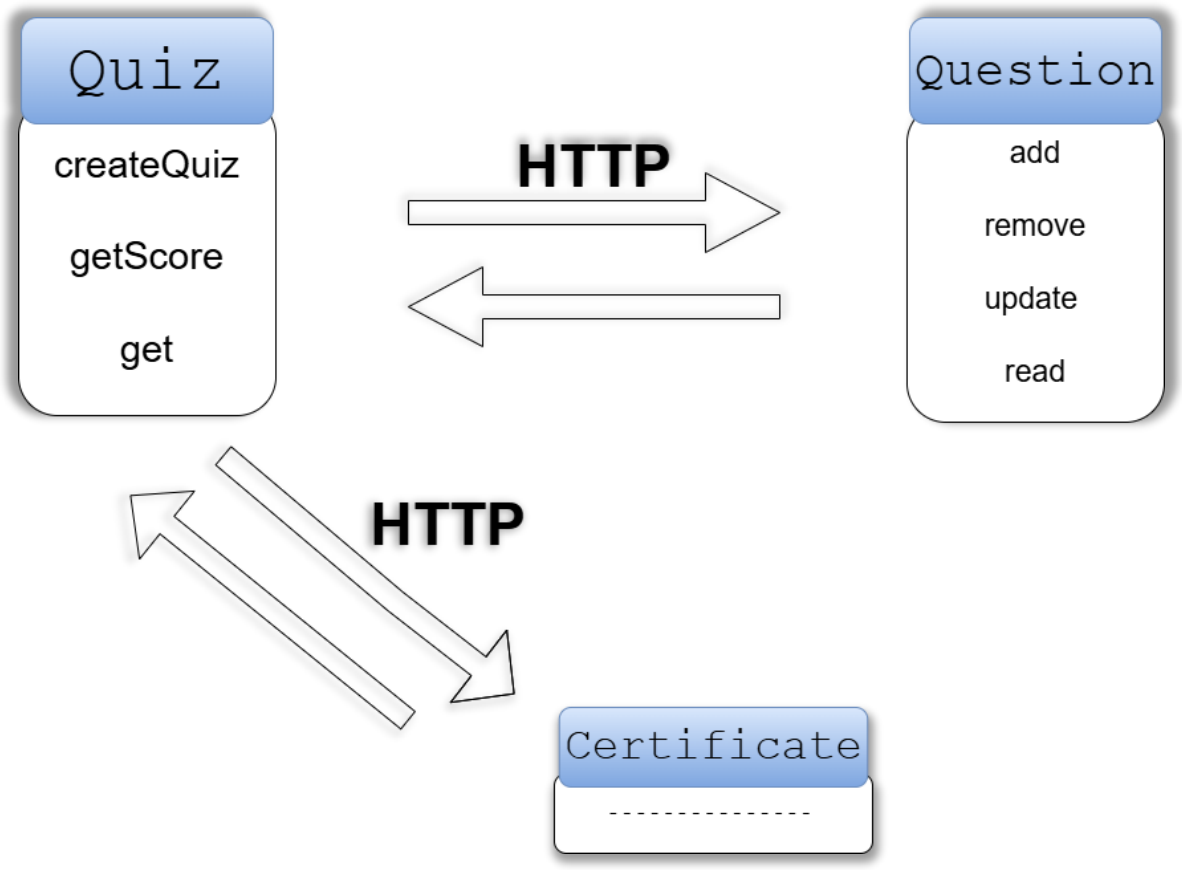
Let's understand this transition using our Quiz application example:

👉 Monolithic Quiz Application

- One big application handling all functionality
- Contains both **QuestionController** and **QuizController** in the same codebase
- Everything tightly coupled together

👉 Microservices Approach

- Split the application into smaller, independent services
- Each service runs independently and has its own responsibility
- Services communicate with each other when needed



👉 Separating Services

➢ In our Quiz application, we'll separate it into two microservices:

- **Question Service:** Manages all question-related operations
- **Quiz Service:** Handles quiz creation and management

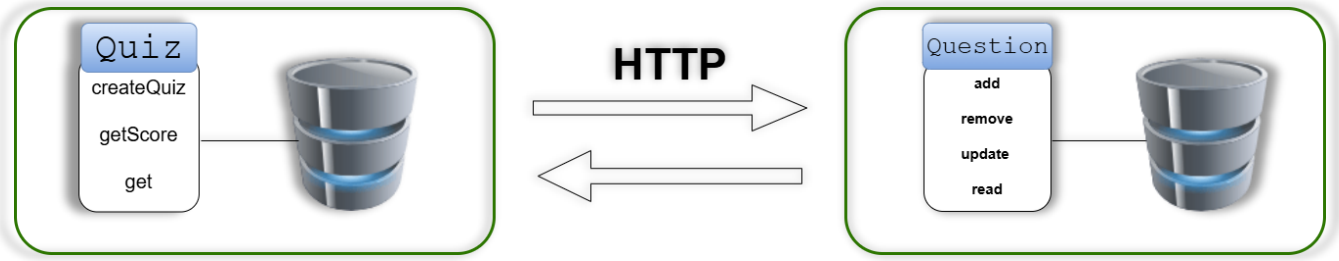
➢ This separation allows each service to:

- Focus on its specific functionality
- Have its own database
- Scale independently
- Be developed and deployed separately

👉 How Services Communicate

In our monolithic Quiz application, creating a quiz involved directly accessing the QuestionDao to fetch random questions. But with microservices:

- Quiz Service no longer has direct access to QuestionDao
- Quiz Service must make a network request to Question Service
- Each service has its own database
- Communication happens over the network through APIs

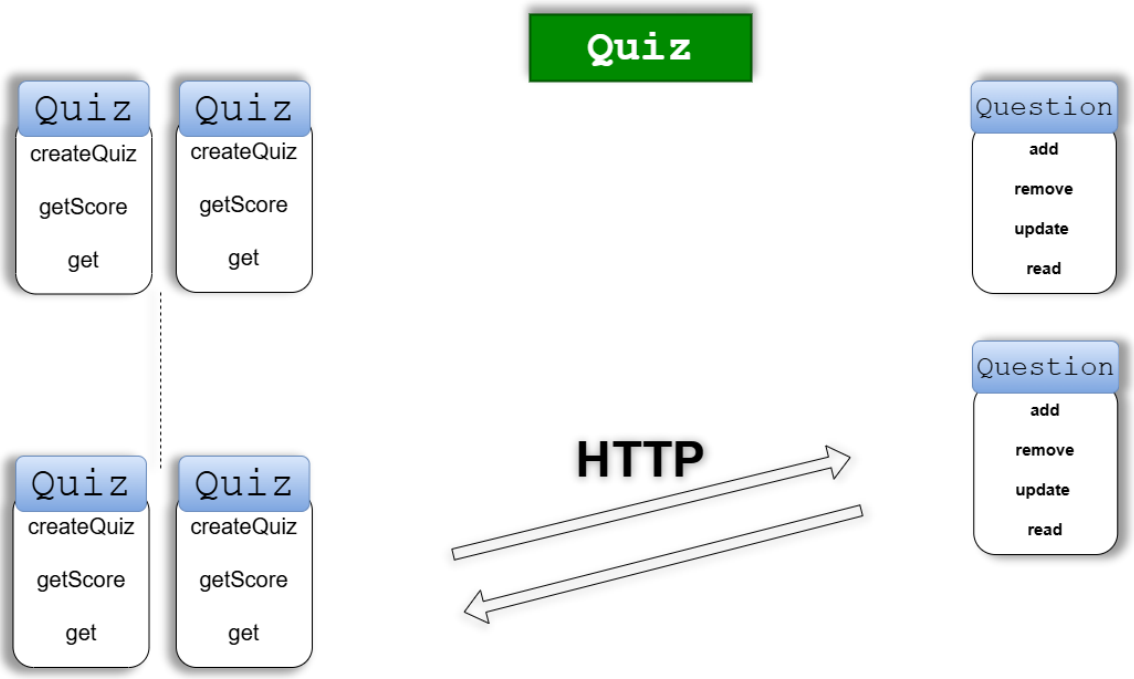


👉 Challenges in Microservices

➢ **Scaling and Discovery**

When we separate our application into microservices, new challenges emerge:

- **Scaling:** We can now scale services independently based on demand (unlike monoliths where we scale the entire application)
- **Multiple Instances:** We might have multiple instances of each service with different IP addresses
- **Service Discovery:** How does one service find another?
- **Load Balancing:** How do we distribute traffic across multiple instances of the same service?



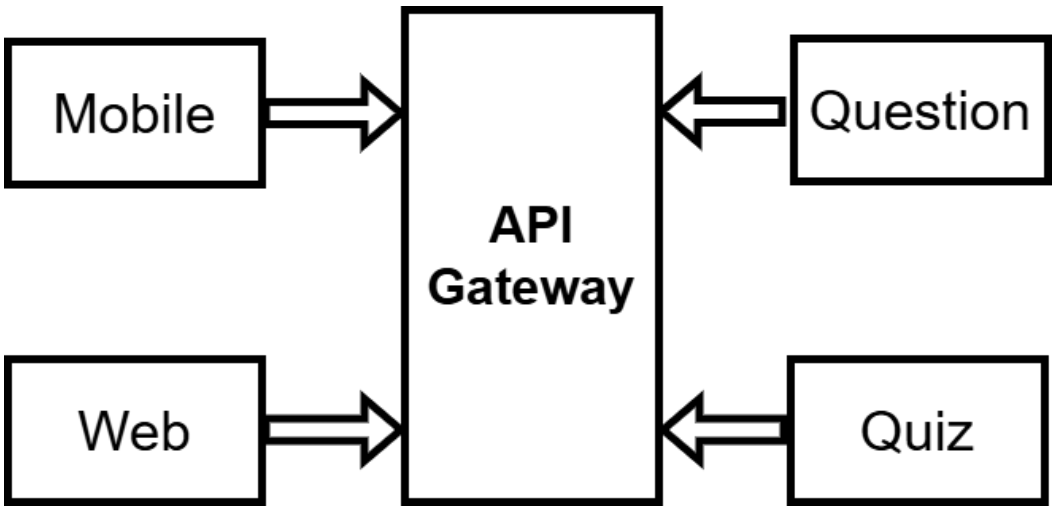
👉 Client Access and API Gateway

Another challenge is how clients interact with our services:

- Clients shouldn't need to know about all the different services and their URLs
- Clients shouldn't have to remember multiple endpoints

Solution: API Gateway

- Acts as a single entry point for all client requests
- Routes requests to the appropriate service
- Simplifies the client experience



👉 Service Registry

When we have many services (possibly 10 or more):

- Services need to locate each other
- Remembering IP addresses isn't practical
- Services constantly change (new instances added, old ones removed)

➢ **Solution: Service Registry**

- Services register themselves when they start up
- Services can look up other services by name
- No need to hardcode IP addresses

👉 Fault Tolerance

What happens when a service fails?

For example, if the Quiz Service depends on the Question Service, and the Question Service is down:

- We don't want Quiz Service to hang indefinitely
- We don't want users to wait without any response

➢ **Solutions:**

- **Circuit Breakers:** Detect failures and prevent cascading failures
- **Fail Fast:** Quickly identify when a service is unavailable
- **Fallbacks:** Provide alternative responses when a service is down
- **Proper Error Handling:** Communicate issues to clients appropriately

👉 Conclusion

Building individual microservices is relatively straightforward, but connecting them and handling all these challenges requires careful planning and implementation. The benefits of microservices include:

- Independent scaling
- Technology diversity (use the right tool for each service)
- Faster deployment
- Improved fault isolation
- Better team organization (teams can focus on specific services)

However, microservices also introduce complexity in:

- Service communication
- Service discovery
- Load balancing
- Fault tolerance
- Monitoring and tracing