Building Microservices with .NET 9 Web API

Part 1: Conceptual Overview

Microservice Architecture

Microservices break an application into many small, independent services, each responsible for a specific business capability. In practice this means **loosely coupled, independently deployable services** rather than one large monolithcloud.google.commicroservices.io. Each service owns its own code, runtime, and **private data store**, enabling small autonomous teams to develop and deploy it independentlycloud.google.comlearn.microsoft.com. This architecture improves development velocity and scalability, but introduces distributed system challenges (network faults, data consistency, testing, etc.). A typical microservices application has many services communicating over well-defined interfaces, often with an **API Gateway** as the single entry point for clientsmicroservices.io.

Service Decomposition for E-Commerce

In an e-commerce domain, we decompose by business capabilities or bounded contexts (e.g. via Domain-Driven Design). For example, we might implement separate services for **Catalog** (product listings), **Basket/Cart** (customer shopping cart), **Ordering** (processing orders), **Payment**, **Shipping**, and **Identity/Authentication**. Each service handles its own data and logic. The Microsoft eShopOnContainers reference implementation uses exactly such services (Catalog, Basket, Ordering, Identity, etc.) to illustrate a .NET microservices systemgithub.com. For instance, *CatalogService* manages product data and inventory, *BasketService* manages user carts (often using Redis or a fast store), and *OrderService* handles order placement and fulfillment. This decomposition allows each service to use the best data storage and tech for its needs, and to be deployed and scaled separately.

Inter-Service Communication (Sync vs Async)

Services must communicate. For **synchronous** request/response calls, common choices are REST over HTTP or gRPC. REST/JSON is ubiquitous and easy for external clients, but uses HTTP/1.1 and has fixed HTTP verbs. gRPC (using HTTP/2 and Protobuf) offers high-performance, strongly-typed RPC methods and streaming; it's especially suited for high-throughput internal RPCs. For **asynchronous** decoupled communication, we use message/event brokers like RabbitMQ or Apache Kafka. **A message broker lets one service publish an event without knowing which services will consume it.** In short: for 1:1 real-time calls use REST/gRPC; for **pub/sub or event-driven** patterns use RabbitMQ/Kafkastackoverflow.com. **For example, the BasketService might publish a "CheckoutRequested" event to RabbitMQ, which the OrderService and InventoryService consume independently. (Kafka is similar but adds durable logs and replay; use Kafka when you need a persistent event log and high throughput.)**

> **Choosing a communication style:** "If you need a synchronous response on a 1:1 service call, use gRPC. If you don't care which service will consume a message (asynchronous decoupling), use RabbitMQ. If you need a distributed log of events for later reuse, use Kafka."stackoverflow.com.

Data Management and Polyglot Persistence

Each microservice **manages its own data** – a core principle to avoid couplinglearn.microsoft.com. Services should never share a database schema directly. This leads naturally to *polyglot persistence*: each service can choose the database type that best fits its needslearn.microsoft.com. For example, the CatalogService might use a relational PostgreSQL (for complex queries and consistency), the BasketService might use Redis (fast in-memory store for cart sessions), and the OrderService might use a document store like MongoDB (flexible order schemas, easy to scale). Because services are independent, they can use different database technologies or versions without impacting others. (Physically you could even use separate schemas on one DB server, but logically isolate each service's data.) The tradeoff is complexity: data consistency across services must be handled via events or sagas instead of ACID transactions.

Containerization: Docker & Docker Compose

Containers are the de facto way to package microservices. **Docker** lets you package each service (and its runtime and dependencies) into an image that runs identically anywhere. An ASP.NET Core Web API service image includes the .NET runtime and your compiled code. Using Docker ensures environment consistency and portability. **Docker Compose** then orchestrates multi-container setups on a single host. With a `docker-compose.yml`, you can define all services (API container, databases, message broker, etc.) and start them together. As Docker's docs note, Compose "simplifies the control of your entire application stack" so developers can bring up the full system with one commanddocs.docker.com. In production, one often moves to Kubernetes or another orchestrator, but Compose is ideal for development and testing multiple microservices together.

API Gateway (Ocelot/YARP)

In a microservices system, clients shouldn't call each service directly (that would expose too many endpoints and coupling). Instead we use an **API Gateway** as the single public entrypoint. The gateway receives client HTTP requests and routes them to the appropriate service. It can also handle cross-cutting concerns like authentication, rate limiting, and routing transformations. In .NET environments, popular gateways include **Ocelot** (an open-source .NET API Gateway library) and **YARP** (Microsoft's "Yet Another Reverse Proxy" project). An API Gateway effectively acts as a specialized reverse proxymilanjovanovic.tech. For example, you might configure the gateway so that `/catalog/api/products` calls the CatalogService, while `/basket/api/items` calls the BasketService. The gateway hides the internal service URLs and ports from the client. In the Microsoft eShop reference, Ocelot was used as the gateway, but newer designs may use YARP (which is high-performance and configurable)milanjovanovic.tech.

Authentication and Identity (IdentityServer/Duende)

Security is critical. A common pattern is to centralize user auth in an **Identity microservice** that issues tokens (JWTs) to authenticated clients or services. In .NET, IdentityServer (now Duende IdentityServer) is a standard solution. Duende IdentityServer is “a highly extensible, standards-compliant framework for implementing the OpenID Connect and OAuth 2.x protocols in ASP.NET Core”docs.duendesoftware.com. You would configure it with your clients (e.g. a client ID/secret for each microservice or SPA), identity resources, and API scopes. For example, the IdentityService might use ASP.NET Core Identity for user management and then issue JWTs to clients. Other services (Catalog, Basket, etc.) would validate incoming JWTs (checking issuer and audience) on each request. Integration steps typically include adding `services.AddIdentityServer()` and configuring JWT Bearer authentication in the ASP.NET Core pipeline.

Observability & Logging

Monitoring a distributed system is essential. We collect metrics, logs, and traces across services. **Metrics:** use Prometheus to scrape metrics from each service. ASP.NET Core can be instrumented (e.g. via OpenTelemetry) to expose performance metrics (request rates, latencies, custom counters). Grafana can then visualize those metrics in dashboards. **Distributed Tracing:** tools like Jaeger or Zipkin (via OpenTelemetry) can trace a request across services. **Logging:** use structured logging (for example, Serilog). Serilog “provides diagnostic logging to files, the console, and elsewhere” with fully structured event dataserilog.net. You can sink logs to a system like ELK (Elasticsearch) or Loki for centralized log searching. Each service writes logs (with context and trace IDs), and an external system aggregates them. Overall, the goal is full **observability**: being able to monitor service health (e.g. with Health Checks), inspect logs, and trace issues end-to-end.

CI/CD (GitHub Actions)

Continuous integration and deployment pipelines automate building, testing, and deploying microservices. A typical GitHub Actions workflow for our .NET solution would: checkout code, install .NET 9, run `dotnet restore`, `dotnet build`, `dotnet test`, and then build Docker images for each service. You might then push images to a container registry and deploy (e.g. to Kubernetes or Docker Compose on a server). GitHub Actions has official actions like `actions/setup-dotnet` for .NET and `docker/build-push-action` for containers. Automating this ensures that every commit is validated and a new version of each microservice can be deployed seamlessly.

Resilience Patterns (Circuit Breaker, Retry, Health Checks, Service Discovery)

Building a robust microservices system requires resilience. Key patterns include **Retries** and **Circuit Breakers**: for example, using the Polly library in .NET. Polly lets you define policies such as retry with exponential backoff and circuit breaker thresholdslearn.microsoft.com. For instance, an HTTP call from Service A to Service B can be wrapped with a retry policy and a circuit breaker so that transient errors are retried and persistent failures open the circuit. **Health Checks:** ASP.NET Core has built-in health check middlewarelearn.microsoft.com. You configure health endpoints (e.g. `/health`) that check database connectivity, external dependencies, etc. Orchestrators and load balancers can poll these endpoints to ensure a service instance is healthy. **Service Discovery:** in more dynamic environments (like Kubernetes), services discover each other via DNS or a discovery service (e.g. Consul, Eureka). In simpler setups, Compose or orchestration can link services by name. Finally, **Observability** itself (as discussed above) is a cross-cutting concern: all services should emit metrics, logs, and tracing information for monitoring.

Summary of Key Patterns: Use **Circuit Breakers** and **Retry** (Polly) to handle transient faults; implement **Health Checks** in each service and expose `/health` endpoints; use an **API Gateway** for entry; secure calls with JWT/OAuth2; ensure **Service Discovery** (static or dynamic) so services can call each other; and make everything observable (Prometheus, Grafana, OpenTelemetry) to detect issues quickly.