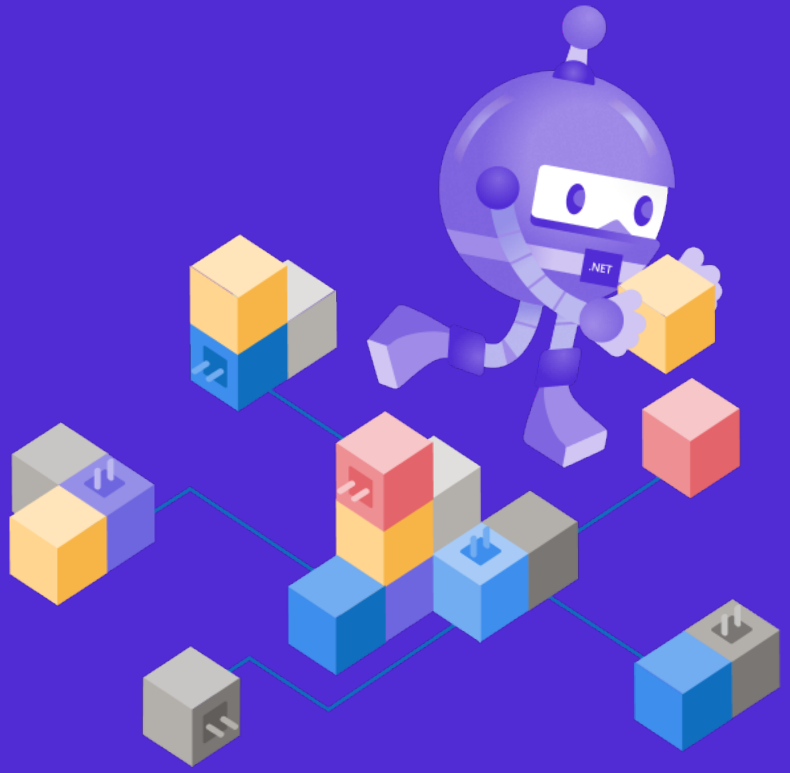
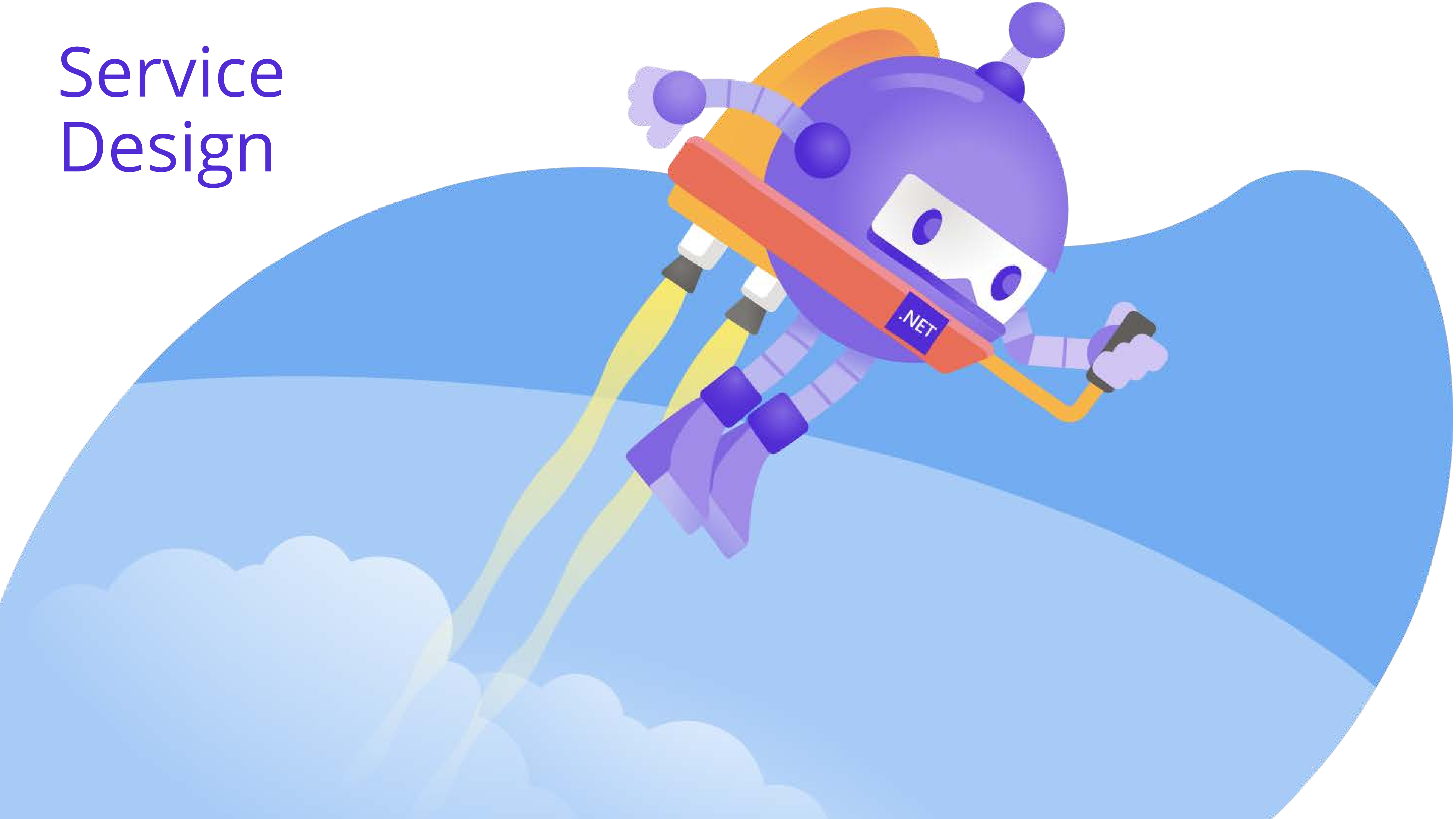
Architecting Microservices

Rob Vettor

Monu Bambroo

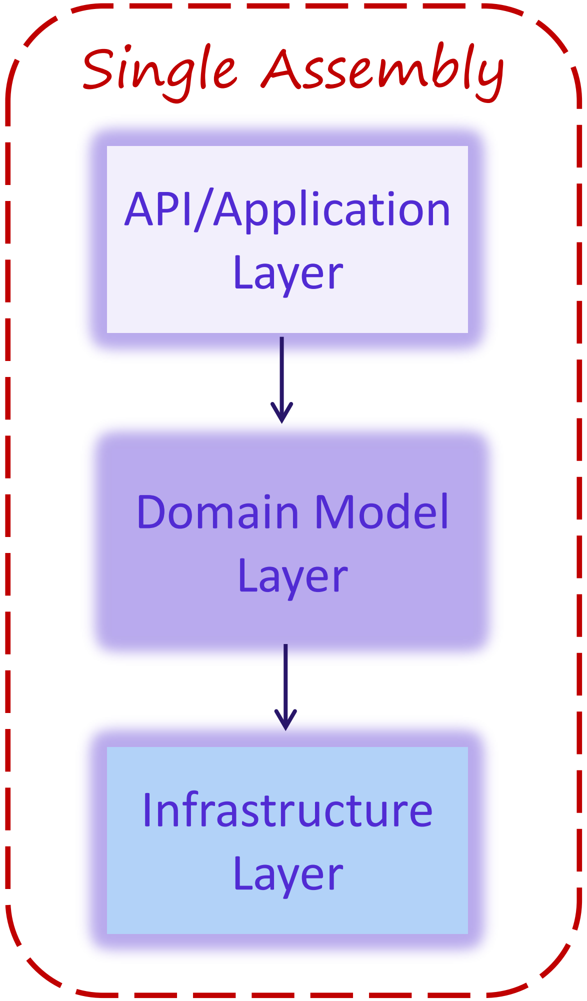


Service Design



Simple Data-Driven, CRUD Microservice

- Contained in a *single assembly*
- Implements a logical *layered architecture design* to enforce SoC



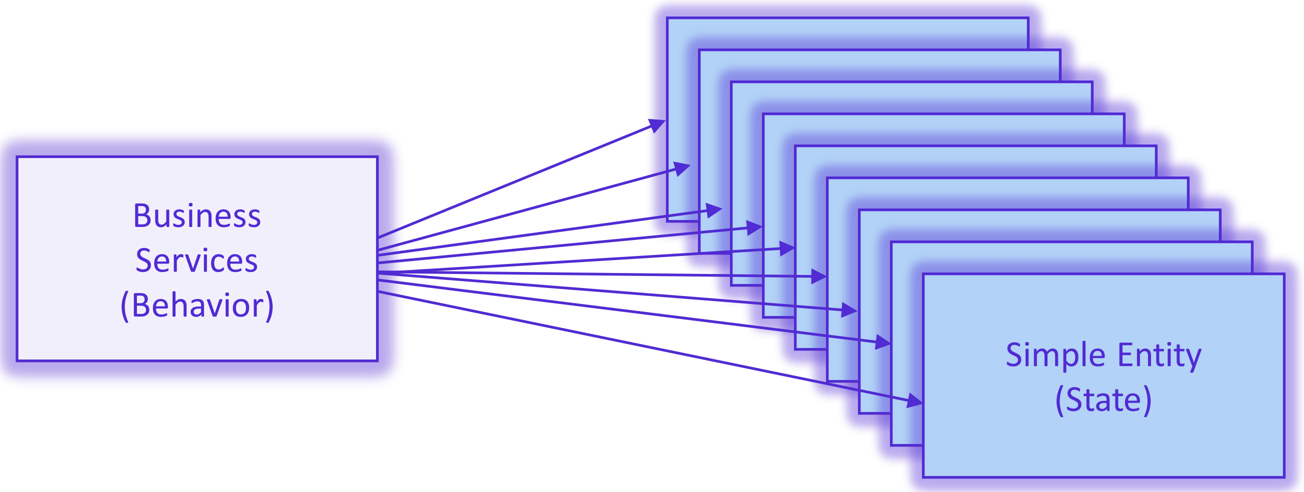
API/Application Layer: Entry point into service. Exposes endpoints and enforces validation. No business logic.

Domain Layer: Heart of the software. Contains business rules and logic. Business operations are implemented here.

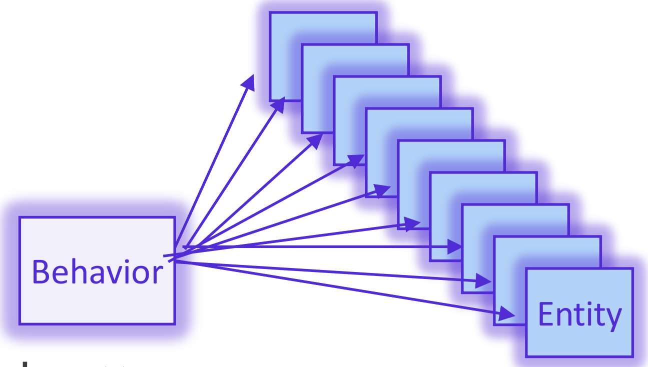
Infrastructure Layer: Provides infrastructure plumbing. Primary responsibility is persistence of business state.

Anemic Domain Model

- Basket and Catalog both implement an *Anemic Domain model*
- Common procedural design with the following characteristics:
  - A business service class that contains behavior, rules and logic
  - Multiple entity classes that only contain state (getter and setter properties)



Anemic Domain Model - Considerations

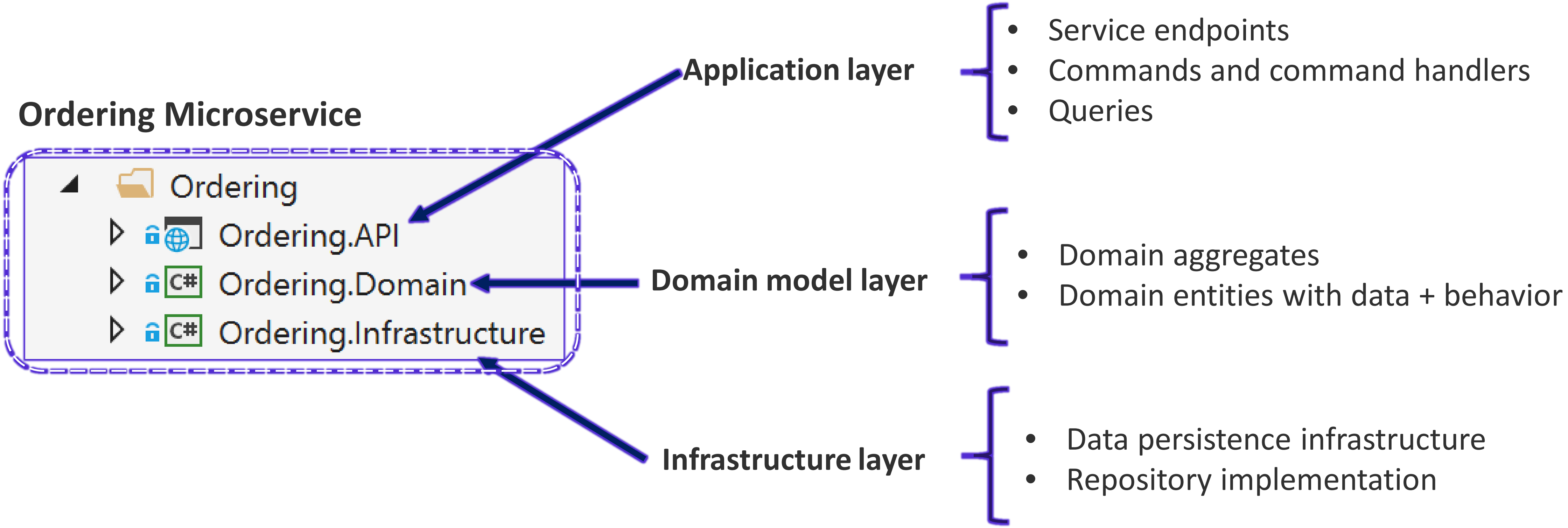


- Seductively simple to implement
- Scatters business functionality across multiple classes
  - Expose simple entity classes that only contain data (state) – getters and setters
  - Business service classes consolidate business logic (behavior)
- Can become difficult to understand, test and maintain
- Can fosters duplication
- Appropriate for simple services, but for those with complex or frequently changing business logic

Demo: Anemic Domain Model

Domain Driven Design Approach

- When a service is complex or incurs frequent change
  - Isolate concerns (*physical separation*) across assemblies
  - Implement select *Domain Driven Design (DDD) patterns*
- Consider the Ordering Microservice



Ordering Microservice

- Ordering
  - Ordering.API
  - Ordering.Domain
  - Ordering.Infrastructure

Application layer

- Service endpoints
- Commands and command handlers
- Queries

Domain model layer

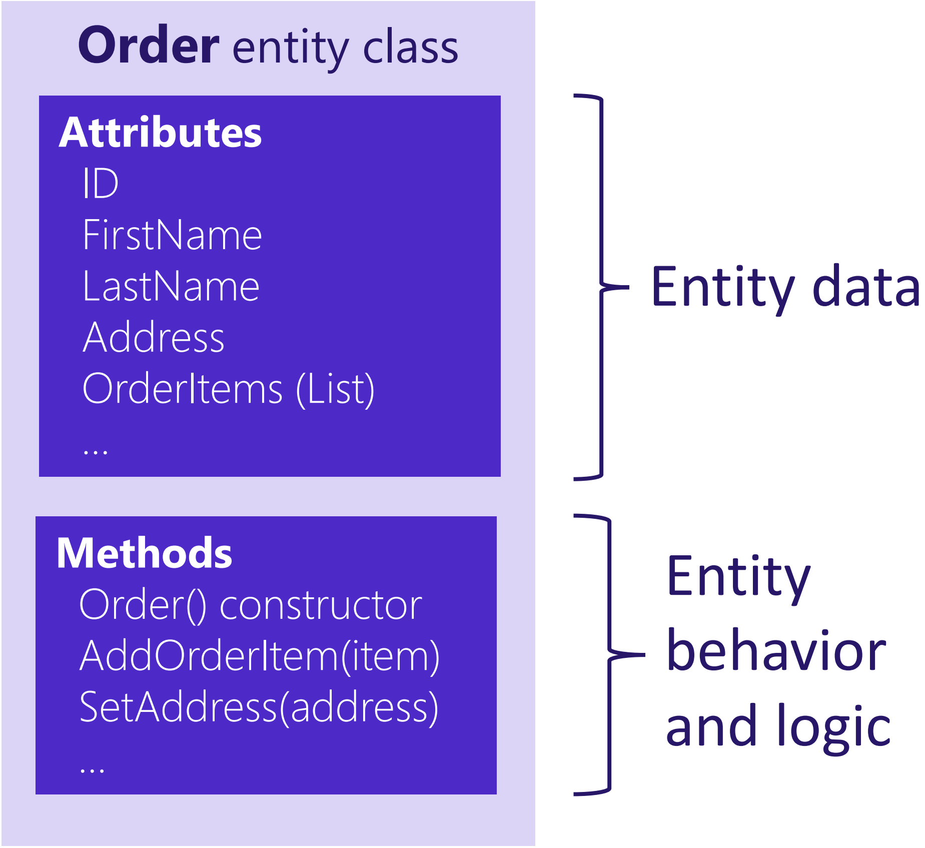
- Domain aggregates
- Domain entities with data + behavior

Infrastructure layer

- Data persistence infrastructure
- Repository implementation

Domain Entity Pattern

- *Domain Entity Pattern*
  - A pattern for implementing business functionality in applications and services
- Each entity class exposes *both state* and *behavior*
- Benefits…
  - One-stop: Encapsulates business logic, state, rules, and relationships inside each entity
  - Improves maintainability
  - Improves testability
  - Helps enforces data integrity
- Better choice for *complex* services
- But, developers must understand DDD principles



Order entity class

Attributes
ID
FirstName
LastName
Address
OrderItems (List)
…

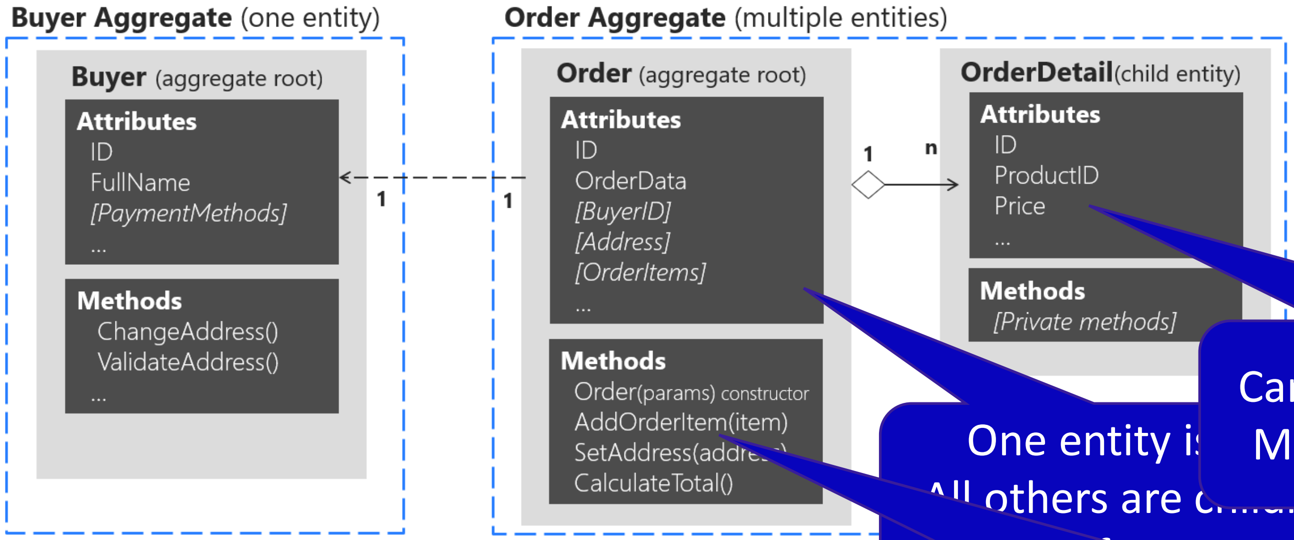
Methods
Order() constructor
AddOrderItem(item)
SetAddress(address)
…

Entity data

Entity behavior and logic

Domain Aggregates

- The Ordering microservice also implements the *Domain Aggregate* patt
- Groups together related entities as an *aggregate*, i.e., Orders and OrderDetails
- Each aggregate is self-contained, encapsulating related state, behavior and business rules



Cannot reference children directly
Must reference through the root

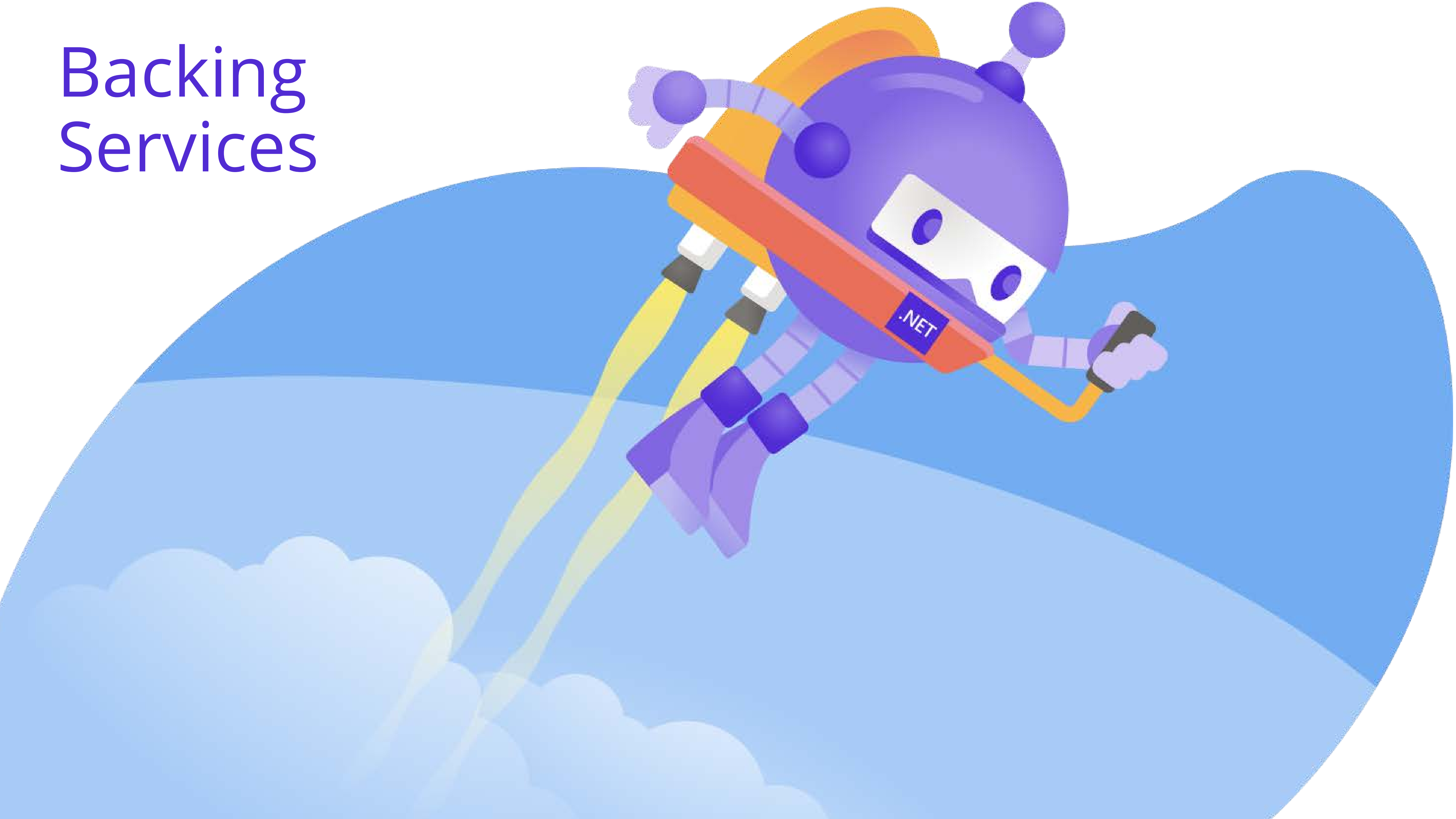
One entity i
All others are children

Root exposes members that access both root and children

- The aggregate acts as a single unit and impleme
- Guarantees consistency at the root level, forbidding external objects from holding references to its internal members

Demo:
DDD Approach

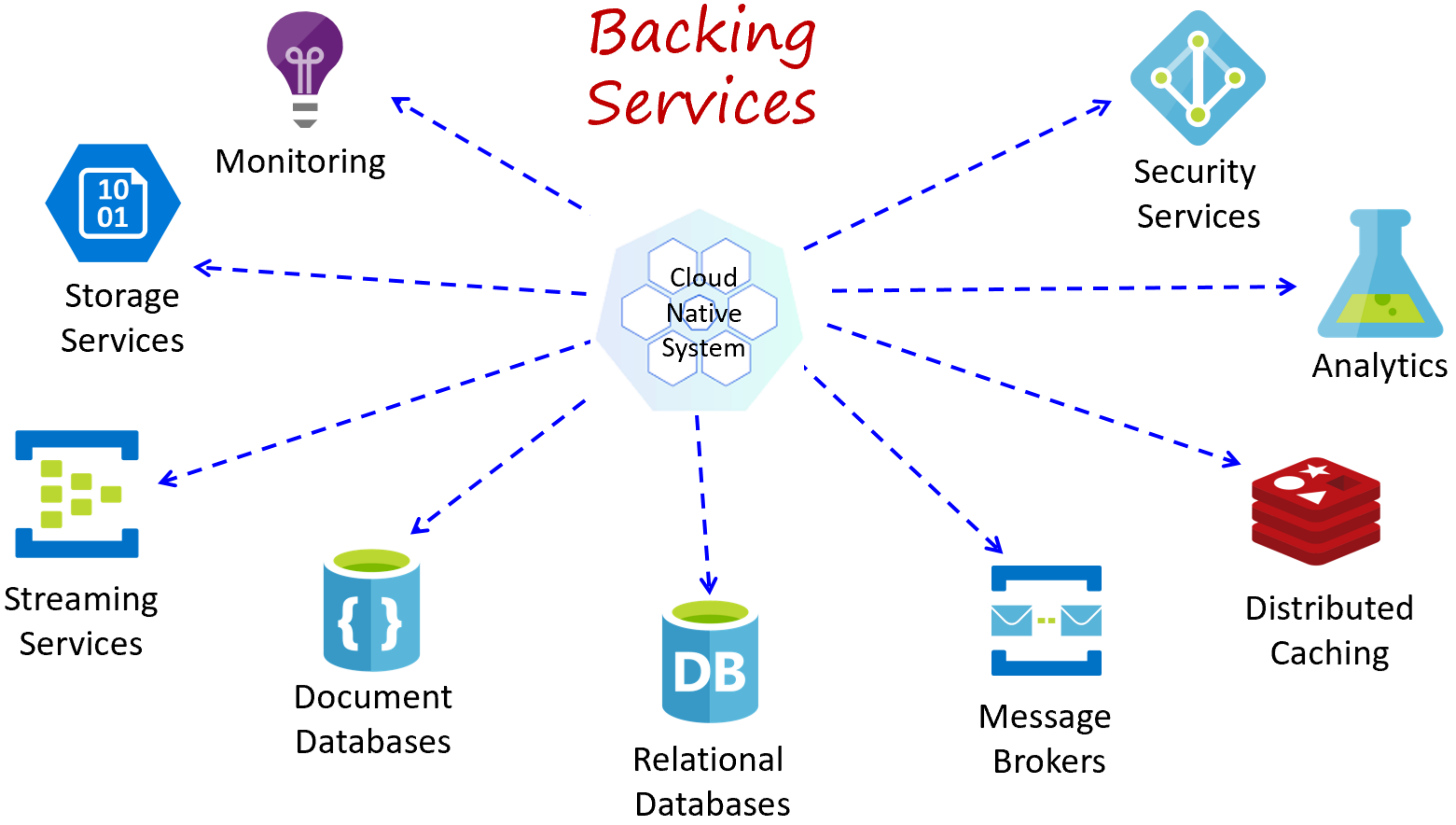
Backing Services



Backing services

- Cloud native systems depend on ancillary resources…
  - Data stores
  - Message brokers
  - Distributed caches
  - Monitoring
  - Identity services

  - These are known as *Backing Services*



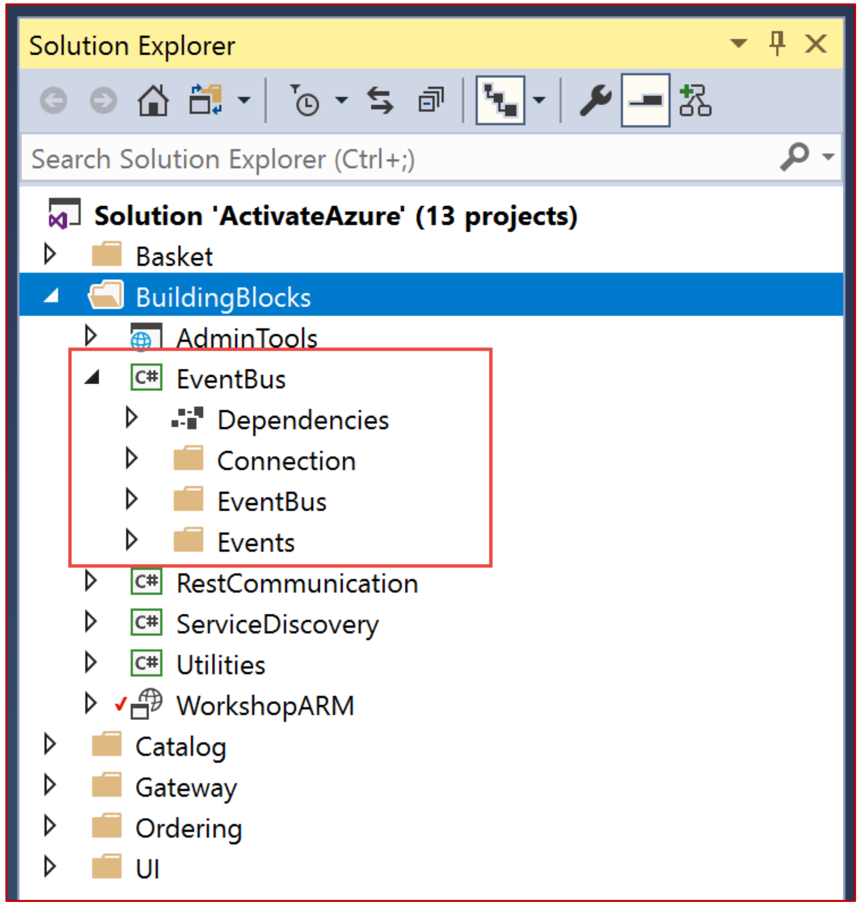
Abstract Backing Services

- Treat backing services as *attached resources…*
  - Attach/detach without application code changes
- Goal: Plug-n-play approach (strategy pattern)…
  - Swap-out a backing service without having to change mainline service code
- Application should never directly reference a backing service (loose coupling)
  - Encapsulate backing service inside an *abstraction shim* (interface)
  - Communicate with shim, not the service
  - In .NET Core start-up, isolate dependency injection configuration for backing services into separate extension class that binds to the startup class
  - Isolate configuration outside of the application

- When possible, implement *managed* backing services from cloud vendors

Abstract Message Broker Plumbing

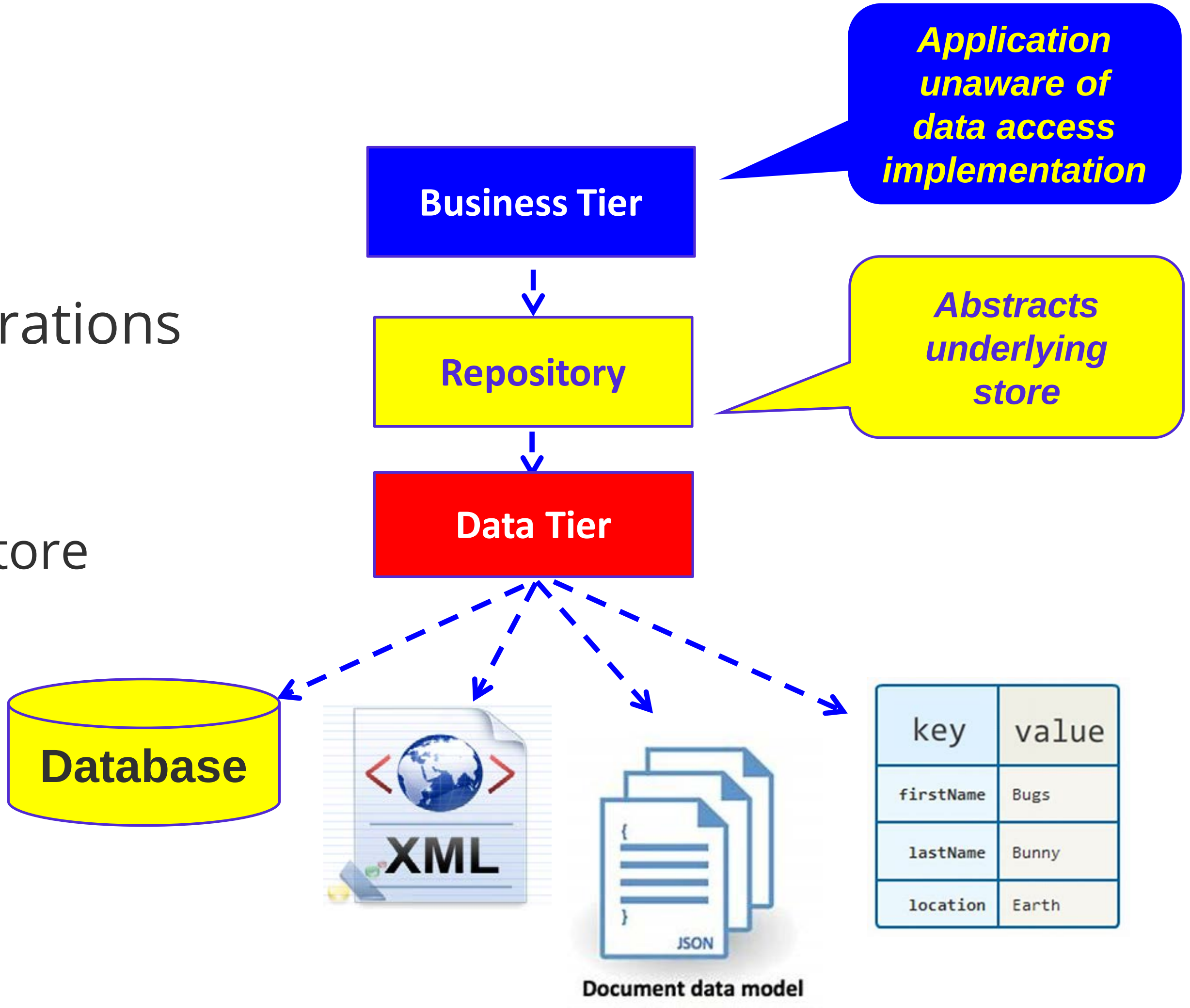
- Best practice: Encapsulate publish/subscribe and command messaging
- Custom contract (interface) that exposes messaging operations
- Implement a strategy pattern with providers for each message broker
- Interchange message brokers without modifying the mainline application



Demo: Abstracting Messaging

Abstract Data Storage with Repository Pattern

- Design pattern for data persistence
- Decouple service from data store implementation
- Repository objects expose CRUD operations
  - Simplify data access
  - Eliminate redundant data access code
  - Insulate consumer from specific data store technology and changes



Business Tier

Application unaware of data access implementation

Repository

Abstracts underlying store

Data Tier

Database

XML

JSON

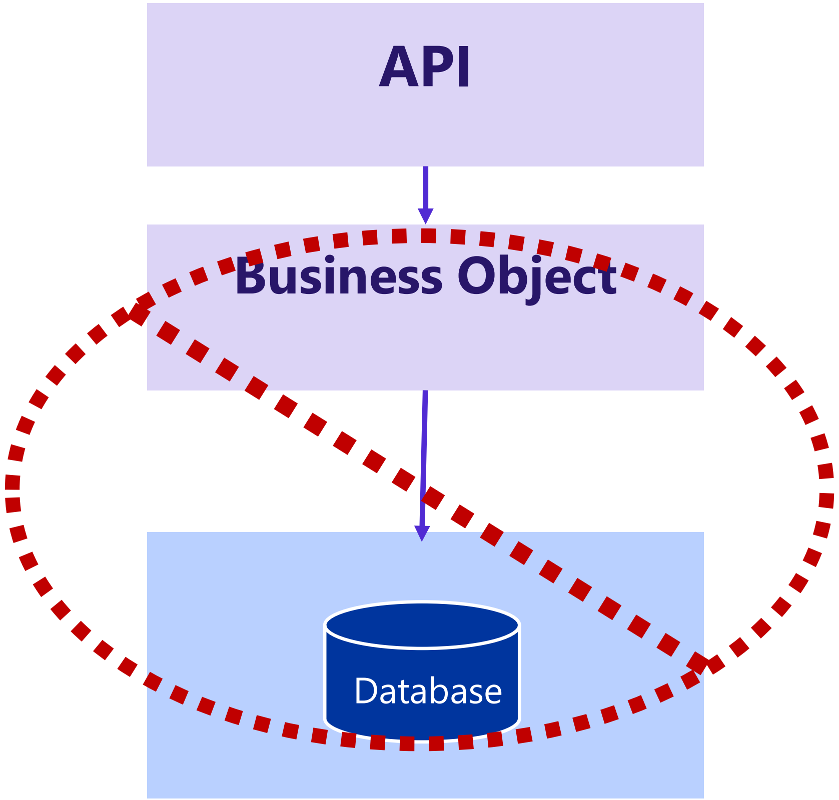
Document data model

key	value
firstName	Bugs
lastName	Bunny
location	Earth

Life with and without a repository

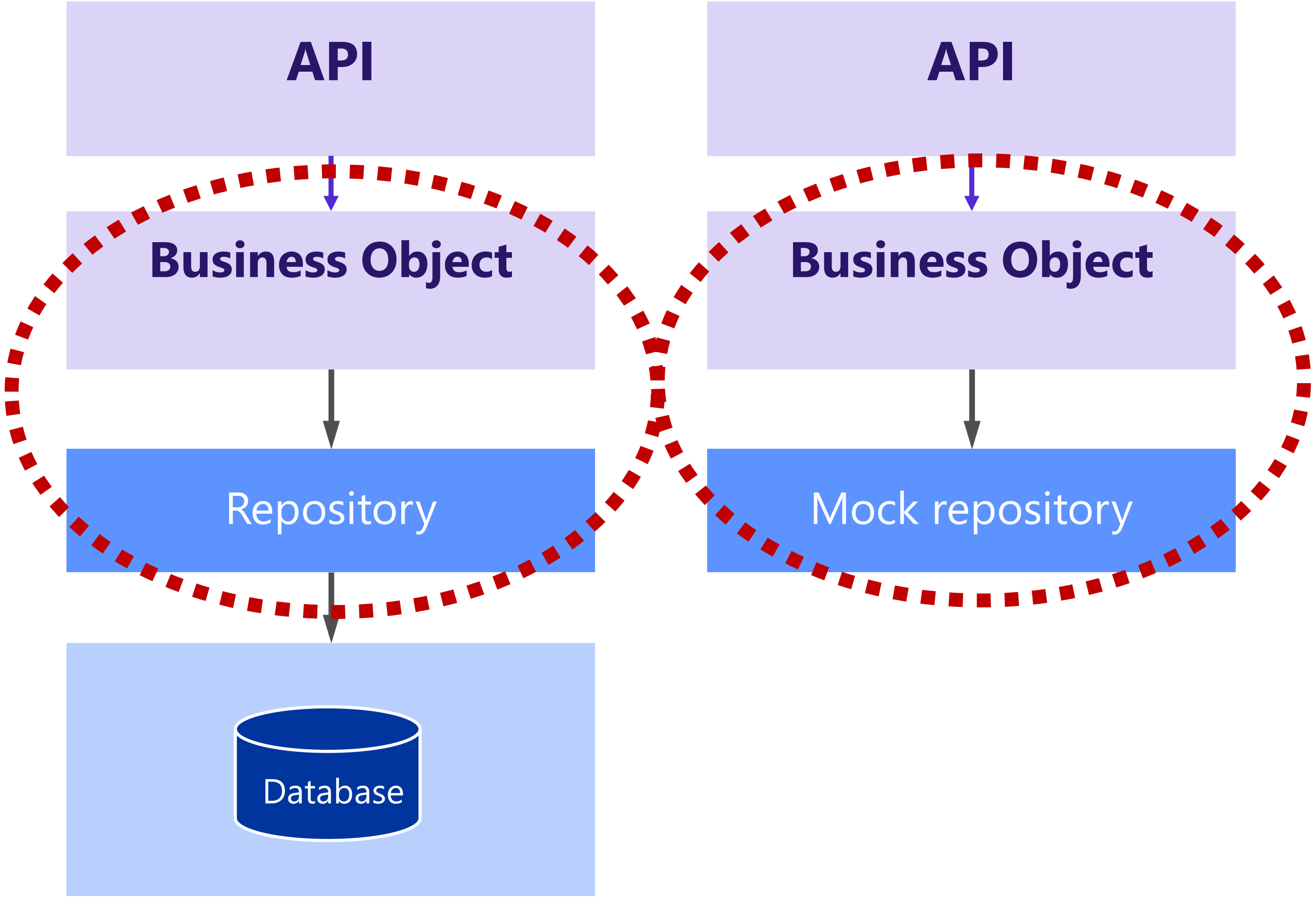
Without Repository

Direct access to database from controller/business layer



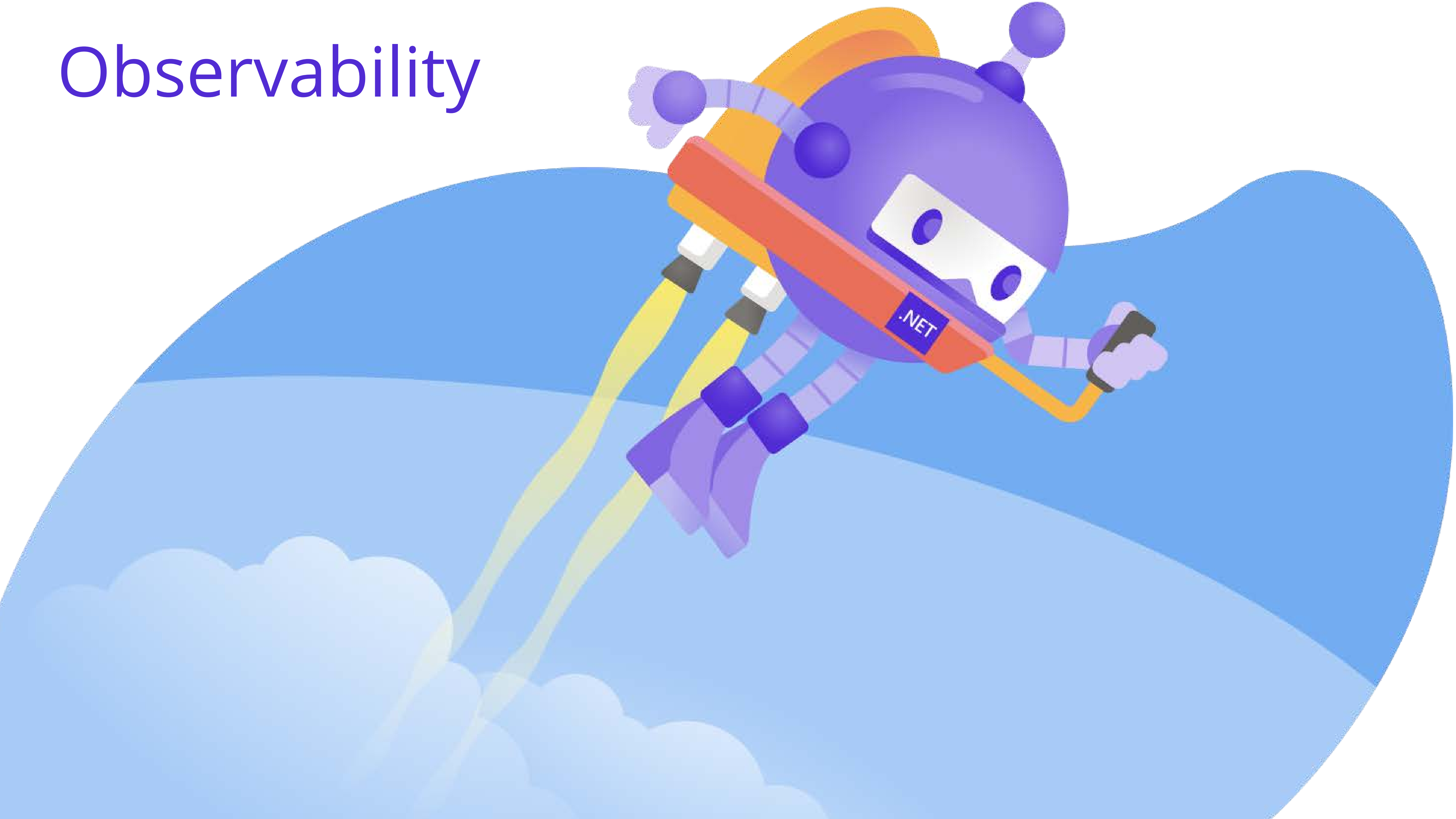
With Repository

Abstraction layer between business layer and database context.
Unit tests can mock data to facilitate testing.



Demo: Abstracting Data

Observability



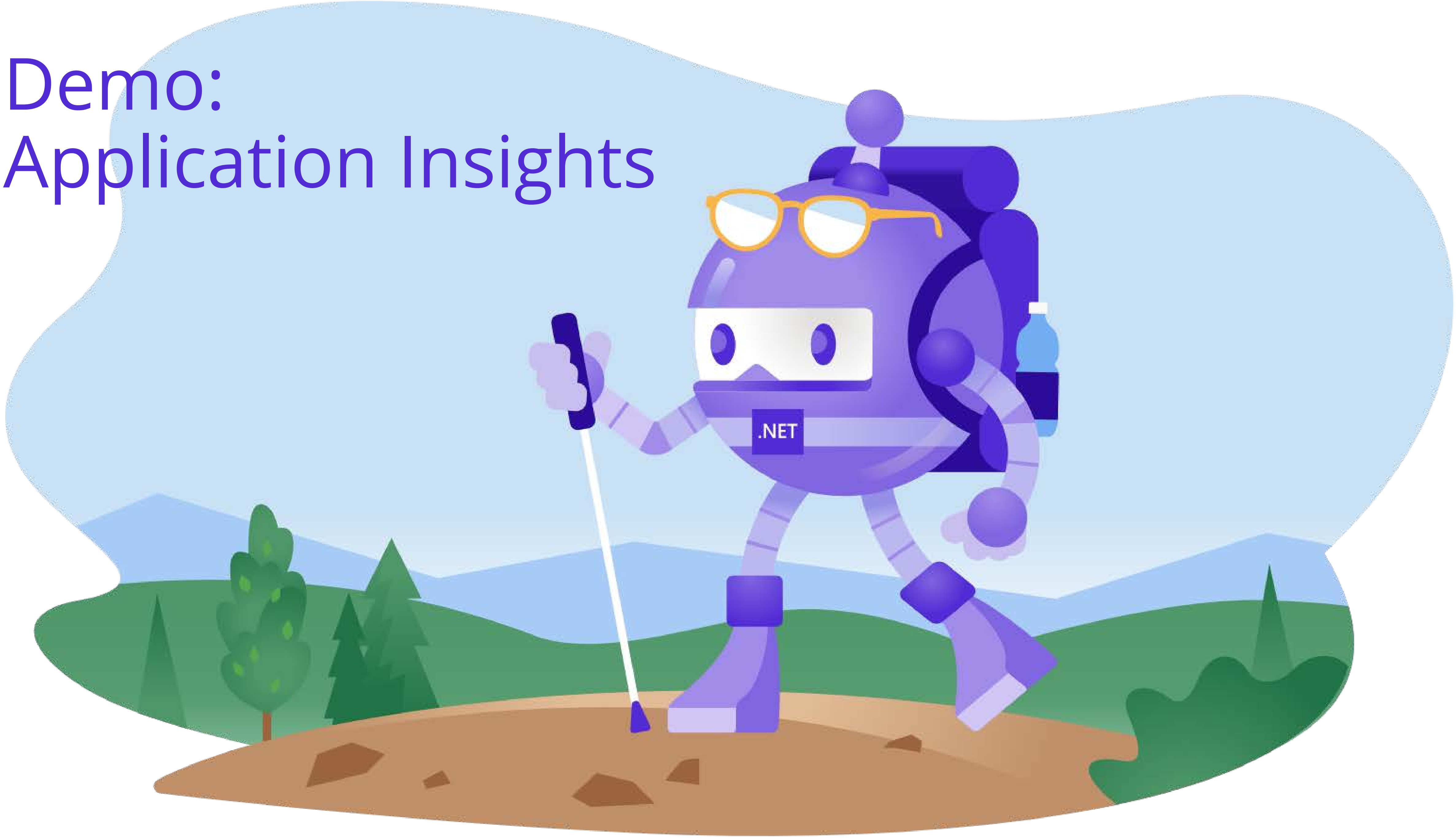
Observability

- Collecting, measuring, and analyzing diagnostics from the system
- Gathering traces, logging, events, metrics

- Choosing an observability platform (Azure App Insights)
- Choosing the frameworks (Serilog)

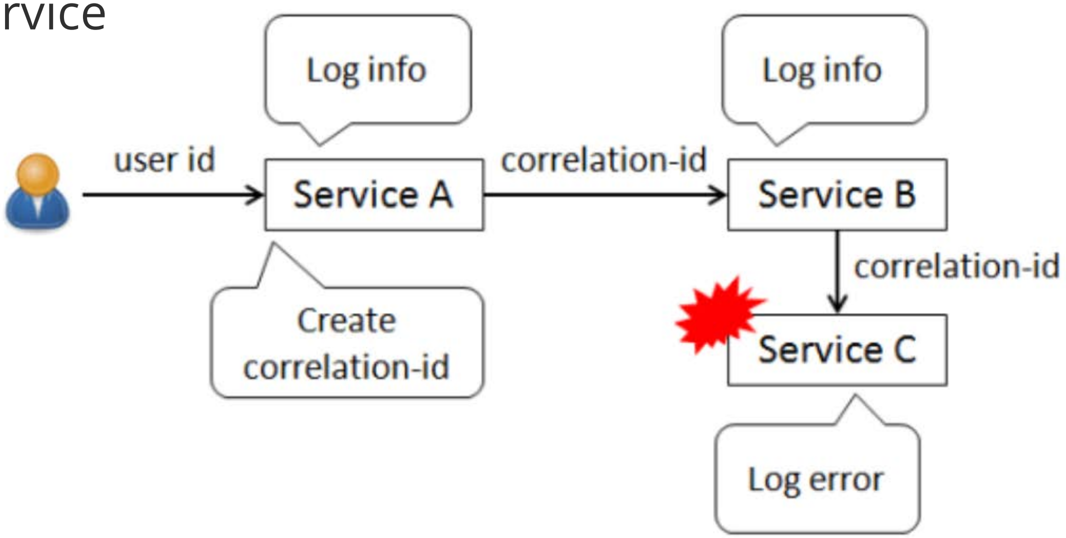
- We chose Azure Application Insights
  - Managed service for collection and analyzing telemetry
  - Feature rich
  - Easy to configure
  - Tightly integrated with Azure backing services
  - Built-in correlation

Demo:
Application Insights

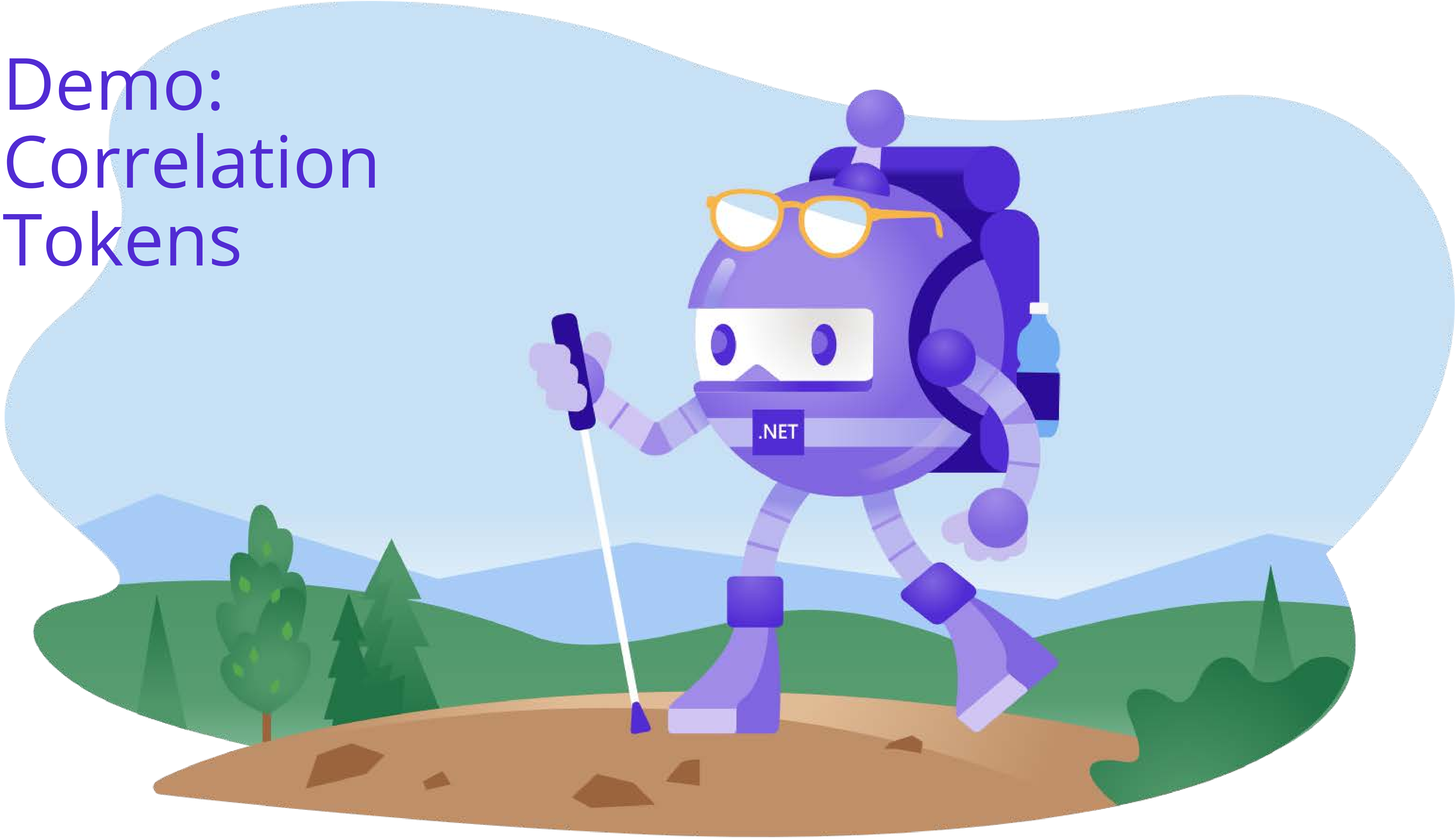


Distributed Tracking and Correlation Tokens

- Each microservice operation should be logged to gain operational insight
- However, correlating events across a set of independent services can be challenging
- A correlational token is a best practice
- A unique identifier generated for each user request
- Tracks flow of a single user request across all services consumed in an operation
  - Generate unique token at beginning of request
  - Ensure that it is passed across each operation in each service
  - Ensure that every logged event includes the token
- Use to tie related messages from different logs



Demo: Correlation Tokens



Microsoft