Implementing API Versioning

Mark Heath
CLOUD ARCHITECT

@mark_heath www.markheath.net

Overview

Entity Framework Core migrations

Microsoft.AspNet.Core.Versioning NuGet package

- Customize API versioning

Maintain backwards compatibility

- Integration testing

Swagger documentation

New Requirements

Events can have multiple tickets

- e.g. Standard and premium tickets

Breaking change to the Event DTO

- Remove the “price” property
- Add a collection of “Ticket” DTOs

Updating the Event DTO

Original definition:

```
public class Event
{
    public Guid EventId { get; set; }
    public string Name { get; set; }
    // only one price is supported
    public int Price { get; set; }
    public string Artist { get; set; }
    public DateTime Date { get; set; }
    public string ImageUrl { get; set; }
    // ...
}
```

Updated definition:

```
public class Event
{
    public Guid EventId { get; set; }
    public string Name { get; set; }
    public string Artist { get; set; }
    public DateTime Date { get; set; }
    // several tickets available:
    public Ticket[] Tickets { get; set; }
    // ...
}
```

Ticket DTO

New Ticket DTO:

```
public class Ticket
{
    public Guid Id { get; set; }
    // e.g. “Standard” or “Premium”
    public string Name { get; set; }
    public int Price { get; set; }
}
```

Potential to expand in the future:

```
public class Ticket
{
    public Guid Id { get; set; }
    public string Name { get; set; }
    public int Price { get; set; }
    public Availability Availability { get; set; }
    public Eligibility Eligibility { get; set; }
    public string ViewImageUrl { get; set; }
    // ...
}
```

Good API design minimizes the need for breaking changes in the future

Shopping Basket DTOs

```
public class BasketLine
{
    public Guid BasketLineId { get; set; }
    public Guid BasketId { get; set; }
    public Guid EventId { get; set; }
    public int Price { get; set; }
    public int TicketAmount { get; set; }
    public Event Event { get; set; }
        public Guid TicketId { get; set; }
}
```

n.b. We still need to support older clients who don't supply a value for TicketId

Demo

Database Migrations

Incremental Update Strategy

1. Update the database schema (add new Tickets table)
2. Provide ticket information for all existing events
3. Update code to read from the new Tickets table
4. Update database schema (remove price column from Events table)

Demo

Enabling Versioning

Supported Version Formats

Date-based (e.g. 2019-05-02.3.0)

Status (e.g. 2.0-Alpha)

Semantic versioning

- Increment major version for breaking changes
- Increment minor version for non-breaking additive changes

Document your APIs

- Clearly indicate the version the documentation relates to
- Still valuable for internal APIs

Demo

Implementing a second version of the Events API

First, publish the new version of your microservice.

Then, update the clients.

Demo

Refactoring code

- Remove V2 suffixes

Demo

Configuring versioning

Configuring Versioning

?api-version=1.4

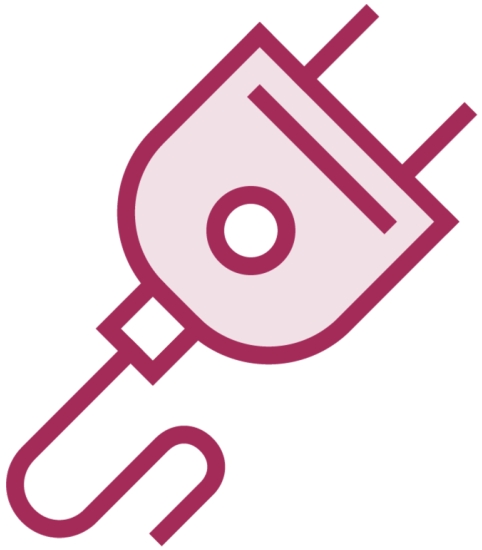
Alternative versioning techniques

- Path-based (e.g. api/v2/)
- Request header

Demo

URL-based versioning

Backwards Compatibility Testing



Ensure that you can still support previous versions of clients

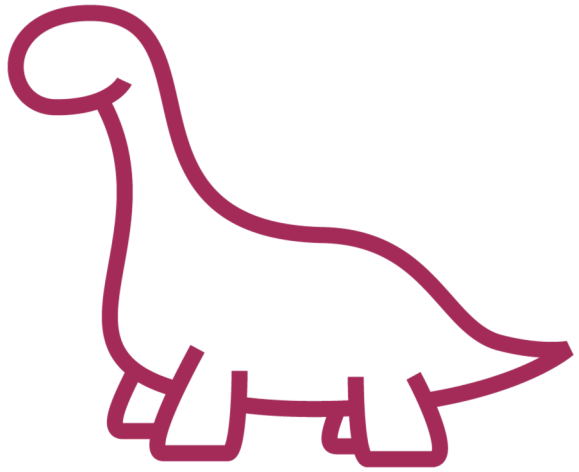
Unit tests

- Fast, in-memory, mock external dependencies

Integration tests

- Actually call the microservice
- Exercise the middleware, database access and mappings

Retiring Old Versions of APIs



Should we support old clients forever?

- Introduces maintenance overhead

Have a policy for retiring old versions

- Support previous version
- Encourage older clients to upgrade
- Third party clients take longer

Demo

Swagger documentation

Summary

Microsoft.AspNetCore.Mvc.Versioning

Use different versioning schemes

Database schema migrations

Integration tests

Swagger documentation

Up next...

Versioning Messages