1. Evolution of Applications: Desktop to Web

Applications have evolved from isolated desktop programs to interconnected web systems in response to changing business needs, technological capabilities, and user expectations. This transition has fundamentally changed how we design, develop, and deploy software, setting the stage for modern API development including REST-based services.

The evolution of applications has been marked by several key transitions:

Desktop Applications (1980s-1990s)

- Self-contained software installed directly on user computers
- Limited connectivity, primarily standalone operation
- Examples: Microsoft Office, Adobe Photoshop, early accounting software

Client-Server Applications (1990s)

- Centralized data storage with distributed processing
- Local clients connecting to central servers
- Examples: Early enterprise resource planning (ERP) systems, database applications

Web Applications (Late 1990s-Present)

- Browser-based interfaces accessible from anywhere
- Centralized deployment and maintenance
- Examples: Gmail, Office 365, Salesforce

Mobile Applications (2007-Present)

- Native apps designed for smartphones and tablets
- App store distribution model
- Examples: Instagram, Uber, banking apps

Cloud-Native Applications (2010s-Present)

- Microservices architecture
- Containerization and orchestration
- Examples: Netflix, modern e-commerce platforms

This transition was driven by several factors:

- Increased internet connectivity and speeds

- Need for centralized maintenance and updates
- Device independence and accessibility
- Business requirements for real-time data and collaboration

2. Communication Between Heterogeneous Systems

Modern software rarely exists in isolation. Systems need to communicate across different platforms, programming languages, and organizational boundaries.

What are heterogeneous systems: These are different types of computer systems with varying architectures, operating systems, programming languages, or data formats that need to work together despite their differences.

Why they are important: Organizations typically use multiple systems (e.g., accounting software, CRM, inventory management) that need to share data. Without effective communication between these heterogeneous systems, businesses would have data silos, redundant processes, and inability to automate cross-system workflows.

RPC (Remote Procedure Call)

- Definition: Allows programs to execute procedures on remote systems
- Examples: gRPC, XML-RPC
- Characteristics: Function-oriented, synchronous communication
- Advantages: Familiar programming model, efficient for procedure-oriented tasks
- Limitations: Tight coupling, limited platform independence

SOAP (Simple Object Access Protocol)

- Definition: XML-based messaging protocol for exchanging structured information
- Characteristics: Protocol-independent, formal contracts via WSDL, extensive standards
- Advantages: Strong typing, built-in error handling, enterprise features (WS-*)
- Limitations: Verbose XML format, complex implementation, performance overhead

REST (Representational State Transfer)

- Definition: Architectural style using standard HTTP methods
- Characteristics: Stateless, resource-oriented, uniform interface
- Advantages: Simplicity, scalability, cacheability, loose coupling
- Limitations: Limited to CRUD operations, potential over-fetching of data

GraphQL

- **Definition**: Query language and runtime for APIs
- **Characteristics**: Client-specified data retrieval, single endpoint
- **Advantages**: Precise data fetching, reduced network overhead, strong typing
- **Limitations**: Complex server implementation, caching challenges

WebSockets

- **Definition**: Protocol for full-duplex communication channels over TCP
- **Characteristics**: Persistent connection, bidirectional communication
- **Advantages**: Real-time data, reduced overhead for frequent updates
- **Limitations**: Stateful, more complex to implement and scale

3. REST: Origins and Concepts

REST has become the dominant architectural style for building web APIs, making it essential knowledge for modern web developers. Understanding its foundations helps implement it correctly.

Historical Context

- Introduced by Roy Fielding in his 2000 doctoral dissertation
- Emerged as a response to the complexity of SOAP and other RPC-based approaches
- Designed to leverage existing web infrastructure (HTTP)
- Became popular with the rise of mobile applications and public APIs

Benefits of REST

- **Simplicity**: Easy to understand and implement
- **Scalability**: Stateless nature allows for horizontal scaling
- **Flexibility**: Support for multiple data formats (JSON, XML, etc.)
- **Visibility**: Operations visible in HTTP methods
- **Portability**: Can be consumed by any client that speaks HTTP
- **Reliability**: Built on proven web technologies

Disadvantages of REST

- **Over-fetching**: May return more data than needed
- **Under-fetching**: May require multiple requests to get complete data
- **Limited operations**: Primarily CRUD-based
- **Versioning challenges**: No standard approach to API versioning
- **Statelessness limitations**: Challenging for complex transactions

4. Main Characteristics of REST

Many APIs claim to be RESTful without actually following REST principles. Now we will take a look at the main characteristics of REST.

4.1. Resource-Based

Explanation: Everything is a resource identified by a unique URI **Example**: `/api/customers/42` represents a specific customer resource **Importance**: Establishes a clear, hierarchical organization of data

4.2. Uniform Interface

Explanation: Consistent operations across all resources

Example:

- GET retrieves a resource
- POST creates a new resource
- PUT updates an existing resource
- PATCH czesciowa aktualizacje zasobu
- DELETE removes a resource

Importance: Predictability and learnability for developers

4.3. Statelessness

Explanation: Each request contains all information needed to process it **Example**: Authentication tokens sent with each request rather than stored in server session **Importance**: Enables scalability and reliability as servers don't need to maintain client state

4.4. Client-Server Architecture

Explanation: Clear separation of concerns between client and server **Example**: Mobile app (client) consuming ASP.NET Web API (server) **Importance**: Allows independent evolution of client and server components

4.5. Cacheability

Explanation: Responses explicitly define themselves as cacheable or non-cacheable **Example**:

```
Cache-Control: max-age=3600
ETag: "33a64df551425fcc55e4d42a148795d9f25f89d4"
```

Importance: Improves performance and reduces server load

4.6. Layered System

Explanation: Intermediate servers may exist between client and resource **Example**: Load balancers, API gateways, CDNs **Importance**: Enhances scalability and security

4.7. Code on Demand (Optional)

Explanation: Servers can temporarily extend client functionality **Example**: JavaScript sent to browser for execution **Importance**: Enhances client capabilities without permanent changes

5. Richardson Maturity Model - Assessing REST Implementation

The Richardson Maturity Model provides a framework to evaluate how "RESTful" an API truly is, helping developers understand where their implementations stand and how they might evolve to gain more REST benefits.

The Richardson Maturity Model, introduced by Leonard Richardson, defines four levels (0-3) of REST implementation maturity:

Level 0: The Swamp of POX (Plain Old XML)

Characteristics:

- Single URI endpoint for all operations
- Typically uses only POST method regardless of operation type
- Request body contains operation details and parameters
- Often RPC-style interactions over HTTP

Example:

```
POST /api/service HTTP/1.1
Content-Type: application/xml

<operation>
  <name>getCustomer</name>
  <id>42</id>
</operation>
```

Assessment: Not RESTful at all; merely using HTTP as a transport protocol

Level 1: Resources

Characteristics:

- Multiple URI endpoints, each representing a resource
- Still primarily uses POST for all operations
- Resources are properly identified but operations aren't standard

Example:

```
POST /api/customers/42 HTTP/1.1
Content-Type: application/json

{
  "action": "delete"
}
```

Assessment: Basic resource modeling but lacking proper use of HTTP methods

Level 2: HTTP Verbs

Characteristics:

- Proper use of HTTP methods (GET, POST, PUT, DELETE)
- Standard status codes (200, 201, 404, etc.)
- Appropriate use of HTTP headers

Example:

```
DELETE /api/customers/42 HTTP/1.1
```

Assessment: This level represents what most people consider "RESTful APIs"

Level 3: Hypermedia Controls (HATEOAS)

Characteristics:

- Hypermedia As The Engine Of Application State
- Responses include links to related resources and possible actions
- Client discovers API capabilities through navigation, not prior knowledge

Example:

```
{
  "id": 42,
  "name": "John Doe",
  "links": [
    {"rel": "self", "href": "/api/customers/42"},
    {"rel": "orders", "href": "/api/customers/42/orders"},
    {"rel": "update", "href": "/api/customers/42", "method": "PUT"},
    {"rel": "delete", "href": "/api/customers/42", "method": "DELETE"}
```

```
  ]
}
```

Assessment: Fully RESTful API, implementing all of Fielding's constraints

Practical Implications

Most Common Implementation Level: Most real-world "REST" APIs are at Level 2, using proper HTTP methods and status codes but not implementing HATEOAS.

Benefits of Higher Levels:

- Level 3 enables greater decoupling between client and server
- API evolution with minimal client changes
- Self-documenting interfaces
- Greater discoverability

Trade-offs:

- Higher levels require more development effort
- Level 3 can add complexity and verbosity
- Sometimes practical concerns justify stopping at Level 2

Assessment Approach: When evaluating an API's REST maturity, consider:

- Does it use unique URIs for different resources?
- Does it use HTTP methods appropriately?
- Does it use standard status codes?
- Does it include hypermedia controls?
- Is it stateless?

In ASP.NET development, most Web API projects aim for Level 2 maturity, which provides most of the practical benefits of REST while remaining straightforward to implement and consume.

6. Introduction to ASP.NET REST API

ASP.NET provides a robust, mature framework for building REST APIs in the Microsoft ecosystem. Understanding its architecture and capabilities helps developers leverage the full power of the platform for creating efficient, scalable, and maintainable REST services.

ASP.NET: Evolution and Overview

What is ASP.NET? ASP.NET is Microsoft's web application development framework that enables developers to build dynamic web applications, web services, and web APIs. It has evolved significantly since its introduction:

- **Classic ASP.NET (2002)**: Based on the WebForms model with server controls and postbacks
- **ASP.NET MVC (2009)**: Introduced the Model-View-Controller pattern for better separation of concerns
- **ASP.NET Web API (2012)**: Specialized framework optimized for building HTTP services and REST APIs
- **ASP.NET Core (2016)**: Complete rewrite as a cross-platform, high-performance, open-source framework

Key Features of ASP.NET

- Strong typing and compile-time checking
- Comprehensive security features
- Extensive middleware ecosystem
- Integration with the broader .NET ecosystem
- Enterprise-grade performance and reliability

7. ASP.NET REST API Development

ASP.NET Web API vs. ASP.NET Core API

ASP.NET offers two primary approaches to building REST APIs:

1. **ASP.NET Web API (Traditional)**:
   - Part of the .NET Framework
   - Windows-only
   - Separate from MVC stack
2. **ASP.NET Core API**:
   - Cross-platform (.NET Core/.NET 5+)
   - Unified programming model with MVC
   - Better performance
   - Modern middleware pipeline
   - Built-in dependency injection

8. MVC Architecture in ASP.NET

The Model-View-Controller (MVC) pattern is a fundamental architectural pattern in ASP.NET that separates an application into three main components, promoting separation of concerns and maintainability. When applied to REST API development, this pattern provides a structured approach to handling HTTP requests and responses.

8.1. Model

- Represents the application's data and business logic
- Handles data access, validation, and business rules
- Remains independent of the user interface
- In REST APIs, models typically represent resources and their relationships

Example:

```
public class Customer
{
    public int Id { get; set; }

    [Required]
    [StringLength(50)]
    public string Name { get; set; }

    [EmailAddress]
    public string Email { get; set; }

    public DateTime RegistrationDate { get; set; }

    public virtual ICollection<Order> Orders { get; set; }
}
```

8.2. View

- In traditional MVC web applications, views render the UI
- In REST APIs, "views" are replaced by formatted responses (JSON/XML)
- Handled by formatters that serialize model objects into response formats
- Can be customized through output formatters or view models/DTOs

Example of a DTO (Data Transfer Object):

```
public class CustomerDto
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
    public int OrderCount { get; set; }
    public decimal TotalSpent { get; set; }
}
```

8.3. Controller

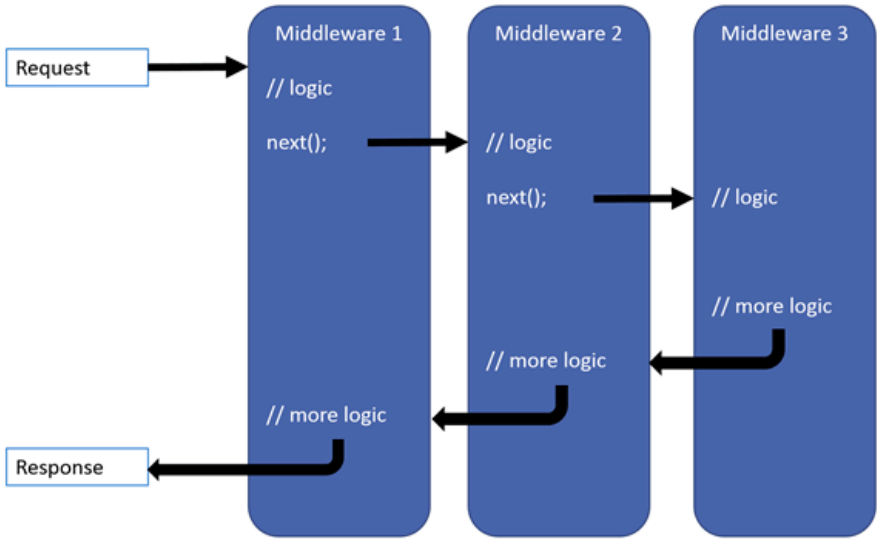
- Processes incoming HTTP requests
- Coordinates between models and views
- Contains action methods that correspond to HTTP verbs
- Returns appropriate HTTP responses
- Focuses on HTTP concerns, not business logic

Example:

```
[Route("api/[controller]")]
[ApiController]
public class CustomersController : ControllerBase
{
    private readonly ICustomerRepository _repository;
    private readonly IMapper _mapper;

    public CustomersController(ICustomerRepository repository, IMapper
mapper)
    {
        _repository = repository;
        _mapper = mapper;
    }

    [HttpGet]
    public async Task<ActionResult<IEnumerable<CustomerDto>>> GetAll()
    {
        var customers = await _repository.GetAllAsync();
        return Ok(_mapper.Map<IEnumerable<CustomerDto>>(customers));
    }

    [HttpGet("{id}")]
    public async Task<ActionResult<CustomerDto>> GetById(int id)
    {
        var customer = await _repository.GetByIdAsync(id);
```

```
        if (customer == null)
            return NotFound();

        return Ok(_mapper.Map<CustomerDto>(customer));
    }

    [HttpPost]
    public async Task<ActionResult<CustomerDto>> Create([FromBody]
CustomerCreateDto dto)
    {
        if (!ModelState.IsValid)
            return BadRequest(ModelState);

        var customer = _mapper.Map<Customer>(dto);
        await _repository.AddAsync(customer);

        var resultDto = _mapper.Map<CustomerDto>(customer);
        return CreatedAtAction(nameof(GetById), new { id = customer.Id
}, resultDto);
    }
}
```

9. ASP.NET Request Execution Pipeline Overview



The ASP.NET request execution pipeline (also known as middleware pipeline in modern ASP.NET Core) describes the sequence of steps that happen when an HTTP request arrives until a response is returned.

Basic Flow in ASP.NET Core

1. **Request Arrival**: The web server (Kestrel, IIS, etc.) receives an HTTP request.
2. **Middleware Processing**: The request passes through a series of middleware components configured in the `Startup.cs` file (or `Program.cs` in minimal APIs):
   - Each middleware can:
     - Process the request
     - Pass it to the next middleware
     - Short-circuit the pipeline
     - Modify the response
3. **Routing**: The routing middleware determines which endpoint should handle the request.
4. **Controller/Action Invocation** (in MVC): If using MVC, the appropriate controller and action method are located and executed.
5. **Response Generation**: The action method or endpoint handler generates a response.
6. **Middleware (Reverse)**: The response passes back through the middleware chain in reverse order.
7. **Response Sent**: The complete HTTP response is sent back to the client.

Common middleware includes authentication, authorization, exception handling, static files, CORS, and more, which you can configure in the order needed for your application.

10. Routing

Routing is the mechanism that maps incoming HTTP requests to specific controller action methods in an ASP.NET Core application. It's a critical part of the request pipeline that determines which code executes when a request arrives.

10.1. Conventional Routing vs. Attribute Routing

Conventional Routing

Conventional routing defines route patterns centrally in the `Startup.cs` file (or `Program.cs` in minimal APIs) using a route template syntax. These routes are defined once and apply to all controllers.

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

In this example:

- `{controller}` matches the controller name (without the "Controller" suffix)
- `{action}` matches the action method name
- `{id?}` is an optional parameter
- Default values are provided for controller and action

Advantages:

- Centralized route configuration
- Easier to get a global view of all routes
- Can change routing patterns without modifying controller code

Disadvantages:

- Less explicit connection between routes and action methods
- Can be harder to understand which route maps to which action when reading controller code
- Limited flexibility for complex routing scenarios

Attribute Routing

Attribute routing defines routes directly on controllers and action methods using attributes. This makes the connection between routes and actions explicit and easier to understand.

```
[Route("api/[controller]")]
public class ProductsController : ControllerBase
{
    [HttpGet]
    public IActionResult GetAll() { ... }

    [HttpGet("{id}")]
    public IActionResult GetById(int id) { ... }
}
```

Advantages:

- Clear, direct association between routes and actions
- More flexible for complex routing scenarios
- Better support for REST API conventions
- Easier to maintain as the application grows

Disadvantages:

- Routes are scattered throughout the codebase
- Can be harder to get a global view of all routes
- Requires changing controller code to modify routes

When to Use Each Approach

- Conventional Routing: Better for traditional MVC applications with views
- Attribute Routing: Better for REST APIs and complex routing scenarios

Many applications use a combination of both approaches.

11. Routing Attribute Types

ASP.NET Core provides several route-related attributes to handle different scenarios:

Basic Route Attributes

[Route]

Defines a route template for a controller or action.

```
[Route("api/[controller]")]
public class ProductsController : ControllerBase { ... }

[Route("items")]
public IActionResult GetItems() { ... }
```

[HttpGet], [HttpPost], [HttpPut], [HttpDelete], [HttpPatch]

Define routes for specific HTTP methods.

```
[HttpGet]  // Maps to GET request
public IActionResult Get() { ... }

[HttpPost] // Maps to POST request
public IActionResult Create() { ... }
```

These can include route templates:

```
[HttpGet("active")]  // GET /api/products/active
public IActionResult GetActive() { ... }

[HttpGet("{id:int}")] // GET /api/products/5
public IActionResult GetById(int id) { ... }
```

Route Constraint Attributes

Route constraints limit what values can match a route parameter:

```
[HttpGet("{id:int:min(1)}")]  // Must be integer >= 1
public IActionResult GetById(int id) { ... }

[HttpGet("by-date/{date:datetime}")]  // Must be valid datetime
public IActionResult GetByDate(DateTime date) { ... }
```

```
[HttpGet("items/{sku:regex(^[A-Z]\\d{{3}}$)}")]  // Must match regex
public IActionResult GetBySku(string sku) { ... }
```

Common constraints include:

- `int` - Must be an integer
- `bool` - Must be a boolean
- `datetime` - Must be a valid DateTime
- `decimal` - Must be a decimal number
- `min`, `max` - Specifies minimum or maximum value
- `length` - Specifies string length
- `regex` - Must match the regular expression
- `alpha` - Must be alphabetical characters
- `required` - Parameter must be present

Route Name and Order Attributes

[Route("template", Name = "RouteName")]

Assigns a name to a route for URL generation.

```
[HttpGet("{id}", Name = "GetProductById")]
public IActionResult GetById(int id) { ... }

// Later, generate URL to this route:
var url = Url.RouteUrl("GetProductById", new { id = 123 });
```

[Route("template", Order = 1)]

Sets the route evaluation order when multiple routes could match.

```
[HttpGet("details/{id}", Order = 2)]  // Evaluated second
public IActionResult Details(int id) { ... }

[HttpGet("{action}/{id?}", Order = 1)]  // Evaluated first
public IActionResult Index(int? id) { ... }
```

12. Passing Data to Action Methods

ASP.NET Core provides several ways to pass data from HTTP requests to action methods:

Route Values

Data extracted from the URL route segments.

```
// URL: /api/products/5
[HttpGet("{id}")]
public IActionResult GetById(int id) { ... }  // id = 5
```

Query String Parameters

Data from URL query strings.

```
// URL: /api/products?category=electronics&sort=price
[HttpGet]
public IActionResult Get([FromQuery] string category, [FromQuery] string
sort) { ... }
```

The `[FromQuery]` attribute is optional but makes the binding source explicit.

Request Body

Data from the request body, typically in JSON format.

```
// POST /api/products with JSON body: {"name":"Product1","price":29.99}
[HttpPost]
public IActionResult Create([FromBody] ProductModel product) { ... }
```

Form Data

Data from form submissions.

```
// POST with form data
[HttpPost]
public IActionResult Create([FromForm] ProductModel product) { ... }
```

Headers

Data from HTTP headers.

```
[HttpGet]
public IActionResult Get([FromHeader(Name = "User-Agent")] string
userAgent) { ... }
```

Model Binding with Multiple Sources

You can create a model class that combines data from multiple sources:

```
public class SearchParameters
{
    [FromRoute] public int? CategoryId { get; set; }
    [FromQuery] public string SearchTerm { get; set; }
    [FromQuery] public string SortBy { get; set; }
    [FromQuery] public int Page { get; set; } = 1;
    [FromHeader(Name = "Accept-Language")] public string Language { get;
set; }
}

[HttpGet("categories/{categoryId?}")]
public IActionResult Search([FromServices] IProductService service,
                            SearchParameters parameters) { ... }
```

Value Providers and Custom Model Binding

ASP.NET Core uses value providers to extract values from the request:

1. Form value provider
2. Route value provider
3. Query string value provider
4. Header value provider
5. Body value provider (for JSON/XML)

You can create custom value providers for specialized scenarios.

13. Route Template Syntax

Basic Template Components

- Literal segments: `/products`
- Parameter segments: `{id}`
- Optional parameters: `{id?}`
- Default values: `{controller=Home}`
- Constraints: `{id:int}`
- Catch-all parameters: `{**slug}`

Catch-all Parameters

Match multiple segments:

```
[HttpGet("docs/{**path}")]
public IActionResult ShowDocumentation(string path) { ... }
// Matches: /docs/getting-started/installation
// path = "getting-started/installation"
```

Token Replacement

Special tokens in route templates:

```
[Route("api/[controller]")]  // [controller] replaced with controller
name
public class ProductsController : ControllerBase { ... }
// Route becomes "api/products"

[HttpGet("[action]")]  // [action] replaced with action method name
```

```
public IActionResult Details(int id) { ... }
// Route becomes "api/products/details"
```

14. Practical Examples

```
[Route("api/[controller]")]
[ApiController]
public class ProductsController : ControllerBase
{
    // GET /api/products
    [HttpGet]
    public IActionResult GetAll([FromQuery] string category = null,
                                [FromQuery] string sort = null) { ... }

    // GET /api/products/5
    [HttpGet("{id:int}")]
    public IActionResult GetById(int id) { ... }

    // GET /api/products/by-sku/ABC123
    [HttpGet("by-sku/{sku}")]
    public IActionResult GetBySku(string sku) { ... }

    // POST /api/products
    [HttpPost]
    public IActionResult Create([FromBody] ProductModel product) { ... }

    // PUT /api/products/5
    [HttpPut("{id}")]
    public IActionResult Update(int id, [FromBody] ProductModel product)
{ ... }

    // DELETE /api/products/5
    [HttpDelete("{id}")]
    public IActionResult Delete(int id) { ... }
}
```

Complex Routing Scenarios

```
[Route("catalog")]
public class CatalogController : Controller
{
    // GET /catalog
    [HttpGet]
    public IActionResult Index() { ... }

    // GET /catalog/categories
    [HttpGet("categories")]
```

```
    public IActionResult Categories() { ... }

    // GET /catalog/categories/5/products
    [HttpGet("categories/{categoryId}/products")]
    public IActionResult ProductsByCategory(int categoryId) { ... }

    // GET /catalog/products/available?minPrice=10&maxPrice=50
    [HttpGet("products/available")]
    public IActionResult AvailableProducts([FromQuery] decimal?
minPrice,
                                           [FromQuery] decimal?
maxPrice) { ... }

    // POST /catalog/products/import
    [HttpPost("products/import")]
    public IActionResult ImportProducts([FromForm] IFormFile file) { ...
}
}
```