ASP.NET WEB API CORE

Lesson 02: Introduction to .Net Core WebApi Configuration

Lesson Objectives

- In this lesson we will cover the following
  - Configuring WebApi core
  - Web API Parameter Biding
  - Action Return Type
  - WebApi Filters
  - Security and Roles (OpenId / OAuth)
  - Content Negotiation

Configuring Web Api Core

ASP.NET Core doesn't use the App_Start folder or the Global.asax file.

- Additionally, the web.config file is added at publish time.

- The Startup class:
  - \Replaces Global.asax.
  - Handles all app startup tasks.

- The ASP.NET Core API project template includes endpoint routing configuration in the generated code.

- The following UseRouting and UseEndpoints calls:
  - Register route matching and endpoint execution in the middleware pipeline.
  - Replace the ProductsApp project's App_Start/WebApiConfig.cs file.

- The preceding [Route] attribute configures the controller's attribute routing pattern. The [ApiController] attribute makes attribute routing a requirement for all actions in this controller.

Configuring Web Api Core

- Attribute routing supports tokens, such as [controller] and [action].

- At runtime, each token is replaced with the name of the controller or action, respectively, to which the attribute has been applied.

- The tokens:
  - Reduce the number of magic strings in the project.
  - Ensure routes remain synchronized with the corresponding controllers and actions when automatic rename refactorings are applied.

Web API Parameter Biding

- Action methods in Web API controller can have one or more parameters of different types.
- It can be either primitive type or complex type.

- Web API binds action method parameters either with URL's query string or with request body depending on the parameter type.

- By default, if parameter type is of .NET primitive type such as int, bool, double, string, GUID, DateTime, decimal or any other type that can be converted from string type then it sets the value of a parameter from the query string.

- And if the parameter type is complex type then Web API tries to get the value from request body by default.

Web API Parameter Biding Contin..

```
Public HttpResponseMessage Put(int id, Employee employee)
{
…
…
}
```

In this example, ID parameter is a primitive type so Web API tried to get this parameter value from the URL and employee parameter is a complex type, so Web API tried to get the value of this parameter from request body, using media-type formatter.

We can also force Web API to get the parameter value from the request body or request URL by using FromUri and FromBody attribute.

FromUri Attribute

FromUri Attribute forces the Web API to read the parameter value from the requested URL. In the following example, I have defined the complex type and forced the Web API to read the value for complex type from the requested parameter.

```
public classTestData
{
    public string Name
    {
       get;
       set;
   }
public int Id
   {
       get;
     set;
    }
}
```

FromUri Attribute Continue..

Get method of Employee controller class shown below:

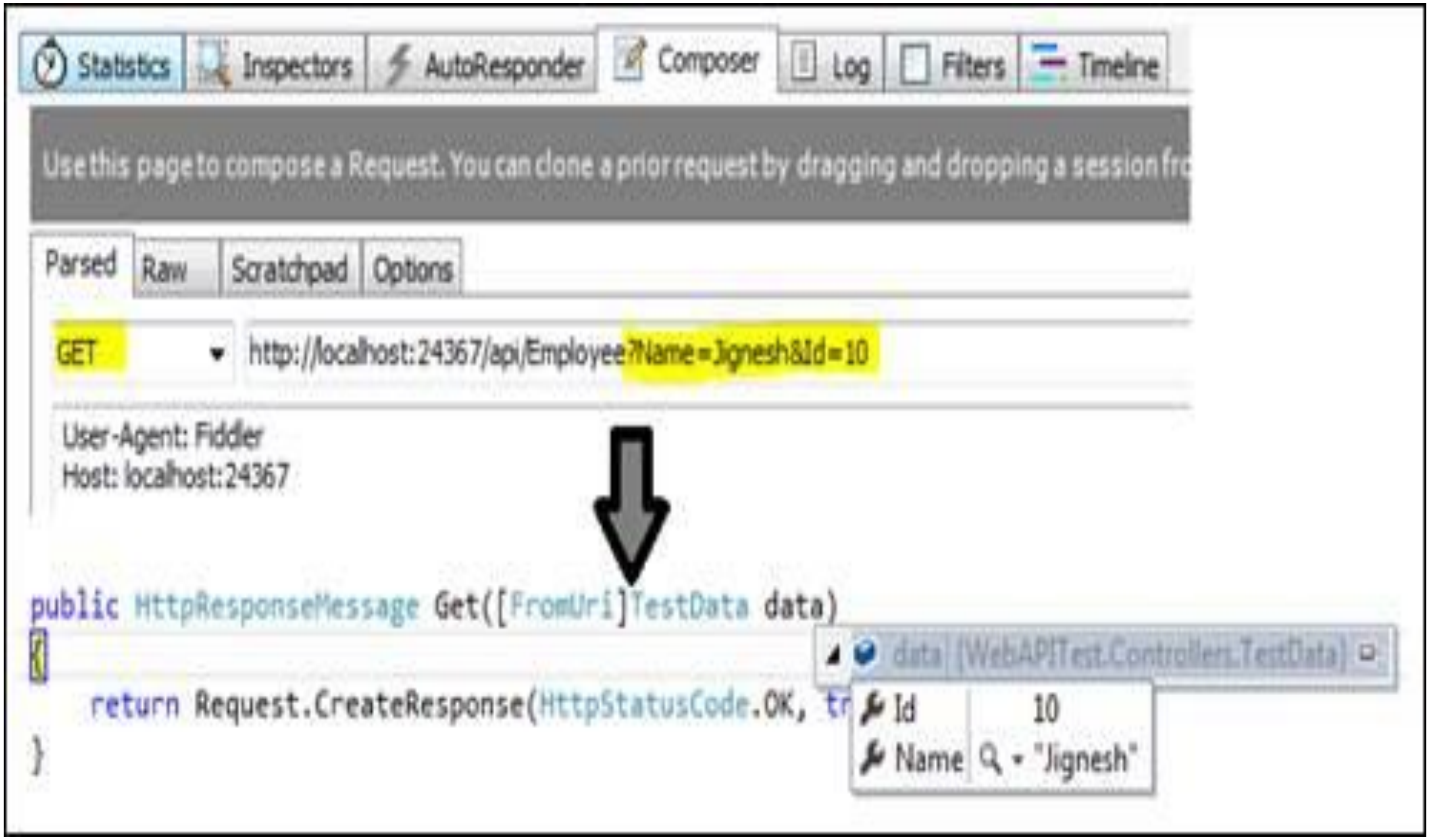
```
public HttpResponseMessage Get([FromUri] TestData data)
{……
    return Request.CreateResponse(HttpStatusCode.OK, true);
}
```

Client can put the value of ID and Name property of TestData class in the query string. Web API uses them to construct TestData class

.

Example:

URI: http://localhost:24367/api/Employee?Name=Jignesh&Id=10



Use this page to compose a Request. You can clone a prior request by dragging and dropping a session fro

GET http://localhost:24367/api/Employee?Name=Jignesh&Id=10

```
User-Agent: Fiddler
Host: localhost:24367
```

```
public HttpResponseMessage Get([FromUri]TestData data)
{
    return Request.CreateResponse(HttpStatusCode.OK, tr
}
```

```
data {WebAPITest.Controllers.TestData}
Id      10
Name    "Jignesh"
```

FromBody Attribute

Binding is a process to set values for the parameters when Web API calls a controller action method. In this article, we learn how to map Web API methods with the different types of the parameters and how to customize the binding process.

Web API tries to get the value from the URI. If a parameter type is a primitive type (Bool, int, Double, log, Timespan, DateTime, Guid, String) Web API tries to read the value of the parameter from the body for complex type, using media-type formatter.

Example:

FromBody Attribute

It forces the Web API to read the parameter value from the requested body. In the following example, I forced the Web API to read the value for simple type from the requested body by using FromBody attribute.

FromBody Attribute(Contd…)

```
[HttpPost]
public HttpResponseMessage Post([FromBody] string name)
{
    ……
    return Request.CreateResponse(HttpStatusCode.OK, true);
}
```

In the above example, I posted the one parameter to the requested body, and Web API used media-type formatter to read the value of the name from the requested body.

To select the media formatter, Web API uses the content type header. In the above example, content type is set to "application/json" and requested body contains a string value, so it binds to the parameter at Web API.

It supports the JSON string, not the JSON object, so only one parameter is allowed to read from the message body.

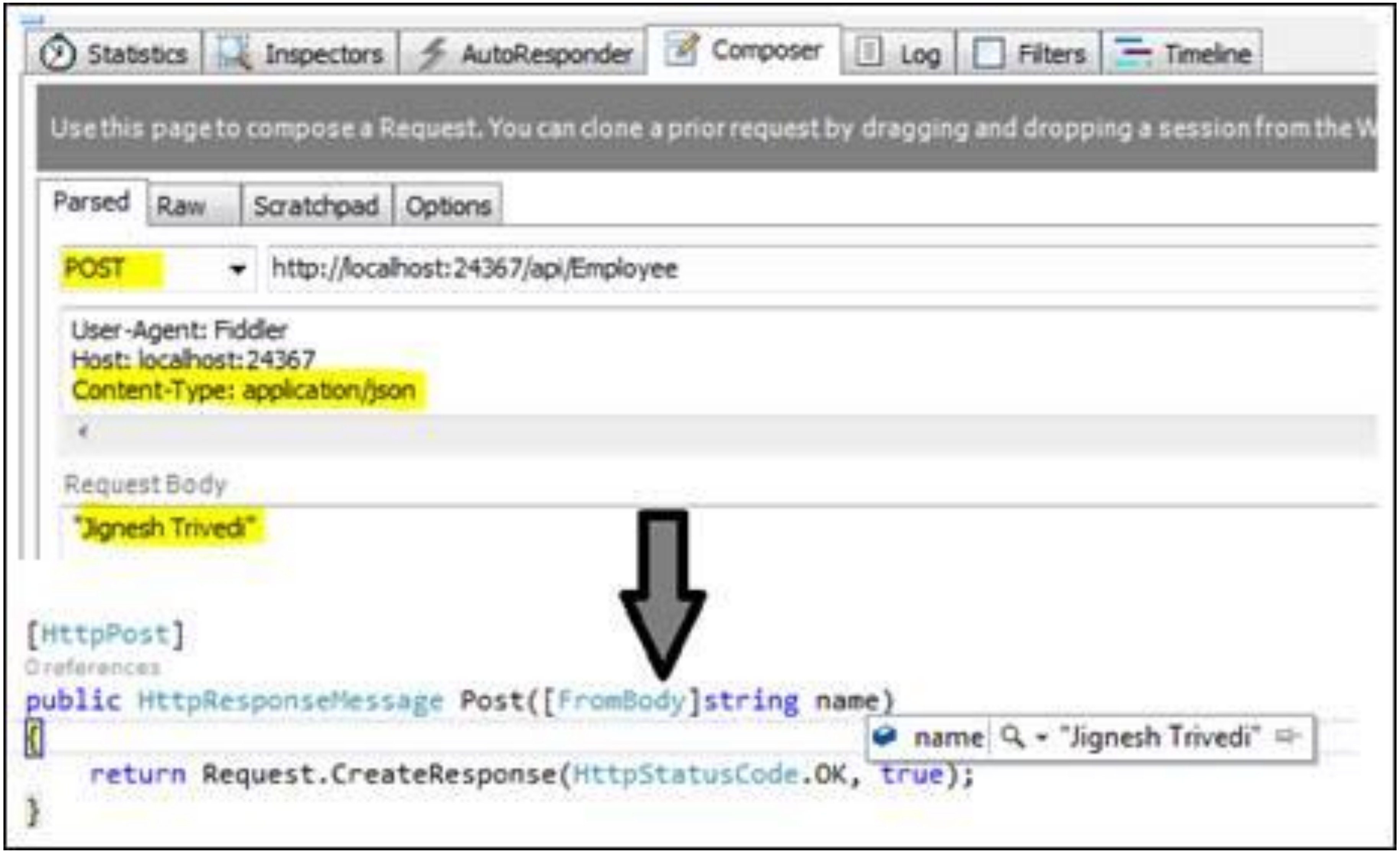
FromBody Attribute(Contd…)

In the above example, I posted the one parameter to the requested body, and Web API used media-type formatter to read the value of the name from the requested body.

To select the media formatter, Web API uses the content type header. In the above example, content type is set to "application/json" and requested body contains a string value, so it binds to the parameter at Web API.

It supports the JSON string, not the JSON object, so only one parameter is allowed to read from the message body.

FromBody Attribute(Contd…)



Use this page to compose a Request. You can clone a prior request by dragging and dropping a session from the W

POST http://localhost:24367/api/Employee

```
User-Agent: Fiddler
Host: localhost:24367
Content-Type: application/json
```

```
"Jignesh Trivedi"
```

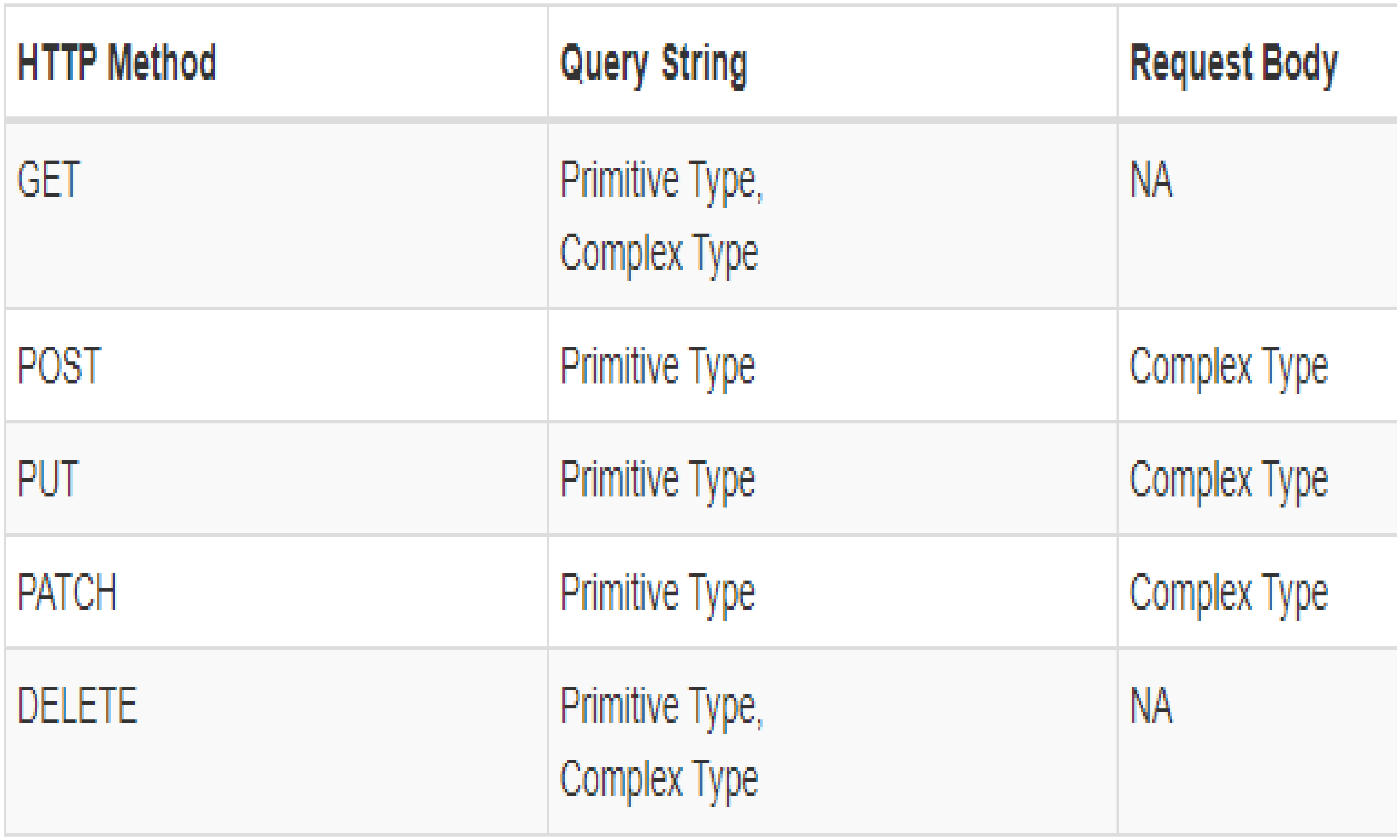
```
[HttpPost]
0 references
public HttpResponseMessage Post([FromBody]string name)
{
    return Request.CreateResponse(HttpStatusCode.OK, true);
}
```

name "Jignesh Trivedi"

Default Parameter Binding in Web API

- By default, if the parameter type is of the primitive type such as int, bool, double, string, GUID, DateTime, decimal or any other type that can be converted from the string type then Web API Framework sets the action method parameter value from the query string.
- And if the action method parameter type is a complex type then Web API Framework tries to get the value from the body of the request and this is the default nature of Parameter Binding in Web API Framework.
-

Default rules for Web API Parameter Binding.



HTTP Method	Query String	Request Body
GET	Primitive Type, Complex Type	NA
POST	Primitive Type	Complex Type
PUT	Primitive Type	Complex Type
PATCH	Primitive Type	Complex Type
DELETE	Primitive Type, Complex Type	NA

Demo

- Demo:- Implementing Parameter Binding in Asp.NET core Web API

Controller Action Return Types in Core web API

ASP.NET Core offers the following options for web API controller action return types:

1. Specific type
2. IActionResult
3. ActionResult<T>

Action Return Type (Contnd..)

Specific type

The simplest action returns a primitive or complex data type (for example, string or a custom object type). Consider the following action, which returns a collection of custom Product objects:

```
[HttpGet]
public List<Product> Get() =>
    _repository.GetProducts();
```

Without known conditions to safeguard against during action execution, returning a specific type could suffice. The preceding action accepts no parameters, so parameter constraints validation isn't needed.

Action Return Type (Contnd..)

IActionResult type

The IActionResult return type is appropriate when multiple ActionResult return types are possible in an action.

The ActionResult types represent various HTTP status codes.

Any non-abstract class deriving from ActionResult qualifies as a valid return type. Some common return types in this category are

- BadRequestResult (400),
- NotFoundResult (404),
- OkObjectResult (200).

- Alternatively, convenience methods in the ControllerBase class can be used to return ActionResult types from an action.
- For example, return BadRequest(); is a shorthand form of return new BadRequestResult();.

Action Return Type (Contnd..)

- Because there are multiple return types and paths in this type of action, liberal use of the [ProducesResponseType] attribute is necessary. This attribute produces more descriptive response details for web API help pages generated by tools like Swagger. [ProducesResponseType] indicates the known types and HTTP status codes to be returned by the action.

Action Return Type (Contnd..)

ActionResult<T> type

ASP.NET Core 2.1 introduced the ActionResult<T> return type for web API controller actions. It enables you to return a type deriving from ActionResult or return a specific type. ActionResult<T> offers the following benefits over the IActionResult type:

The [ProducesResponseType] attribute's Type property can be excluded. For example, [ProducesResponseType(200, Type = typeof(Product))] is simplified to [ProducesResponseType(200)].

The action's expected return type is instead inferred from the T in ActionResult<T>.

Implicit cast operators support the conversion of both T and ActionResult to ActionResult<T>. T converts to ObjectResult, which means return new ObjectResult(T); is simplified to return T;.

Return HTTP Status Code from ASP.NET Core

It is advisable to return the proper HTTP status code in response to a client request.

This helps the client to understand the request’s result and then take corrective measure to handle it.

Proper use of the status codes will help to handle a request’s response in an appropriate way. Out of the box, ASP.NET Core has inbuilt methods for the most common status codes. Like,

- return Ok(); // Http status code 200
- return Created(); // Http status code 201
- return NoContent(); // Http status code 204
- return BadRequest(); // Http status code 400
- return Unauthorized(); // Http status code 401
- return Forbid(); // Http status code 403
- return NotFound(); // Http status code 404

Filters in WebApi Core

- *Filters* in ASP.NET Core allow code to be run before or after specific stages in the request processing pipeline.

Built-in filters handle tasks such as:

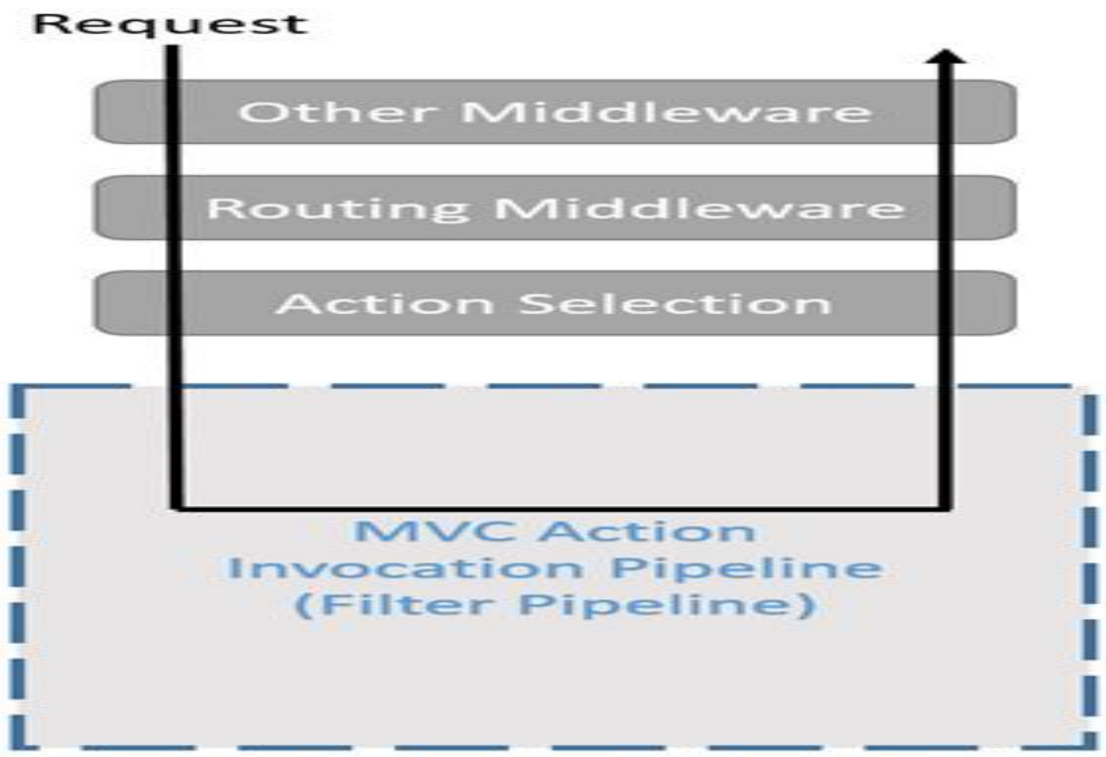
- Authorization (preventing access to resources a user isn't authorized for).

- Response caching (short-circuiting the request pipeline to return a cached response).

- Custom filters can be created to handle cross-cutting concerns. Examples of cross-cutting concerns include error handling, caching, configuration, authorization, and logging. Filters avoid duplicating code. For example, an error handling exception filter could consolidate error handling.

How Filters Work

- Filters run within the *ASP.NET Core action invocation pipeline*, sometimes referred to as the *filter pipeline*. The filter pipeline runs after ASP.NET Core selects the action to execute.



Filter Types

- **Authorization Filters** run first and are used to determine whether the user is authorized for the request. Authorization filters short-circuit the pipeline if the request is not authorized.

- **Resource Filters**:
  - Run after authorization.
  - OnResourceExecuting runs code before the rest of the filter pipeline.
  - For example, OnResourceExecuting runs code before model binding.
  - OnResourceExecuted runs code after the rest of the pipeline has completed.

- **Action Filters**:
  - Run code immediately before and after an action method is called.
  - Can change the arguments passed into an action.
  - Can change the result returned from the action.
  - Are not supported in Razor Pages.

Filter Types (Contnd..)

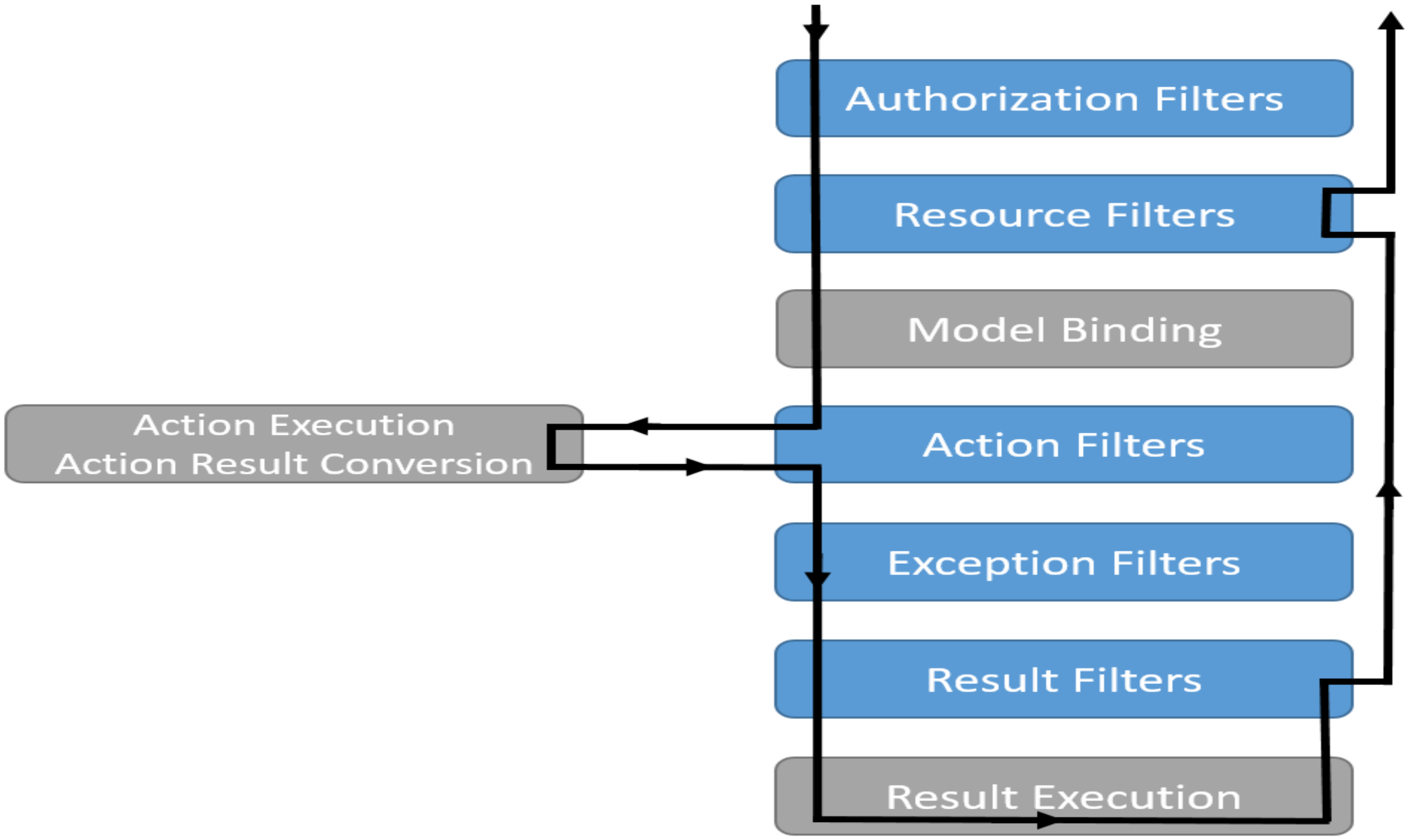
Exception Filters apply global policies to unhandled exceptions that occur before the response body has been written to.

Result Filters run code immediately before and after the execution of action results. They run only when the action method has executed successfully. They are useful for logic that must surround view or formatter execution.

The following diagram shows how filter types interact in the filter pipeline.

Filter Types (Contnd..)

The following diagram shows how filter types interact in the filter pipeline.

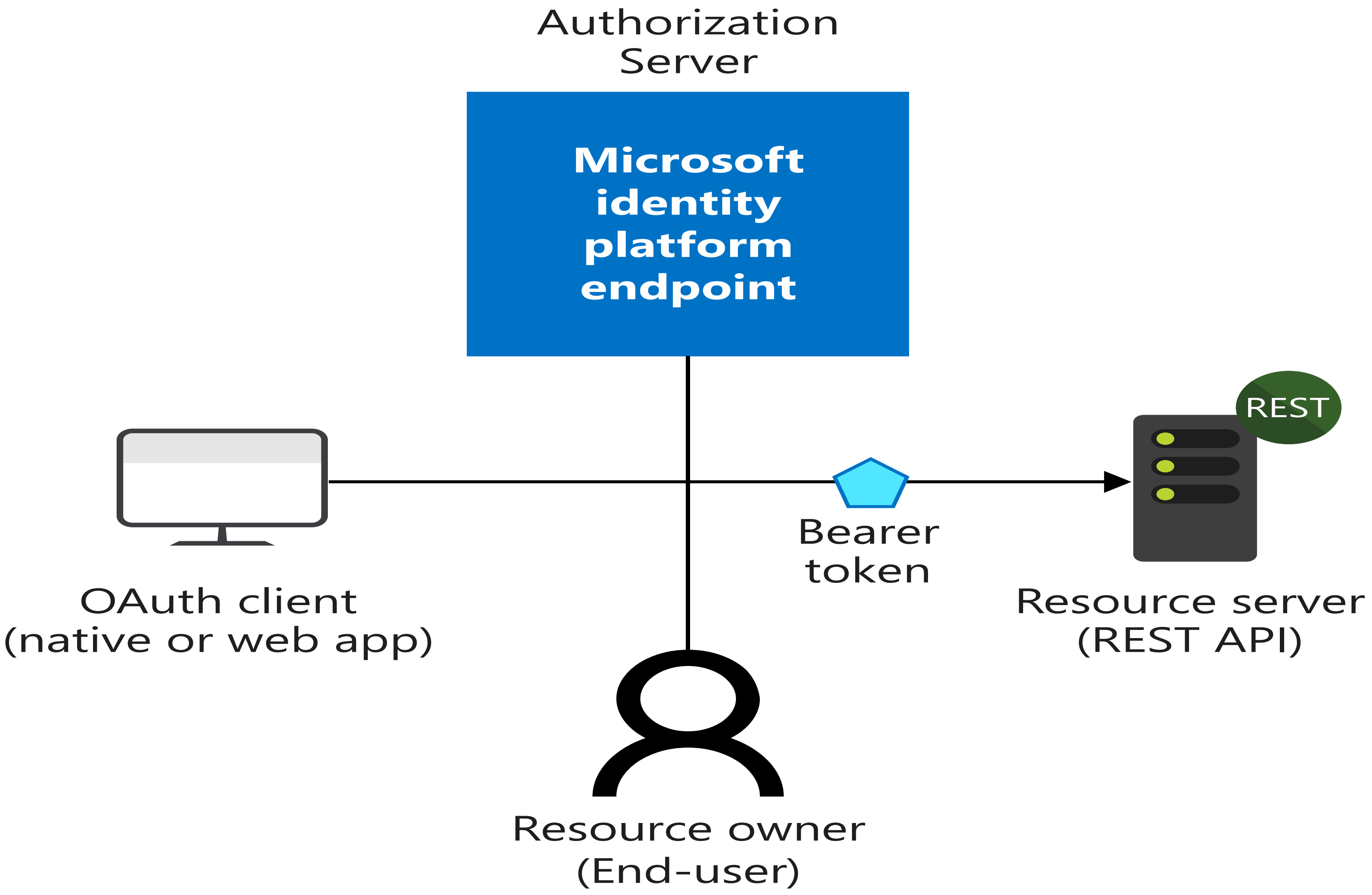


OAuth 2.0 and OpenID Connect

- The Microsoft identity platform endpoint for identity-as-a-service implements authentication and authorization with the industry standard protocols OpenID Connect (OIDC) and OAuth 2.0, respectively.
- While the service is standards-compliant, there can be subtle differences between any two implementations of these protocols.

- The information here will be useful if you choose to write your code by directly sending and handling HTTP requests or use a third-party open-source library, rather than using one of our open-source libraries.

OAuth 2.0 and OpenID Connect (Contnd..)


Authorization Server
Microsoft identity platform endpoint
OAuth client (native or web app)
Bearer token
REST
Resource server (REST API)
Resource owner (End-user)

OAuth 2.0 and OpenID Connect (Contnd..)

- The **Authorization Server** is the Microsoft identity platform endpoint and responsible for ensuring the user's identity, granting and revoking access to resources, and issuing tokens. The authorization server is also known as the identity provider - it securely handles anything to do with the user's information, their access, and the trust relationships between parties in a flow.
- The **Resource Owner** is typically the end user. It's the party that owns the data and has the power to allow clients to access that data or resource.
- The **OAuth Client** is your app, identified by its application ID. The OAuth client is usually the party that the end user interacts with, and it requests tokens from the authorization server. The client must be granted permission to access the resource by the resource owner.
- The **Resource Server** is where the resource or data resides. It trusts the Authorization Server to securely authenticate and authorize the OAuth Client, and uses Bearer access tokens to ensure that access to a resource can be granted.

OAuth

- OAuth is an authorization framework that allows a client app(could be third party application) to retrieve information from another system using a token which is valid for a limited time.

- Now, there could be a case where the application users authorize the client app to retrieve information on their behalf.
- For example, you (user) are logging into Yelp.com (third party app) using your google account.

- In this case, OAuth provides a way for ***delegated authorization*** which means OAuth provides a way for the client app (Yelp.com) to gain access to a user's protected resources without asking the user (your) to provide its login credentials to the app.

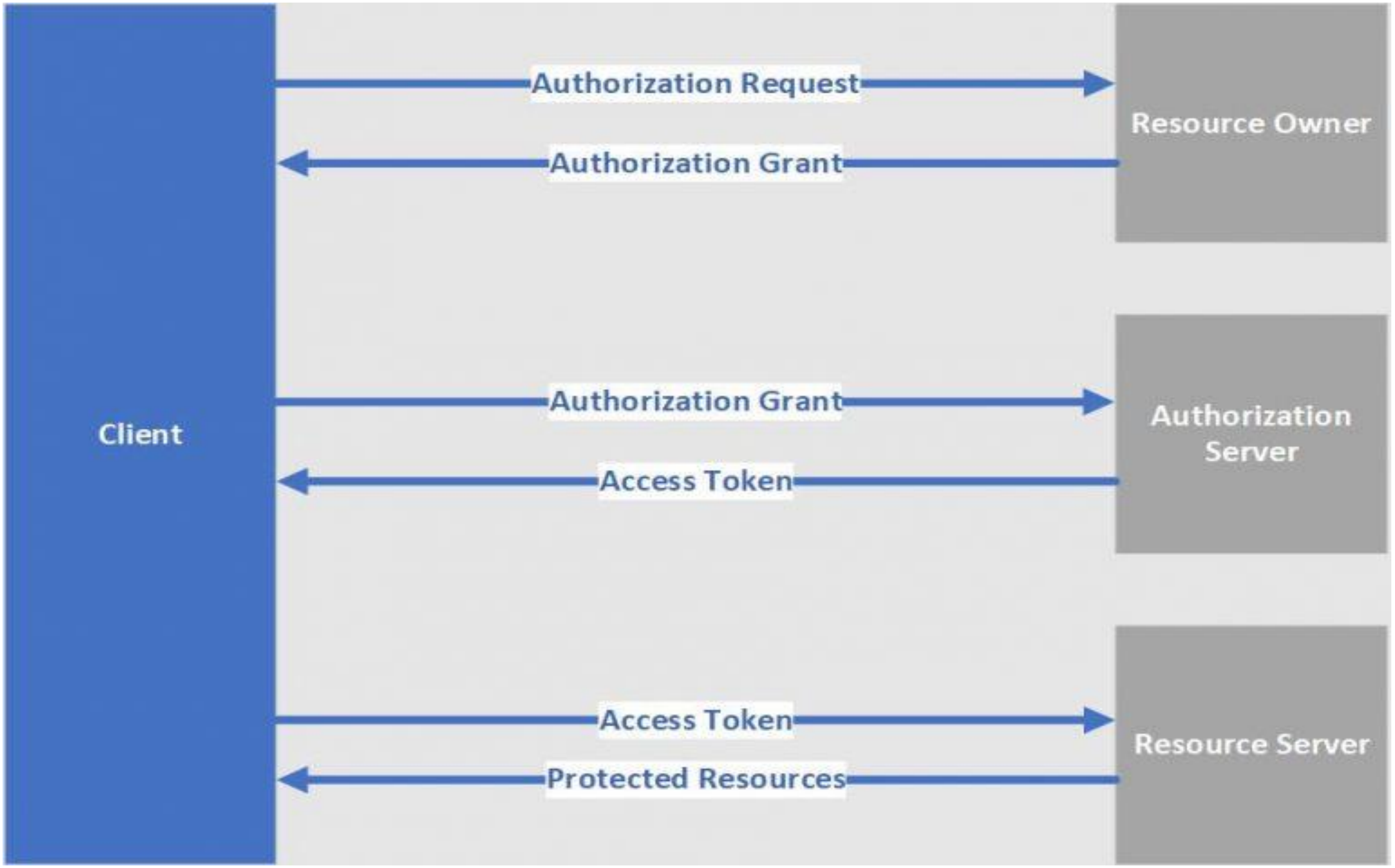
Authentication vs Authorization

- When a connection is successful with client and server, Authentication provides the information about the client, like who the client is? So, in Authentication, the server verifies the client credentials which can be passed in different forms in the HTTP request. If the client failed to authenticate, the server returns HTTP 401 to the client in the response.

- Whereas, asking for Authorization details to the Client, you are checking if the Client is authorized to make the API call. So, in the Authorization check, we verify the client’s access by decoding some kind of token passed in the HTTP request. If client is not authorized to make the API call, server returns HTTP 403 in the response.

- It is not necessary you need to check the Authentication of the client before checking the Authorization details. In most cases, only Authorization is check works just fine with REST API

OAuth Protocol Flow



OAuth 2.0 terminology

- **Resource Owner:** End-user in the process. When you log in to Yelp.com using your google account and you are granting access to yelp to access your basic profile, in this case, “You” are the resource owner. Basically, a resource owner is someone who is capable of granting access to an application on his/her behalf.
- **Client:** This is the client app. In the above example, Yelp is the client and the resource owner (“You”) must grant access to the Client(“Yelp”). When Client registers in the developer portal, a client id/secret is issued by the Authorization Server.
- **Authorization Server:** The server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization. In typical use cases, Octa, Ping or any IDP works as Authorization Server.
- **Resource Server:** The server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens. This is the server where your APIs(protected resources) are hosted.

OAuth 2.0 terminology (Contnd…)

- **Authorization Grant:** An authorization grant is a credential representing the resource owner's authorization (to access its protected resources) used by the client to obtain an access token.
- This specification defines four grant types; authorization code, implicit, resource owner password credentials, and client credentials. Check out the next table for deep dive into different grant types and their use.

- **Redirect URI:** This is the client application's URI where the Authorization Server will send the authorization code. Sometimes, this called callback URI.

- **Access Token:** Access tokens are credentials used to access protected resources. An access token is a string representing an authorization issued to the client. The string is usually opaque to the client. Tokens represent specific scopes and durations of access, granted by the resource owner, and enforced by the resource server and authorization server.

Content Negotiation

- Content negotiation occurs when the client specifies an Accept header. The default format used by ASP.NET Core is **JSON**. Content negotiation is:

  Implemented by **ObjectResult**.

- Built into the status code-specific action results returned from the helper methods. The action results helper methods are based on ObjectResult.
- When a model type is returned, the return type is ObjectResult.
- By default, ASP.NET Core supports **application/json**, **text/json**, and **text/plain** media types. Tools such as **Fiddler** or **Postman** can set the Accept request header to specify the return format. When the Accept header contains a type the server supports, that type is returned.

The Accept Header

Content negotiation takes place when an **Accept header** appears in the request. When a request contains an accept header, ASP.NET Core:

- Enumerates the media types in the accept header in preference order.
- Tries to find a formatter that can produce a response in one of the formats specified.

If no formatter is found that can satisfy the client's request, ASP.NET
Core:

- Returns **406 Not Acceptable** if MvcOptions has been set, or -
- Tries to find the first formatter that can produce a response.

Browsers and Content Negotiation

Unlike typical API clients, web browsers supply Accept headers. Web browser specify many formats, including wildcards. By default, when the framework detects that the request is coming from a browser:

- The Accept header is ignored.
- The content is returned in JSON, unless otherwise configured.

This provides a more consistent experience across browsers when consuming APIs.

Demo

- Demo:- Implementing Content Negotiaition in Web API

Summary

- In this lesson you have learnt about:
  - Configuring WebApi core
  - Web API Parameter Biding
  - Action Return Type
  - WebApi Filters
  - Security and Roles (OpenId / OAuth)
  - Content Negotiation