

# Test the REST

Testing RESTful web services using C# and RestSharp

An open source workshop by ...

# What are we going to do?

- \_RESTful APIs

- \_RestSharp

- \_Hands-on exercises

# Preparation

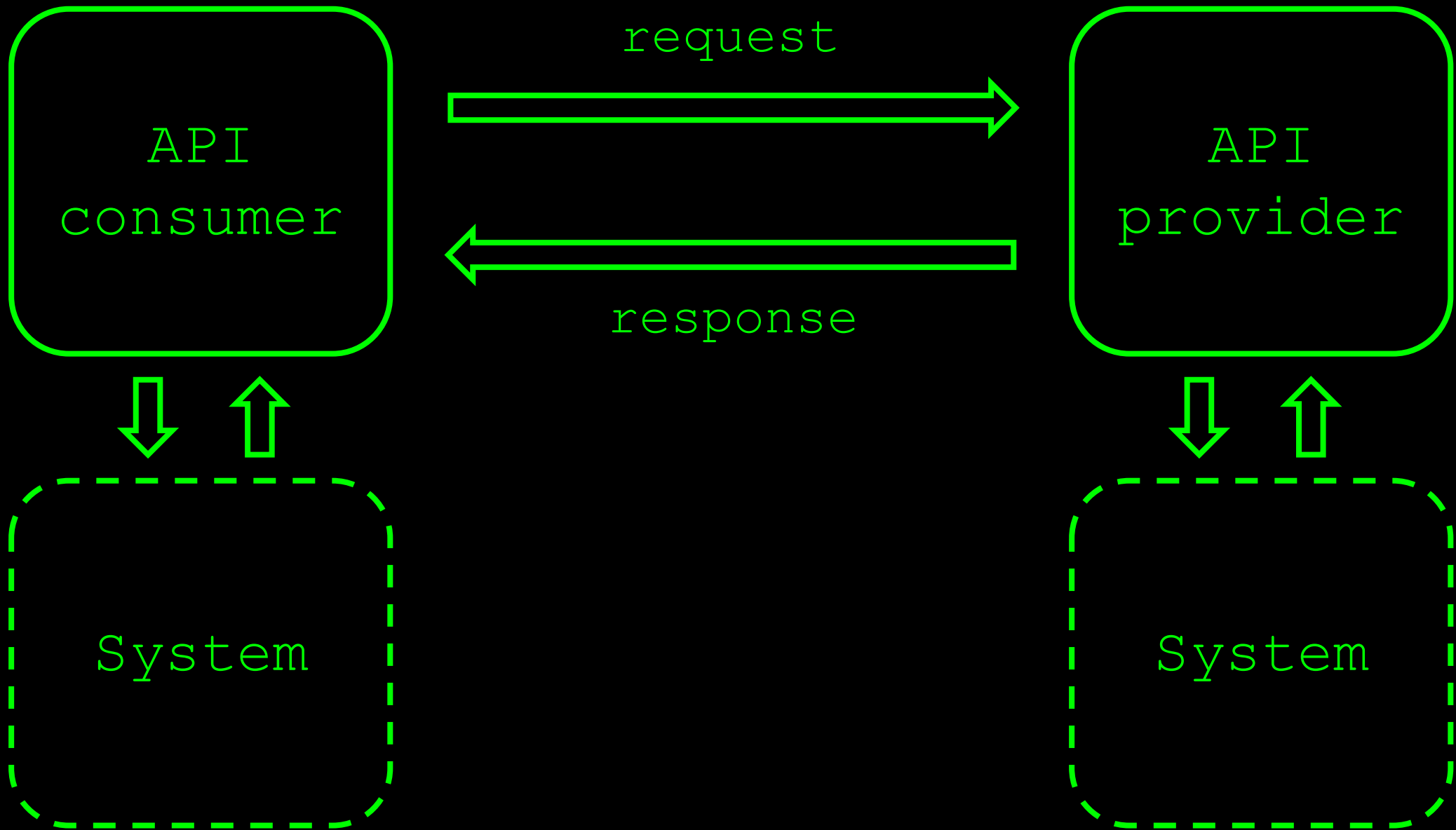
- \_Install .NET 6

- \_Install Visual Studio 2022 (or any other IDE)

- \_Import project into your IDE

  - \_<https://github.com/basdijkstra/restsharp-workshop>

(RESTful) APIs are  
commonly used to  
exchange data between  
two parties



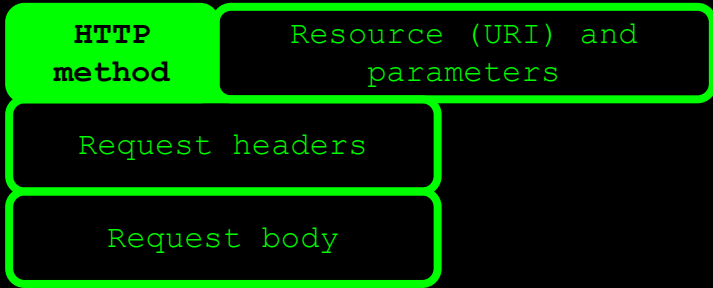
# A REST API request

HTTP method

Resource (URI) and parameters

Request headers

Request body



# HTTP methods

\_GET, POST, PUT, PATCH, DELETE, OPTIONS, ...

\_CRUD operations on data

POST Create

GET Read

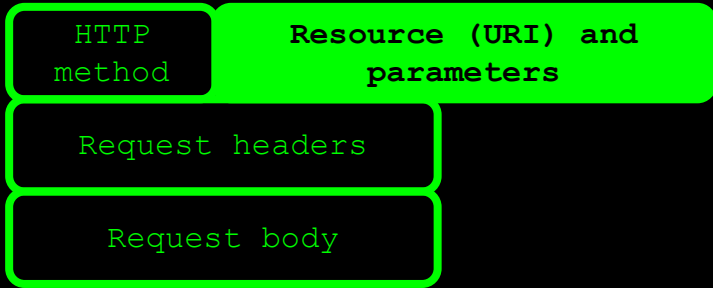
PUT / PATCH Update

DELETE Delete

...

...

\_Conventions, not standards!



# Resources and parameters

\_Uniform Resource Identifier

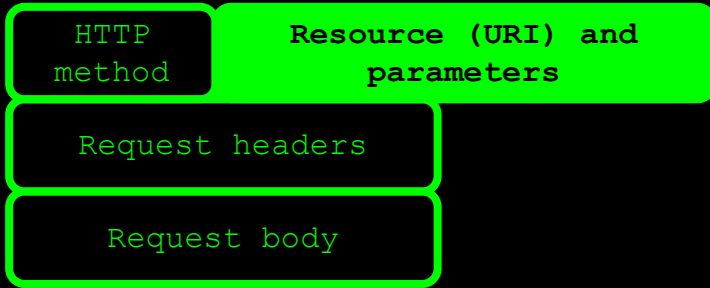
\_Uniquely identifies the resource to operate on

\_Can contain parameters

\_Query parameters

\_Path parameters





# Resources and parameters

## \_ Path parameters

\_ `http://api.zippopotam.us/us/90210`

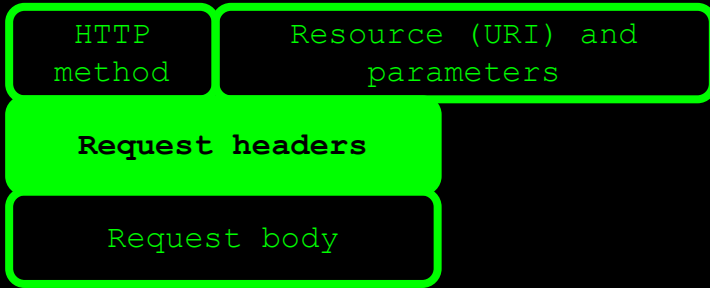
\_ `http://api.zippopotam.us/ca/B2A`

## \_ Query parameters

\_ `http://md5.jsontest.com/?text=testcaseOne`

\_ `http://md5.jsontest.com/?text=testcaseTwo`

\_ There is no official standard!



# Request headers

- \_ Key-value pairs

- \_ Can contain metadata about the request body

- \_ Content-Type (what data format is the request body in?)

- \_ Accept (what data format would I like the response body to be in?)

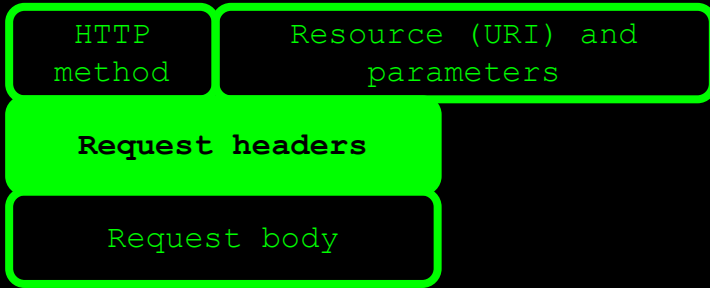
- \_ ...

- \_ Can contain session and authorization data

- \_ Cookies

- \_ Authorization tokens

- \_ ...



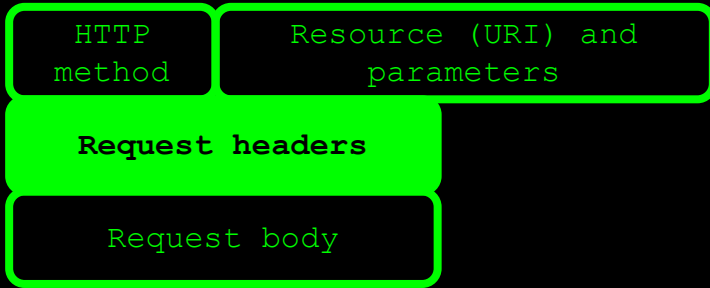
# Authorization: Basic

\_Username and password sent with every request

\_Base64 encoded (not really secure!)

\_Ex: username = aladdin and password = opensesame

*Authorization: Basic YWxhZGRpbjpvcGVuc2VzYW1l*



# Authorization: Bearer

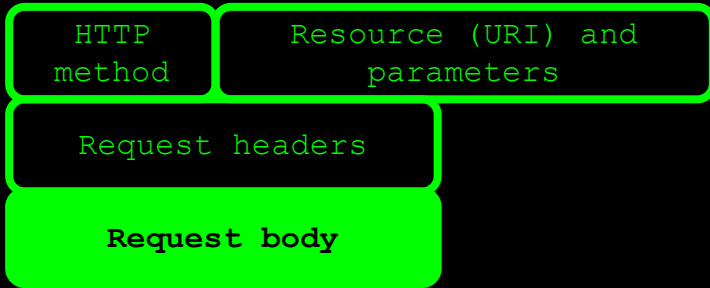
\_Token with limited validity is obtained first

\_Token is then sent with all subsequent requests

\_Most common mechanism is OAuth(2)

\_JWT is a common token format

*Authorization: Bearer RsT50jbzRn430zqMLgV3Ia*



# Request body

- Data to be sent to the provider

- REST does not prescribe a specific data format

- Most common:

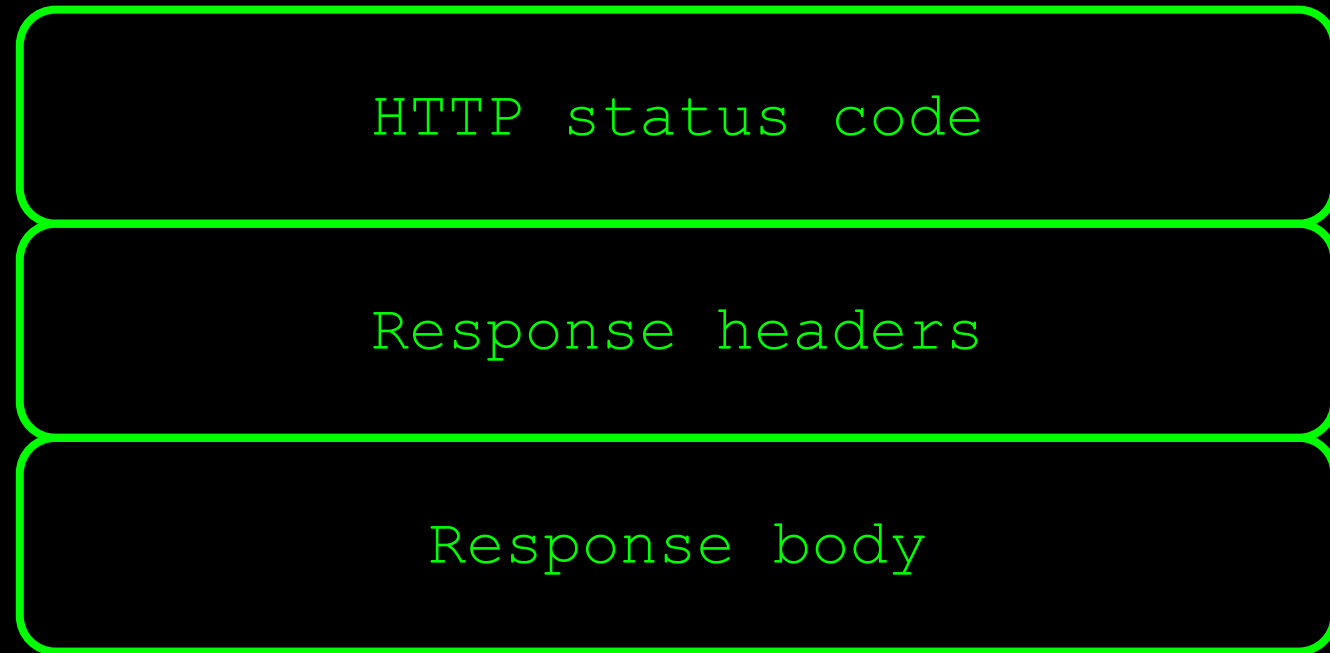
- JSON

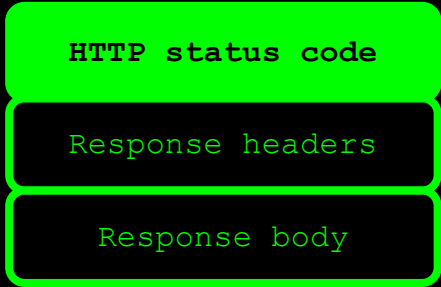
- XML

- Plain text

- Other data formats can be sent using REST, too

# A REST API response





# HTTP status code

— Indicates result of request processing by provider

— Five different categories

— 1XX	Informational	100 Continue
— 2XX	Success	200 OK
— 3XX	Redirection	301 Moved Permanently
— 4XX	Client errors	400 Bad Request
— 5XX	Server errors	503 Service Unavailable

HTTP status code

**Response headers**

Response body

# Response headers

- \_Key-value pairs

- \_Can contain metadata about the response body

  - \_Content-Type (what data format is the response body in?)

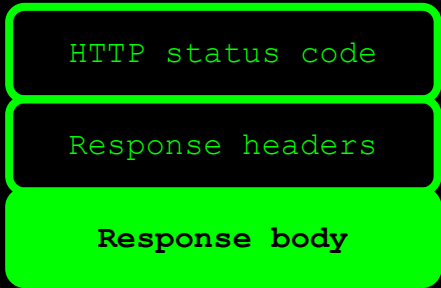
  - \_Content-Length (how many bytes in the response body?)

- \_Can contain provider-specific data

  - \_Caching-related headers

  - \_Information about the server type





# Response body

— Data returned by the provider

— REST does not prescribe a specific data format

— Most common:

— JSON

— XML

— Plain text

— Other data formats can be sent using REST, too

# An example

\_GET http://ergast.com/api/f1/2018/drivers.json

```
{
  - MRData: {
    xmlns: "http://ergast.com/mrd/1.4",
    series: "f1",
    url: "http://ergast.com/api/f1/2018/drivers.json",
    limit: "30",
    offset: "0",
    total: "20",
    - DriverTable: {
      season: "2018",
      - Drivers: [
        - {
          driverId: "alonso",
          permanentNumber: "14",
          code: "ALO",
          url: "http://en.wikipedia.org/wiki/Fernando_Alonso",
          givenName: "Fernando",
          familyName: "Alonso",
          dateOfBirth: "1981-07-29",
          nationality: "Spanish"
        },
        - {
          driverId: "bottas",
          permanentNumber: "77",
          code: "BOT"
```

×	Headers	Preview	Response	Timing
▼ General				
Request URL: http://ergast.com/api/f1/2018/drivers.json				
Request Method: GET				
Status Code: 200 OK				
Remote Address: 81.27.85.129:80				
Referrer Policy: no-referrer-when-downgrade				
▼ Response Headers <a href="#">view source</a>				
Access-Control-Allow-Origin: *				
Connection: close				
Content-Length: 4494				
Content-Type: application/json; charset=utf-8				
Date: Tue, 29 Jan 2019 09:39:19 GMT				
Server: Apache/2.2.15 (CentOS)				
X-Powered-By: PHP/5.3.3				
▼ Request Headers <a href="#">view source</a>				
Accept: text/html,application/xhtml+xml,application/xml				

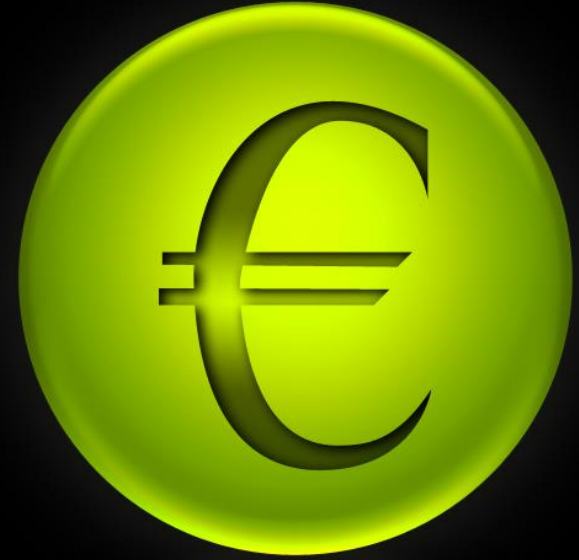
# Where are APIs used?



Mobile



Internet of  
Things



API economy

# Where are APIs used?



Web  
applications



Microservices  
architectures

# Why I ♥ testing at the API level

- \_Tests run much faster than UI-driven tests

- \_Tests are much more stable than UI-driven tests

- \_Tests have a broader scope than unit tests

- \_Business logic is often exposed at the API level

# Tools for testing RESTful APIs

## \_Free / open source

- \_ Postman
- \_ SoapUI
- \_ Code libraries like REST Assured, RestSharp, requests
- \_ ...

## \_Commercial

- \_ Parasoft SOAtest
- \_ SoapUI Pro
- \_ ...

## \_Build your own (using HTTP libraries for your language of choice)

# RestSharp

\_C# library for writing tests for RESTful APIs

\_Removes the need for a lot of boilerplate code

\_Works with all common unit testing frameworks

\_NUnit, MSTest, xUnit

\_<https://restsharp.dev/>

# Configuring RestSharp

\_Install as a NuGet package



# Hello, World!

```
// The base URL for our example tests
private const string BASE_URL = "http://jsonplaceholder.typicode.com";

// The RestSharp client we'll use to make our requests
private RestClient client;

[OneTimeSetUp]
0 references
public void SetupRestSharpClient()
{
    client = new RestClient(BASE_URL);
}

[Test]
0 references
public async Task GetDataForUser1_CheckStatusCode_ShouldBeHttpOK()
{
    RestRequest request = new RestRequest("/users/1", Method.Get);
    RestResponse response = await client.ExecuteAsync(request);
    Assert.That(response.StatusCode, Is.EqualTo(HttpStatusCode.OK));
}
```

Create a RestClient that performs the HTTP calls

Initialize the client with a base URL (and potential other common properties such as headers, etc.)

We're using NUnit here (could also be MSTest, xUnit, ...)

Create a request using an endpoint and the HTTP method to be used

Execute the HTTP call (async!)

Check the response HTTP status code

# Checking status code as an int

[Test]

0 references

```
public async Task GetDataForUser1_CheckStatusCode_ShouldBeHttp200()
{
    RestRequest request = new RestRequest("/users/1", Method.Get);

    RestResponse response = await client.ExecuteAsync(request);

    Assert.That((int)response.StatusCode, Is.EqualTo(200));
}
```

You can cast the HttpStatusCode enum value to an integer if you prefer to do that / think that this is easier to read

# Checking response content type

[Test]

0 references

```
public async Task GetDataForUser2_CheckContentType_ShouldBeApplicationJson()
{
    RestRequest request = new RestRequest("/users/2", Method.Get);

    RestResponse response = await client.ExecuteAsync(request);

    Assert.That(response.ContentType, Does.Contain("application/json"));
}
```

The ContentType property of the RestResponse object contains the response content type (application/json, application/xml, ...)

# Checking other header values

```
[Test]
0 references
public async Task GetDataForUser3_CheckServerHeader_ShouldBeCloudflare()
{
    RestRequest request = new RestRequest("/users/3", Method.Get);

    RestResponse response = await client.ExecuteAsync(request);

    string serverHeaderValue = response.Headers
        .Where(x => x.Name.Equals("Server"))
        .Select(x => x.Value.ToString())
        .FirstOrDefault();

    Assert.That(serverHeaderValue, Is.EqualTo("cloudflare"));
}
```

The Headers property of the RestResponse object is a collection of all response headers.

LINQ queries are very useful here to select the header(s) you're looking for.

# Checking response body values

[Test]

✓ | 0 references

```
public async Task GetDataForUser4_CheckName_ShouldBePatriciaLebsack()
```

```
{
```

```
    RestRequest request = new RestRequest("/users/4", Method.Get);
```

```
    RestResponse response = await client.ExecuteAsync(request);
```

```
    JObject responseData = JObject.Parse(response.Content);
```

```
    Assert.That(responseData.SelectToken("name").ToString(), Is.EqualTo("Patricia Lebsack"));
```

```
}
```

First, parse the response Content property (a string) to a JObject

Then, use `SelectToken()` to retrieve a specific JSON element value from the JSON structure and convert it to a string to assert on its value

# Checking response body values

[Test]

✓ | 0 references

```
public async Task GetDataForUser5_CheckCompanyName_ShouldBeKeeblerLLC()
{
    RestRequest request = new RestRequest("/users/5", Method.Get);

    RestResponse response = await client.ExecuteAsync(request);

    JObject responseData = JObject.Parse(response.Content);

    Assert.That(responseData.SelectToken("company.name").ToString(), Is.EqualTo("Keebler LLC"));
}
```

The argument to `SelectToken` can be a string path to a specific element, or even a JSONPath query. See <https://www.newtonsoft.com/json/help/html/SelectToken.htm> for more details

# Our API under test

\_(Simulation of) an online banking API

\_Customer data (GET, POST)

\_Account data (POST, GET)

\_RESTful API



# Demo

- \_How to use the test suite
  - \_Executing your tests
  - \_Reviewing test results



# Now it's your turn!

- \_Exercises > Exercises01.cs

- \_Simple checks

- \_Verifying status codes and header values
  - \_Verifying JSON response body elements

- \_Answers are in Answers > Answers01.cs

- \_Examples are in Examples > Examples01.cs

# Parameters in RESTful web services

## \_Path parameters

\_ `http://api.zippopotam.us/us/90210`

\_ `http://api.zippopotam.us/ca/B2A`

## \_Query parameters

\_ `http://md5.jsontest.com/?text=testcaseOne`

\_ `http://md5.jsontest.com/?text=testcaseTwo`

\_There is no official standard!

# Using path parameters

— Straightforward string interpolation works fine

```
public async Task GetDataForUser_CheckName_ShouldEqualExpectedName_UsingTestCase  
(int userId, string expectedName)  
{  
    RestRequest request = new RestRequest($"users/{userId}", Method.Get);
```

— Alternatively, you can make the path parameter usage more explicit by using AddUrlSegment()

```
public async Task GetDataForUser_CheckName_ShouldEqualExpectedName_UsingTestCase_Explicit  
(int userId, string expectedName)  
{  
    RestRequest request = new RestRequest("/users/{userId}", Method.Get);  
    request.AddUrlSegment("userId", userId);
```

Exchange data between consumer and provider

GET to retrieve data from provider, POST to send data to provider, ...

# APIs are all about data

Business logic and calculations often exposed through APIs

Run the same test more than once...

... for different combinations of input and  
expected output values

# Parameterized testing

More efficient to do this at the API level...

... as compared to doing this at the UI level

This is more of a  
unit testing  
framework feature  
than a feature of  
RestSharp!

# 'Feeding' test data to your test

Define test cases using the [TestCase] attribute, and don't forget to include a clear test name

```
[TestCase(1, "Leanne Graham", TestName = "User 1 is Leanne Graham")]
[TestCase(2, "Ervin Howell", TestName = "User 2 is Ervin Howell")]
[TestCase(3, "Clementine Bauch", TestName = "User 3 is Clementine Bauch")]
0 references
public async Task GetDataForUser_CheckName_ShouldEqualExpectedName_UsingTestCase
    (int userId, string expectedName)
{
    Use parameters to pass the test data
    values into the method

    RestRequest request = new RestRequest($"{users/{userId}", Method.Get);

    RestResponse response = await client.ExecuteAsync(request);

    JObject responseData = JObject.Parse(response.Content);

    Assert.That(responseData.SelectToken("name").ToString(), Is.EqualTo(expectedName));
}
```

Use parameters in the test method where required

# Running the data driven test

The test method is run three times, once for each array ('test case') in the test data set

```
[TestCase(1, "Leanne Graham", TestName = "User 1 is Leanne Graham")]
[TestCase(2, "Ervin Howell", TestName = "User 2 is Ervin Howell")]
[TestCase(3, "Clementine Bauch", TestName = "User 3 is Clementine Bauch")]
```

0 references

```
public async Task GetDataForUser_CheckName_ShouldEqualExpectedName_UsingTestCase
(int userId, string expectedName)
{
    RestRequest request = new RestRequest($"/users/{userId}", Method.Get);

    RestResponse response = await client.ExecuteAsync(request);

    JObject responseData = JObject.Parse(response.Content);

    Assert.That(responseData.SelectToken("name").ToString(), Is.EqualTo(expectedName));
}
```

✓	Examples02 (3)	266 ms
✓	User 1 is Leanne Graham	197 ms
✓	User 2 is Ervin Howell	40 ms
✓	User 3 is Clementine Bauch	29 ms



# Alternative: use TestDataSource

```
[Test, TestDataSource("UserData")]  
0 references  
public async Task GetDataForUser_CheckName_ShouldEqualExpectedName_UsingTestDataSource  
    (int userId, string expectedName)  
{  
    RestRequest request = new RestRequest($"/users/{userId}" Method.Get);
```

Use the [TestDataSource] attribute (the test method body is the same as the previous example)

Define a static method with the parameter value passed to [TestDataSource] as its name that returns an object of type IEnumerable<TestCase>

```
private static IEnumerable<TestCaseData> UserData()  
{  
    yield return new TestCaseData(1, "Leanne Graham").  
        SetName("User 1 is Leanne Graham - using TestDataSource");  
    yield return new TestCaseData(2, "Ervin Howell").  
        SetName("User 2 is Ervin Howell - using TestDataSource");  
    yield return new TestCaseData(3, "Clementine Bauch").  
        SetName("User 3 is Clementine Bauch - using TestDataSource");  
}
```

Use *yield* to return new TestCaseData instances one by one. Test names can be set using .SetName() - make sure these are unique!

# Now it's your turn!

\_Exercises > Exercises02.cs

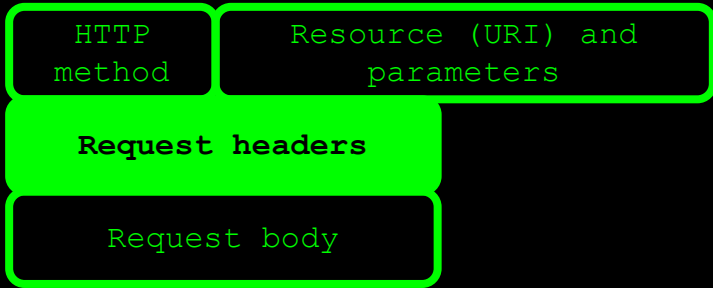
\_Create data driven tests

  \_- Use the [TestCase] attribute

  \_- Use the [TestCaseSource] attribute and a private static  
  method yielding new TestCaseData instances

\_Answers are in Answers > Answers02.cs

\_Examples are in Examples > Examples02.cs



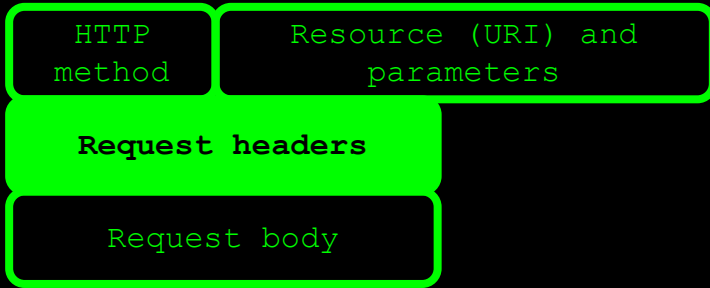
# Authorization: Basic

\_Username and password sent with every request

\_Base64 encoded (not really secure!)

\_Ex: username = aladdin and password = opensesame

*Authorization: Basic YWxhZGRpbjpvcGVuc2VzYW1l*



# Authorization: Bearer

\_Token with limited validity is obtained first

\_Token is then sent with all subsequent requests

\_Most common mechanism is OAuth(2)

\_JWT is a common token format

*Authorization: Bearer RsT50jbzRn430zqMLgV3Ia*

# Authentication in RestSharp

```
// The RestSharp client we'll use to make our requests
private RestClient client;

[OneTimeSetUp]
0 references
public void SetupRestSharpClient()
{
    client = new RestClient(BASE_URL);
}
```

Configure basic authentication for the RestClient

```
client.Authenticator = new HttpBasicAuthenticator("username", "password");
```

Configure OAuth2 (token-based) authentication for the RestClient

```
client.Authenticator = new OAuth2AuthorizationRequestHeaderAuthenticator("access_token", "Bearer");
```

# Now it's your turn!

\_Exercises > Exercises03.cs

\_Use authentication mechanisms

\_Get a token using basic auth

\_Extract token from response and store it

\_Reuse token in OAuth2

\_Answers are in Answers > Answers03.cs

\_Examples are in Examples > Examples03.cs

# (De-)serialization of POCO's

- \_ RestSharp is able to convert C# object instances directly to JSON (and XML) and back
- \_ Useful when dealing with test data objects
  - \_ Creating request body payloads
  - \_ Processing response body payloads

# Example: serialization

\_POCO representing a Post object (think blog posts)

```
public class Post
{
    [JsonProperty("userId")]
    1 reference | 0/1 passing
    public int UserId { get; set; }
    [JsonProperty("title")]
    1 reference | 0/1 passing
    public string Title { get; set; }
    [JsonProperty("body")]
    1 reference | 0/1 passing
    public string Body { get; set; }
}
```

RestSharp respects the [JsonProperty] attribute from Newtonsoft.Json, so you can use these to map C# property names to their JSON element equivalents



# Example: serialization

```
[Test]
0 | 0 references
public async Task PostNewPost_CheckStatusCode_ShouldBeHttpCreated()
{
    Post post = new Post
    {
        UserId = 1,
        Title = "My new post title",
        Body = "This is the body of my new post"
    };

    RestRequest request = new RestRequest("/posts", Method.Post);

    request.AddJsonBody(post);

    RestResponse response = await client.ExecuteAsync(request);

    Assert.That(response.StatusCode, Is.EqualTo(HttpStatusCode.Created));
}
```

Create a new object in your test and assign the desired property values

```
{
    "userId": 1,
    "title": "My new post title",
    "body": "This is the body..."
}
```

Add that object as the request payload using AddJsonBody() and RestSharp handles the rest for you

HTTP 201 (Created) is a typical HTTP status code for a successful POST operation

# Example: deserialization

This tells RestSharp to try and deserialize the response body to an object of type User (which is another POCO like Post from the previous example)

[Test]

0 references

```
public async Task GetDataForUser1_CheckName_ShouldEqualLeanneGraham()
{
    RestRequest request = new RestRequest("/users/1", Method.Get);

    RestResponse<User> response = await client.ExecuteAsync<User>(request);

    User user = response.Data;

    Assert.That(user.Name, Is.EqualTo("Leanne Graham"));
}
```

This extracts the deserialized response body into its own object

You can now refer to specific properties of the POCO like you would do with any other regular C# object

# Now it's your turn!

\_Exercises > Exercises04.cs

\_Practice serialization in sending an Account object

\_Practice deserialization by extracting an API response  
-into a C# object

\_Answers are in Answers > Answers04.cs

\_Examples are in Examples > Examples04.cs

The problem with  
'traditional' REST APIs

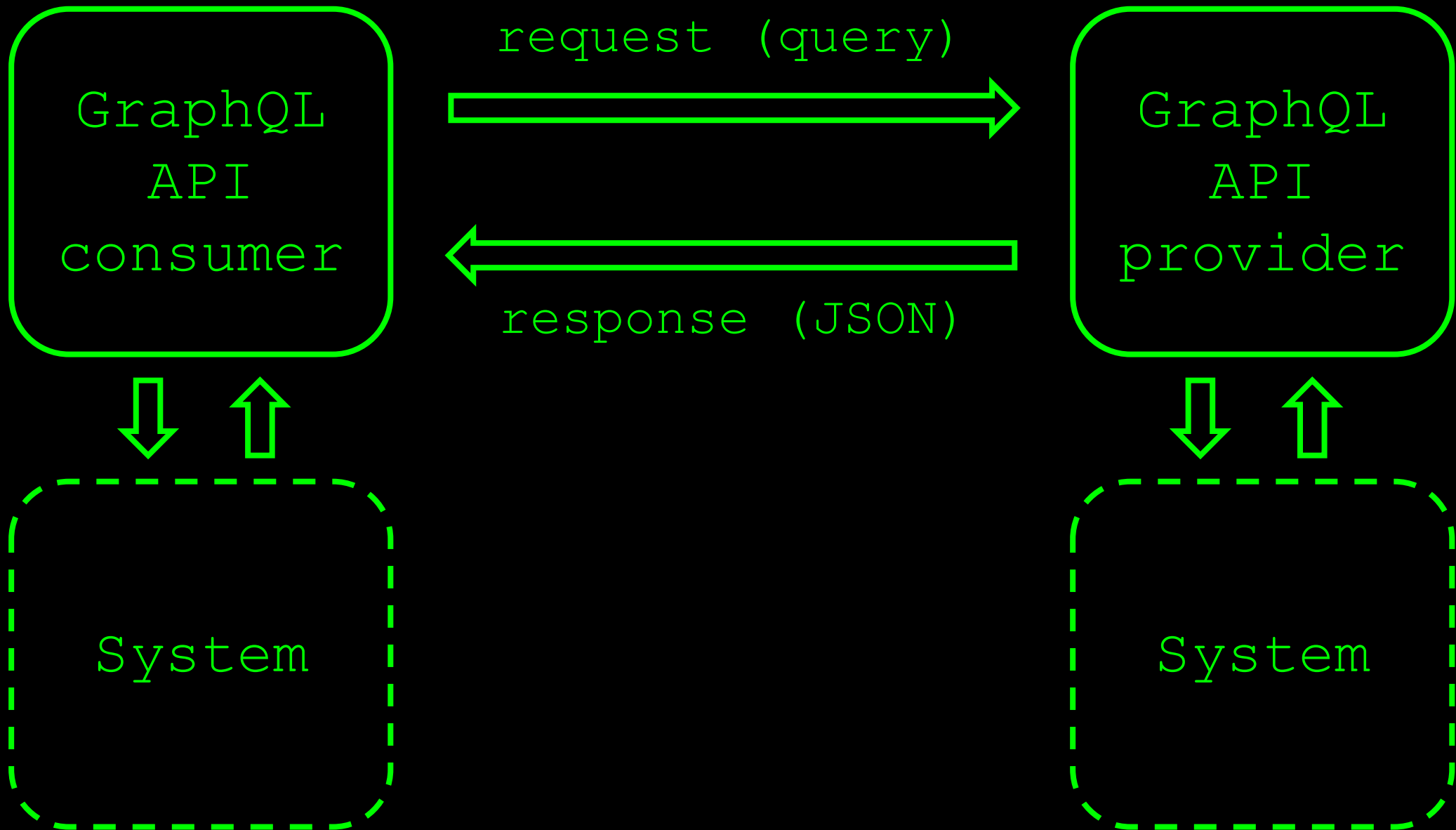
Query language for APIs...

... as well as a runtime to fulfill them

# GraphQL

"Ask for what you need,  
and get exactly that"

<https://graphql.org>



Create a valid GraphQL query...

... and send it in the request body (*query*)

## Sending a GraphQL query

"Ask for what you need,  
and get exactly that"

These are 'regular' REST responses, with...

... an HTTP status code, ...

## GraphQL API responses

... response headers...

... and a JSON response body  
containing the requested data



# Sending a basic GraphQL query

```
string query = @" The query can be a simple (multiline) string
```

```
{
    getCityByName(name: "Amsterdam") {
        weather {
            summary {
                title
            }
        }
    }
}
```

We've seen how to serialize and send the payload in the previous section

```
GraphQLQuery graphQLQuery = new GraphQLQuery
{
    Query = query,
};
```

```
public class GraphQLQuery
{
    [JsonProperty("query")]
    2 references | 1/1 passing
    public string Query { get; set; }
    [JsonProperty("variables")]
    1 reference
    public string Variables { get; set; }
}
```

Using this POCO simplifies creating the GraphQL payload

```
RestRequest request = new RestRequest("/", Method.Post);
request.AddJsonBody(graphQLQuery);

RestResponse response = await client.ExecuteAsync(request);

JObject responseData = JObject.Parse(response.Content);

Assert.That(
    responseData.SelectToken("data.getCityByName.weather.summary.title")
    Is.EqualTo("Clouds")
);
```

A GraphQL API response is plain JSON

# Parameterizing GraphQL queries

```
string query = @"
    query GetWeatherForCity($name: String!)
    {
        getCityByName(name: $name) {
            weather {
                summary {
                    title
                }
            }
        }
    }
";
```

GraphQL queries can be parameterized, too

```
var variables = new
{
    name = "Amsterdam"
};
```

Values for these variables can be sent to a GraphQL API in JSON format, which we're doing here by serializing an anonymous type object

```
GraphQLQuery graphqlQuery = new GraphQLQuery
{
    Query = query,
    Variables = JsonConvert.SerializeObject(variables)
};
```

# A data driven GraphQL test

As we've done with 'regular' REST APIs, we can use this to create a data driven GraphQL test.

This example checks the weather in Amsterdam, Berlin and Rome.

```
[TestCase("Amsterdam", "Clouds", TestName = "In Amsterdam the weather is cloudy")]
[TestCase("Berlin", "Clouds", TestName = "In Berlin the weather is cloudy")]
[TestCase("Rome", "Clear", TestName = "In Rome the weather is clear")]
0 references
public async Task GetWeatherForAmsterdam_CheckSummaryTitle_UsingParameterizedQuery
    (string city, string expectedWeather)
{
    string query = @"
        query GetWeatherForCity($name: String!)
        {
            getCityByName(name: $name) {
                weather {
                    summary {
                        title
                    }
                }
            }
        }
    ";

    var variables = new
    {
        name = city
    };

    GraphQLQuery graphqlQuery = new GraphQLQuery
    {
        Query = query,
        Variables = JsonConvert.SerializeObject(variables)
    },
```

# Now it's your turn!

\_Exercises > Exercises05.cs

\_Work with the SpaceX GraphQL API

- Create and send a fixed (static) GraphQL query and assert on the response
- Create a parameterized GraphQL query and use that in a data-driven GraphQL API test

\_Answers are in Answers > Answers05.cs

\_Examples are in Examples > Examples05.cs



# Contact

Email:      [bas@ontestautomation.com](mailto:bas@ontestautomation.com)

Website:   <https://www.ontestautomation.com/training>

LinkedIn:   <https://www.linkedin.com/in/basdijkstra>