Data-driven systems

Web services, REST and ASP.NET Web API, GraphQL

Zoltán Benedek

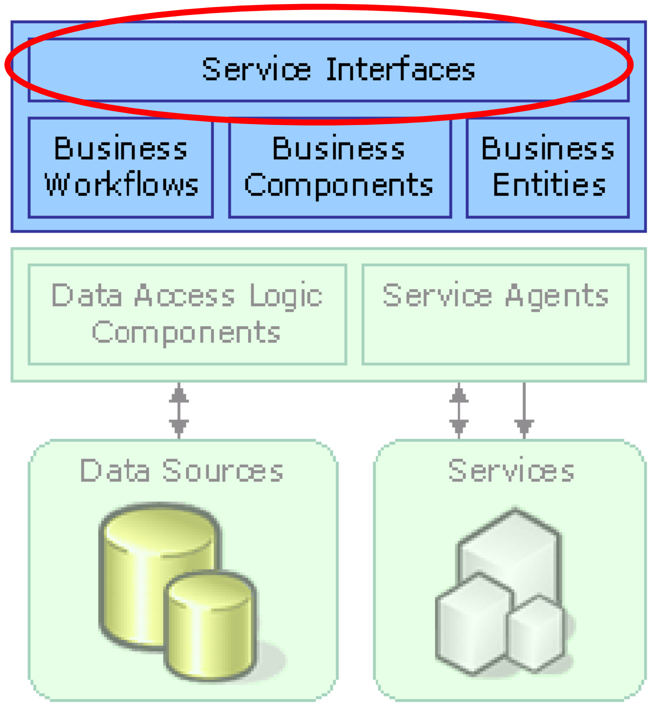
Automatizálási és Alkalmazott Informatikai Tanszék

(Web) services, SOA (Service Oriented Architecture)

What is a service

- Multi-layer architecture
  - Data access – business logic – user interface
- Service interface
- New aspects and expectations
  - Distributed systems
  - Heterogeneous systems

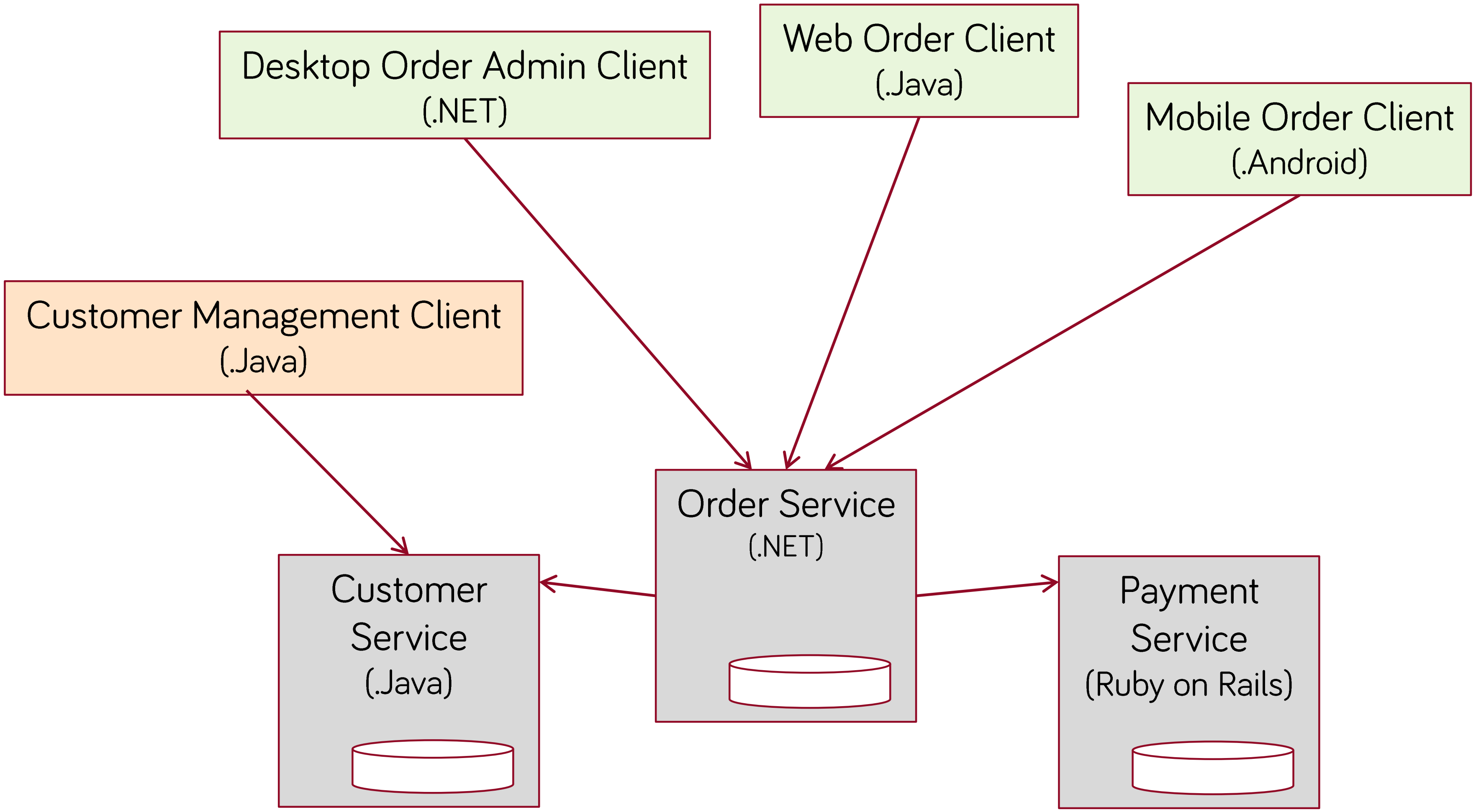
  (cooperation between different platforms)



Service Oriented Architecture - SOA

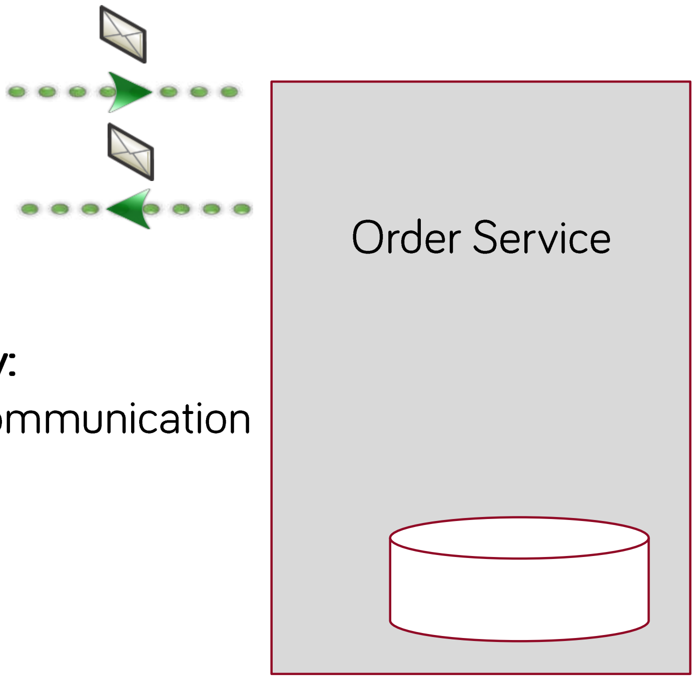
- The system is built-up of services
  - Also in heterogeneous systems (different platforms – Java/.NET/etc.)
  - Web service
  - The client of a service can be
    - A simple client application
    - An other service
- Cooperating services instead of isolated applications

SOA example


Desktop Order Admin Client
(.NET)
Web Order Client
(.Java)
Mobile Order Client
(.Android)
Customer Management Client
(.Java)
Order Service
(.NET)
Customer
Service
(.Java)
Payment
Service
(Ruby on Rails)

The concept of (Web) services

Communication: message-based



Operations:

E.g.:

- New order
- List orders
- Delete an order
- Get the details of on order

Interoperability:
standardized communication

Autonomous units

- Any platform
- Independent versioning
- ...

The implementation is usually layered (DB, DAL, BLL, Service Interface)

The concept of (Web) services

1. A service publishes a set of business functionalities for clients or for other services (contract/interface)

   E.g. save an order; change the state of on order, getting exchange rates, …

2. The communication is message-based
3. Standardized interface and communication
4. Independent versioning, deployment, no technology constraint (interoperability)

SOA - SOAP and REST

- There are two widespread solutions to implement SOA/Web services
  - SOAP (Simple Object Access Protocol)-based services
  - REST (vagy RESTful) services
- Both of them are interop solutions (works between different platforms)
- Another popular alternative is GraphQL, and gRPC (the latter one more in an intranet scenario)

SOAP Web services

SOAP

- The older one
- **Standardized protocol** (SOAP 1.1 and 1.2)
- **XML-based message** (not JSON) that represents a request or a response
- The service has got a standardized **interface description**: WSDL document
- There are standards for many middleware services besides the basic communication
  - WS-Security, WS-Atomic Transaction, WS-ReliableMessaging, etc.
  - **WS* standards**
- **Usually used over HTTP**
- Quite complex

SOAP message example

SOAP kérés (SOAP Envelope, Header and Body)

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>

<soap:Envelope
  xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
  soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">


  <soap:Header> ...

  </soap:Header>


  <soap:Body xmlns:m="http://www.example.org/stock">
    <m:GetStockPrice>
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>

</soap:Envelope>
```

SOAP message example

SOAP response

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>

<soap:Envelope
  xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
  soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

  <soap:Body xmlns:m="http://www.example.org/stock">
    <m:GetStockPriceResponse>
      <m:Price>34.5</m:Price>
    </m:GetStockPriceResponse>
  </soap:Body>

</soap:Envelope>
```

SOAP message

- SOAP XML messages are usually automatically generated from method calls by the framework
- Platforms usually provide high level support to communicate with SOAP messages
  - .NET -> WCF
  - Java -> JAX-WS
- Instead of SOAP messages and we usually define C# /Java interfaces and entity classes (parameters and return types of the operations)
- Sending a soap request is as simple as calling a method (the framework generates the SOAP message)

```
decimal price = stockService.GetStockPrice("IBM");
```

WSDL

- The interface definition for the SOAP service
- WSDL – Web Service Description Language
- XML-based language
  - Defines the operations
    - The schema of the parameters and the return types of the operations
  - And some other information
    - Protocol (e.g. HTTP)
    - Security policy

WSDL example

```
<definitions name="HelloService"
   targetNamespace="http://www.examples.com/wsdl/
             HelloService.wsdl"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"

xmlns:tns="http://www.examples.com/wsdl/HelloService.wsdl"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <message name="SayHelloRequest">
      <part name="firstName" type="xsd:string"/>
   </message>

   <message name="SayHelloResponse">
      <part name="greeting" type="xsd:string"/>
   </message>

   <portType name="Hello_PortType">
      <operation name="sayHello">
         <input message="tns:SayHelloRequest"/>
         <output message="tns:SayHelloResponse"/>
      </operation>
   </portType>

   <binding name="Hello_Binding" type="tns:Hello_PortType">
      <soap:binding style="rpc"
         transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="sayHello">
         <soap:operation soapAction="sayHello"/>
         <input>
            <soap:body

encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
               namespace="urn:examples:helloservice"
               use="encoded"/>
         </input>

         <output>
            <soap:body

encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
               namespace="urn:examples:helloservice"
               use="encoded"/>
         </output>
      </operation>
   </binding>

   <service name="Hello_Service">
      <documentation>WSDL File for
             HelloService</documentation>
      <port binding="tns:Hello_Binding" name="Hello_Port">
         <soap:address
            location="http://www.examples.com/SayHello/" />
      </port>
   </service>
</definitions>
```

The role of WSDL in SOA

- WSDL is hard to read or write. We don’t write it ourselves. Instead:
- The usual workflow
  - Service
    - Define the operations in a platform-specific way, e.g. write a C# or Java interface (together with the entities in the interface)
    - By the help of a tool of the platform generate a WSDL document for the interface (svcutil, Java2wsdl, etc.)
  - Client (it may be an other plaform)
    - Generate C#/Java interface and entities from the WSDL by the help of a plaform-spec tool
    - SOAP messaging will become a simple method call

SOAP error handling

- Standardized error handling
- Concept of **Fault**
  - > **Every operation can define a set of faults it can produce**
  - > Special error response
  - > Similar to exceptions
  - > The SOAP standard precisely specifies how a SOAP fault response message builds up (error code, error message, other descriptive data, etc.)
  - > Platforms usually wrap them to exceptions
  - > Platform specific exceptions cannot be thrown to the client

REST(ful) services

REST – Representational State Transfer

- This is an architecture as well (but not a standard) for distributed applications
- Builds on the HTTP 1.1 protocol, HTTP verbs (GET, POST, PUT, PATCH, DELETE) has got semantic
- Principles: The API defines a set of resources with addresses
  - A resource usually means a data entity (e.g. Task, Product, Todo, Order, etc.).
  - Addressing by URLs, e.g.: **`GET /api/todos/12`**
- Data representation: JSON, XML, … (media type concept)
- Server is stateless - ideally
  - Client sends all the information that is required by the service to perform the operation

REST example

- HTTP request („todo” is the resource)

Getting the todo item with id of 12:

```
GET /api/todos/12
```

List all the todo items

```
GET /api/todos
```

- HTTP response

```
HTTP/1.1 200
OK Content-Type: application/json; charset=utf-8
Server: Microsoft-IIS/10.0
Date: Thu, 18 Jun 2015 20:51:10 GMT Content-Length: 82

[{"ID":1,"Name":“Complete the homework","IsComplete":false}]
```

REST URLs and HTTP verbs

	CRUD	Collection, e.g. http://adatvez.hu/tasks/	One item, e.g. http://autvez.hu/tasks/item23
GET	Read	Returns all the items as resources.	Returns a single item as a resource.
POST	Create	Creates a new item in the resource collection. The new items gets a identifier. The URI of the new item is includes in the location header of the POST response.	Usually not used.
PUT	Update/ Replace	The whole collection is replaces.	Replaces an item in the collection (it may also create it if it doesn't exist).
PATCH	Update/ Modify	Partially update a set of items (rarely used).	Partially updates an item of the collection. The change is sent only.
DELETE	Delete	Deletes the whole collection.	Deletes one item of the collection.

Using HTTP verbs in the right way

- GET
  - No side-effect, e.g. shouldn't modify anything
- PUT, DELETE
  - The operation should be idempotent: the operation can safely be repeated
- POST
  - Not idempotent
- PATCH
  - Relatively new verb (2010), rarely used
  - Not idempotent

    Returns the updated object. Between two PATCH requests someone else may also modify the object so a repeated PATCH may return a different object (PATCH only partially updates). PUT is different as that updates the whole object.

Including data in the request

- URL path
  - Locator type data

    `GET /tasks/23`
- URL query string
  - Filtering, ordering, paging, etc.

    `GET /tasks?sort=priority`
- Request headers & cookie
  - API key
  - Authentication tokens
  - Session ID (though REST is stateless)
- Body
  - Content

Filtering and sorting - example

- Listing tasks with descending priority

```
GET /tasks?sort=-priority
```

- Listing tasks with descending priority + creation time:

```
GET /tasks?sort=-priority,created_at
```

- Listing open tasks

```
GET /tasks?state=open
```

- Filtering and sorting

```
GET /tasks?state=open&sort=-priority,created_at
```

Handling relations

- List the notes of task #23:

  ```
  GET /tasks/23/notes
  ```

  or

  ```
  GET /notes?task=23
  ```

- Get the note #12 of task #23:

  ```
  GET /tasks/23/notes/12
  ```

- Add a new note to task #23:

  ```
  POST /tasks/23/notes
  ```

- Update the note #12 of task #23:

  ```
  PUT /tasks/23/notes/12
  ```

- Delete the note #12 of task #23:

  ```
  DELETE /tasks/23/notes/12
  ```

Error handlig

- Use HTTP status codes
  - 4xx: client errors (the problem is with the client's request)
  - 5xx: server errors (the problem is with the server itself)
- Include a general JSON error structure in the response body that contains additional information about the error

```
{
"code" : 8324,
"message" : "Something bad happened :(", "description" :
"More details about the error here"
}
```

- There is a standard format called „Problem details” (rfc7807)

Important HTTP status codes

- GET

  200 (OK); 404 (Not Found) no item with the given id

- POST

  201 (Created) + URL of the new item in the location header; 409 (Conflict)

- PUT és PATCH

  200 (OK); 204 (No Content); 404 (Not Found)

- DELETE

  200 (OK); 404 (Not Found)

Security

- Use HTTPS
- Standards, such as OAuth2, OIDC (e.g. Facebook, Twitter, etc login integration)
- Make the service stateless
    - Use token instead of SessionID, that contains all the required information (e.g. username, roles, permissions granted, etc.)
    - E.g. JWT (JSON Web Token)

REST advantages

- **Scalability** (stateless)
  - Scales well, popular with microservices
- **Many data representations are supported**: JSON, XML, ... (usually JSON)
- **Simple, easy to consume**
- More platforms are supported that by SOAP services
  - **JavaScript** and **mobil clients**!
  - No heavy XML parsing
- **Better performance**
  - More compact message format, faster parser (JSON vs XML)
  - Better caching solutions (clients may cache the response)

REST vs SOAP

- REST is far more popular than SOAP nowadays
  - „Lightweight”
  - Supports many platforms including mobile and browser
  - Google (e.g: Gmail API), Facebook, Twitter, Amazon, …
- There are very few cases (are there?) when SOAP can be better
  - „Heavyweight” enterprise information systems
    - Formal contract between client and server
    - More security solutions (we only have HTTPS with REST)

REST vs SOAP

SOAP	REST
Standard (protocol)	Architecture
Interop RPC	Resources + CRUD, HTTP verbs

- REST is resource-based and not operation-based:

```
GET /API/books/delete?id=12        ☹
DELETE /API/books/12               ☺
```

REST – to what extent is REST? (important)

- The REST/HTTP „toolset” is quite often used in a not fully RESTful way
  - HTTP GET, POST, stb. + pl. JSON is used
  - But the API does not rigorously follow the "addressable resource" mindset
- Even if we want to be fully RESTful, we are sometimes compelled to make concessions
  - Especially if the service exposes complex business logic (it’s not simple CRUD)
- Still, when making tradoffs, always strive to be RESTful (to a sensible extent)

Példák →

REST – to what extent is REST? (important)

- Example: use operation instead of/besides resource names:
- Alternative 1
  - If we don't have any parameter, we can handle the operation as a resource (remember some examples!):
    - Twitter example for moving an item

      `POST collections/entries/move`
    - Creating multiple users

      `POST user/createMultiple`

    We can handle a few simple parameters in query strings.
- Alternative 2
  - If the query is complex and has structured parameters (URL length can be limiting): instead of GET use POST, send input parameters as JSON

REST – to what extent is REST?

- Let’s not use the expression "REST" for our API if our approach greatly deviates from classic REST:
  - Instead use: „Web API” or „Http API”
- Microsoft also calls its related .NET platform services „Web Api” (and not REST API)

.NET WebAPI

.NET Web API

- Definition
  - Framework for building .NET based REST serviced
  - Not only for strict REST (that’s why it’s called „Web API”)
- .NET
  - .NET Core, .NET 5 and up
  - Full .NET Framework (Web API 2.0)
- ASP.NET MVC model (without the „V” – View)
- Modular, extendable, pipeline support

.NET Web API

- Principles
  - A **Controller** class for every resource type (Product, Order, etc.)
    - E.g. **ProductsController**, **OrdersController**
    - Receives the request and produces the response
  - Note: Starting from .NET 6 a new model called „minimal web API" is also supported, where no Controller classes are involved (we will not look into this alternative model)

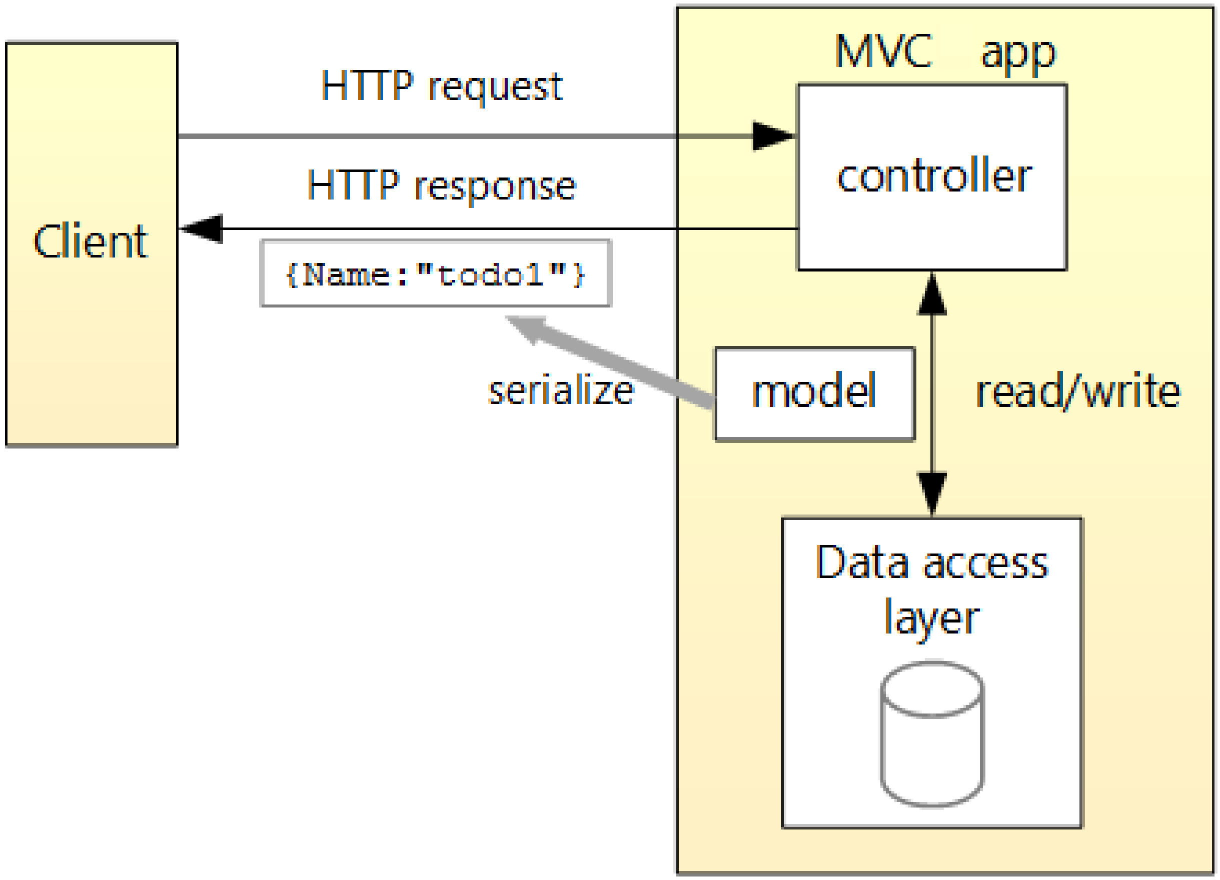
.NET Web API

- Principles
  - The Web API uses .NET routing engine to map a URL to an action/method of a controller. E.g.:

    api/order/12 → specific method of OrdersController

    api/product/24 → specific method of ProductsController
  - Response: return the resource entity from the action/method of the controller
    - JSON (or XML) serialization is automatic, framework solves it
    - These are called model entities

Architecture



.NET Core example- Demo

https://docs.asp.net/en/latest/tutorials/first-web-api.html

The API

API	Description	Request body	Response body
GET /api/todos	Every to-do item	-	To-do item collection
GET /api/todos/{id}	A single to-do item (by id)	-	To-do item
POST /api/todos	New item	To-do item	To-do item
PUT /api/todos/{id}	Update an item	To-do item	-
PATCH /api/todos/{id}	Partially update an item	To-do item	-
DELETE /api/todos/{id}	Delete an item	-	-

Todo or todos

- In most of our examples so far we used plural resource names
- Pl. todos, tasks, orders, stb.
- This is a recommended best practice

.NET Core/.NET 5+ example

- Controller base class
- Get all the items:

`GET API /api/todos`

```
[Route("api/[controller]")]
[ApiController]
public class TodosController : ControllerBase
{
    [HttpGet]
    public ActionResult<IEnumerable<TodoItem>> GetAll()
    {
        return _context.TodoItems.ToList();
    }
}
```

.NET Core /.NET 5+ example

- Derive from the ControllerBase base class
- Routing
  - It has to map each request to an action of a contoller
  - The mapping can be defined by attributes
    - Route attribute
      - In our example te TodosController class receives requests for API/todos url as the name of the class isTodosController
    - HttpGet, HttpPost, HttpPut, HttpDelete: the HTTP verb determine the action/method in the controller

.NET Core /.NET 5+ example

- Get the details of an item: **`GET API /api/todos/1`**

```
[Route("api/[controller]")]
[ApiController]
public class TodosController : ControllerBase
{
     [HttpGet("{id}")]
     public ActionResult<TodoItem> GetById(string id)
     {
         var item = _context.TodoItems.Find(id);

         if (item == null)
              return NotFound(); // 404 status code

        // The response gets JSON serialized
         return item;
     }
}
```

- HttpGet: {id} placehorder, the value in the url is passed as parameter to the method. An equivalent option the application of both of [HttpGet] and [Route("{id}")] attributes on the function.

.NET Core /.NET 5+ example

- Adding a new item:

  `POST /api/todos + JSON object in the body`

```
[Route("api/[controller]")]
[ApiController]
public class TodosController : ControllerBase
{
    [HttpPost]
    public IActionResult Create([FromBody] TodoItem item)
    {
        _context.TodoItems.Add(item);
        _context.SaveChanges();

        // Put the url of the new object into the location
        // header of the response
        return CreatedAtAction(nameof(GetById), new { id = item.Id },
                item);
    }
}
```

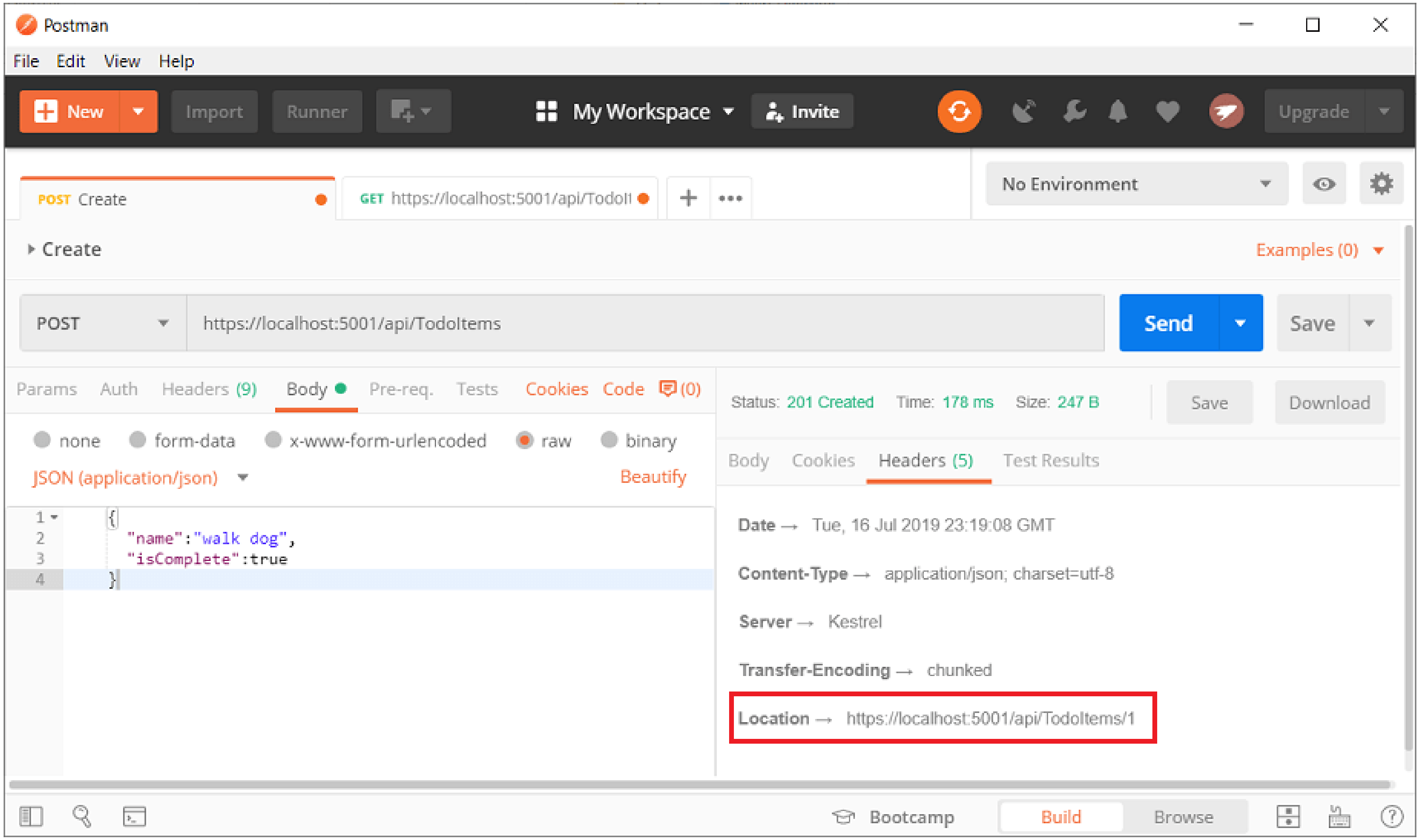
Steps

- Model classes
  - Entities of the request
- Data access
  - E.g. Repository or DbContext
- Controller
  - Receives the requests, creates the response
- Testing:
  - E.g. Postman: Chrome addin for sending HTTP requests

REST client

- How do we call a REST based client?
  - (So far we looked only at how to create REST services using ASP.NET Core)
  - It's supported by most used platforms, let it be web/desktop/mobil/backend (REST is based on simple HTTP)
  - Postman – GUI tool for REST API calls/testing
  - .NET Web API/REST client: HttpClient class

Postman – create (POST) example


Postman
File Edit View Help
New
Import
Runner
My Workspace
Invite
Upgrade
POST Create
GET https://localhost:5001/api/TodoI
No Environment
Create
Examples (0)
Send
Save
Params Auth Headers (9) Body Pre-req. Tests Cookies Code (0)
none form-data x-www-form-urlencoded raw binary
JSON (application/json)
Beautify
Save
Download
Body Cookies Headers (5) Test Results
Bootcamp
Build
Browse

POST https://localhost:5001/api/TodoItems

```
{
    "name":"walk dog",
    "isComplete":true
}
```

Status: 201 Created Time: 178 ms Size: 247 B

Date → Tue, 16 Jul 2019 23:19:08 GMT

Content-Type → application/json; charset=utf-8

Server → Kestrel

Transfer-Encoding → chunked

Location → https://localhost:5001/api/TodoItems/1

.NET client

- **HttpClient** class is recommended
  - .NET Core, .NET 5+ and .NET Standard
  - .NET Framework: starting from .NET 4.5
  - Async/await support
- JSON serialization:
  - Prior to .NET 3.0: Newtonsoft, Json.NET (www.newtonsoft.com), NuGet
  - Starting from .NET 3.0 built into .NET Core/.NET: System.Text.Json namespace (JsonSerializer, …)
- How to start
  - https://dotnet.microsoft.com/apps/aspnet/apis

REST API documentation

- How to document REST APIs?
- A) Ad-hoc soulutions
  - For each resource:
    - Table based description of: URL-s + verbs + supported parameters
    - JSON request and response examples
  - One example: https://dev.twitter.com/rest/reference/get/followers/list
- Open API (old name is Swagger)
  - Similar to SOAP WSDL, but for REST

→

OpenAPI (Swagger) specification

- Started as a „Community” initiative, today it’s considered to be a standard maintained by a consortium
- It’s an API description format for REST APIs, including:
  - Available endpoints (/users) and operations on each endpoint (GET /users, POST /users)
  - Operation parameters Input and output for each operation
  - Authentication methods
  - Contact information, license, terms of use and other information.
- YAML and JSON formats are supported
- https://swagger.io/docs/specification/about/

REST API documentation

- Swagger, as an open-source „tool-set" for
  - Editing/creating the OpenAPI description of a REST API
  - Generating interactive HTML based description from the OpenAPI description (Swagger UI)
  - Discover the REST API from the OpenAPI description
  - Generate client classes (Java, .C#, TypeScript, etc.) from the OpenAPI description
- Demo: http://petstore.swagger.io

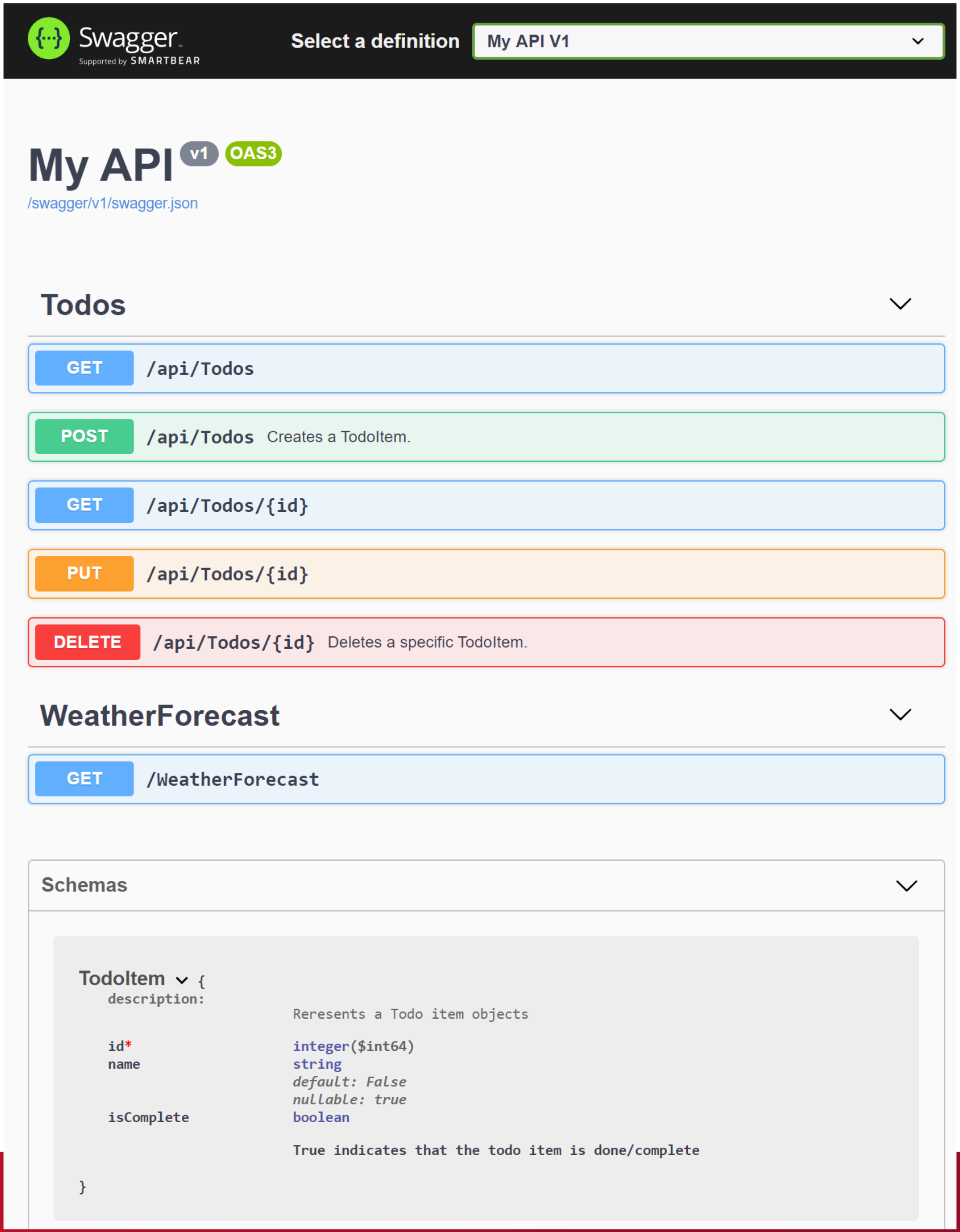
OpenAPI description example (JSON), simplified

```
{
   "swagger": "2.0",
   "info": {
       "version": "v1",
       "title": "API V1"
   },
   "basePath": "/",
   "paths": {
       "/api/Todos": {
           "get": {
               "tags": [
                   "Todo"
               ],
               "operationId": "ApiTodoGet",
               "consumes": [],
               "produces": [
                   "text/plain",
                   "application/json",
                   "text/json"
               ],
               "responses": {
                   "200": {
                       "description": "Success",
                       "schema": {
                           "type": "array",
                           "items": {
                               "$ref":
                        "#/definitions/TodoItem"
                           }
                       }
                   }
                }
           },
           "post": {
```

```
               ...
           }
       },
       "/api/Todos/{id}": {
           "get": {
               ...
           },
           "put": {
               ...
           },
           "delete": {
               ...
   },
   "definitions": {
       "TodoItem": {
           "type": "object",
            "properties": {
                "id": {
                    "format": "int64",
                    "type": "integer"
                },
                "name": {
                    "type": "string"
                },
                "isComplete": {
                    "default": false,
                    "type": "boolean"
                }
            }
       }
   },
   "securityDefinitions": {}
}}
```

Swagger UI

Interactive HTML „help" pages for discovering and calling endpoints based on the OpenAPI description of a REST API



My API v1 OAS3
/swagger/v1/swagger.json

Todos

GET /api/Todos

POST /api/Todos Creates a TodoItem.

GET /api/Todos/{id}

PUT /api/Todos/{id}

DELETE /api/Todos/{id} Deletes a specific TodoItem.

WeatherForecast

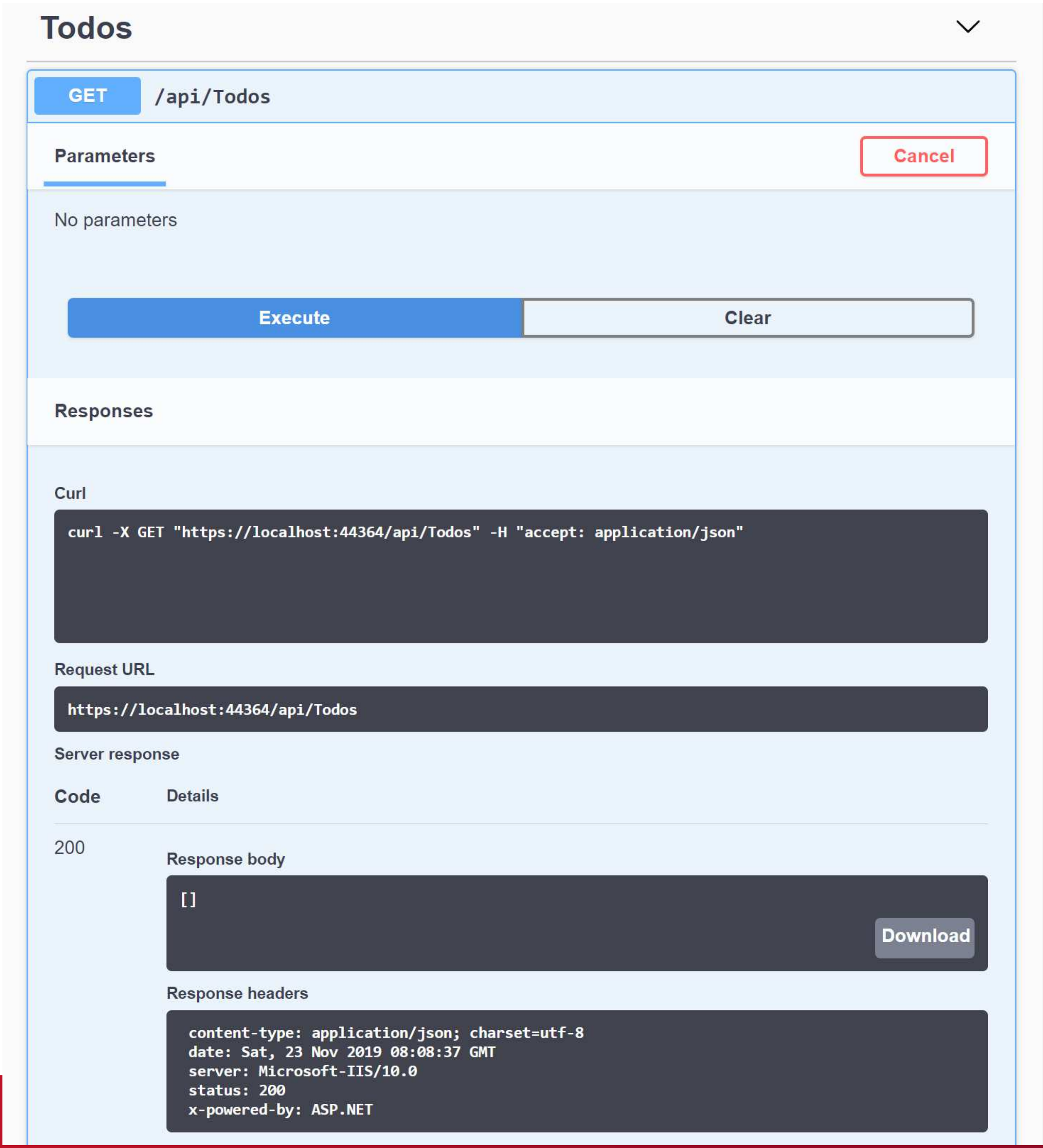
GET /WeatherForecast

Schemas

```
TodoItem {
    description:
                          Reresents a Todo item objects

    id*                   integer($int64)
    name                  string
                          default: False
                          nullable: true
    isComplete            boolean

                          True indicates that the todo item is done/complete

}
```

Swagger UI

Example, detailed view selecting the GET operation



No parameters

Curl

```
curl -X GET "https://localhost:44364/api/Todos" -H "accept: application/json"
```

Request URL

```
https://localhost:44364/api/Todos
```

Server response

Code	Details
200	Response body

```
[]
```

Response headers

```
content-type: application/json; charset=utf-8
date: Sat, 23 Nov 2019 08:08:37 GMT
server: Microsoft-IIS/10.0
status: 200
x-powered-by: ASP.NET
```

OpenAPI/Swagger in .NET env.

- Swashbuckle.AspNetCore (NuGet) is the practical choice (NSwag.AspNetCore is an alternative), contains these packages as alternatives:
  - Swashbuckle.AspNetCore.Swagger: middleware for publishing OpenAPI/Swagger Json endpoints
  - Swashbuckle.AspNetCore.SwaggerGen: OpenAPI/swagger generator
  - Swashbuckle.AspNetCore.SwaggerUI: embedded Swagger UI tool.

- A fentiekről leírás itt:
  - https://docs.microsoft.com/en-us/aspnet/core/tutorials/web-api-help-pages-using-swagger
  - https://docs.microsoft.com/en-us/aspnet/core/tutorials/getting-started-with-swashbuckle

GraphQL

Source: https://www.howtographql.com/

What is GraphQL?

- New API standard, SOAP/REST alternative (first released in 2015)
- More **efficient**, **powerful** and **flexible** alternative to REST
  - At least under certain conditions
- Originally developed and open-sourced by Facebook and is now maintained by a large community of companies

What is GraphQL?

- GraphQL enables declarative *data fetching*
- Client can specify exactly what data it needs from an API
- Instead of multiple endpoints that return fixed data structures, a GraphQL server only exposes a single endpoint and responds with precisely the data a client asked for.
- Not a Database Technology. GraphQL is a query language for APIs - not databases.

Motivation

- **1. Increased mobile usage creates need for efficient data loading**
  Increased mobile usage, low-powered devices and sloppy networks were the initial reasons why Facebook developed GraphQL. GraphQL minimizes the amount of data that needs to be transferred over the network and thus majorly improves applications operating under these conditions.
- **2. Variety of different frontend frameworks and platforms**
  The heterogeneous landscape of frontend frameworks and platforms that run client applications makes it difficult to build and maintain one API that would fit the requirements of all. With GraphQL, each client can access precisely the data it needs.
- **3. Fast development & expectation for rapid feature development**
  Continuous deployment has become a standard for many companies, rapid iterations and frequent product updates are indispensable. With REST APIs, the way data is exposed by the server often needs to be modified to account for specific requirements and design changes on the client-side. This hinders fast development practices and product iterations.

GraphQL features

- Efficient API → The client can specify exactly what data it needs from the server
  - No overfetching (sending more data than needed)
    - See following slide, GitHub example
  - No underfecthing,
    - Underfecthing: a specific endpoint doesn't provide enough of the required information. The client will have to make additional requests to fetch everything it needs. See upcoming slides for an example.
- Flexible API → Rapid Product Iterations on the Frontend
  - Thanks to the flexible nature of GraphQL, changes on the client-side can be made without any extra work on the server.

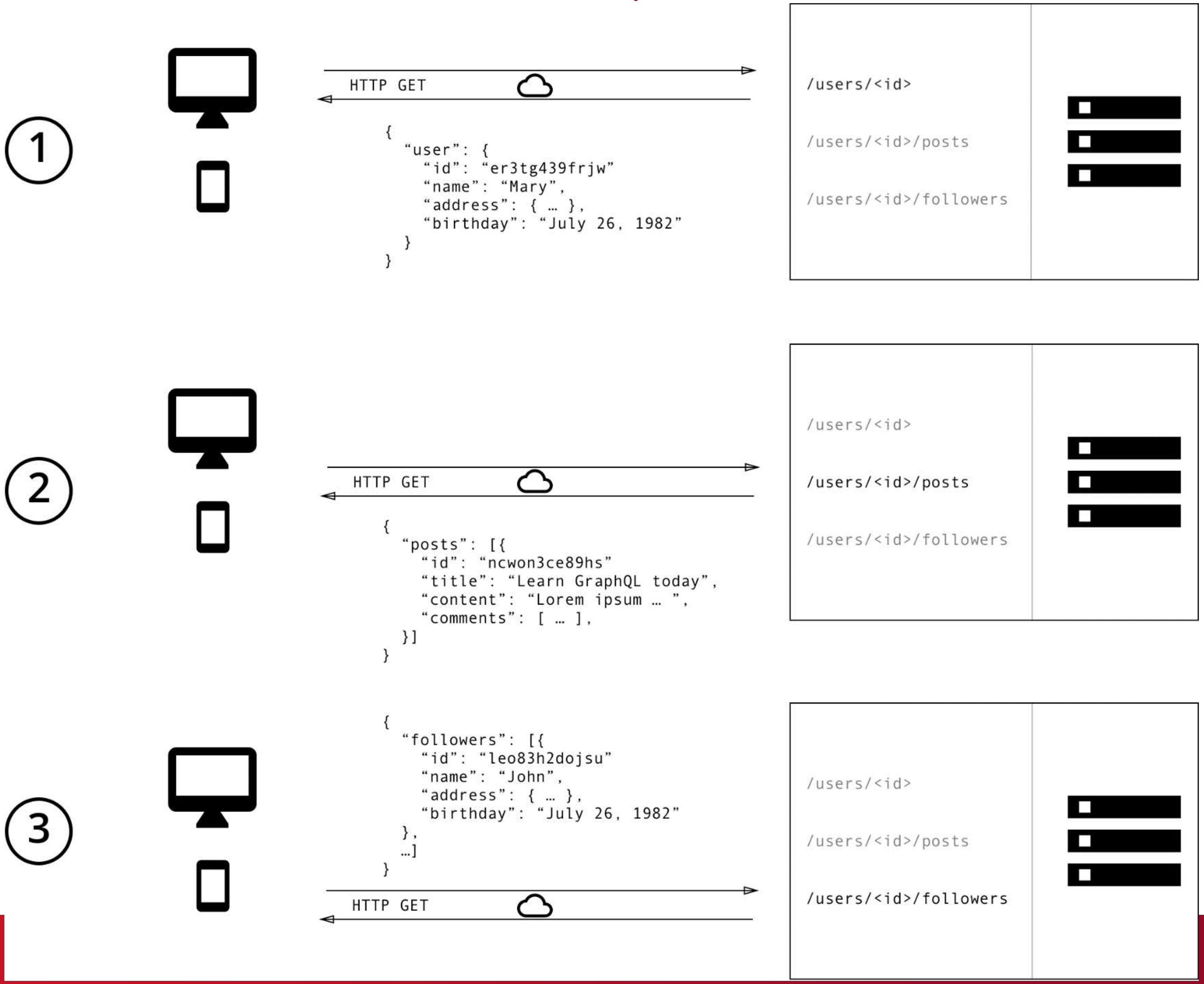
REST API overfetching example

- GitHub REST API **overfetching** example
  - https://developer.github.com/v3/repos/commits/, for each commit GitHub REST API returns a lot of information
  - What if we need only e.g. the „commit message” or „commit hash”? Still, in this case, all superfluous information is sent over the network to the client.

```
[
 {
  "url": "https://api.github.com/repos/octocat/Hello-World/commits/6dcb09b5b57875f334f61aebed695e2e4193db5e",
  "sha": "6dcb09b5b57875f334f61aebed695e2e4193db5e",
  "node_id": "MDY6Q29tbWl0NmRjYjA5YjViNTc4NzVmMzM0ZjYxYWViZWQ2OTVlMmU0MTkzZGI1ZQ==",
  "html_url": "https://github.com/octocat/Hello-World/commit/6dcb09b5b57875f334f61aebed695e2e4193db5e",
  "comments_url": "https://api.github.com/repos/octocat/Hello-
World/commits/6dcb09b5b57875f334f61aebed695e2e4193db5e/comments",
  "commit": {
   "url": "https://api.github.com/repos/octocat/Hello-World/git/commits/6dcb09b5b57875f334f61aebed695e2e4193db5e",
   "author": {
    "name": "Monalisa Octocat",
    "email": "support@github.com",
    "date": "2011-04-14T16:00:49Z"
   },
   "committer": {
    "name": "Monalisa Octocat",
    "email": "support@github.com",
    "date": "2011-04-14T16:00:49Z"
   },
   "message": "Fix all the bugs",
   "tree": {
    "url": "https://api.github.com/repos/octocat/Hello-World/tree/6dcb09b5b57875f334f61aebed695e2e4193db5e",
    "sha": "6dcb09b5b57875f334f61aebed695e2e4193db5e"
   },
   "comment_count": 0,
   "verification": {
    "verified": false,
    "reason": "unsigned",
    "signature": null,
    "payload": null
   }
  },
  "author": {
   "login": "octocat",
   "id": 1,
   "node_id": "MDQ6VXNlcjE=",
   "avatar_url": "https://github.com/images/error/octocat_happy.gif",
   "gravatar_id": "",
   "url": "https://api.github.com/users/octocat",
   "html_url": "https://github.com/octocat",
   "followers_url": "https://api.github.com/users/octocat/followers",
   "following_url": "https://api.github.com/users/octocat/following{/other_user}",
   "gists_url": "https://api.github.com/users/octocat/gists{/gist_id}",
   "starred_url": "https://api.github.com/users/octocat/starred{/owner}{/repo}",
   "subscriptions_url": "https://api.github.com/users/octocat/subscriptions",
   "organizations_url": "https://api.github.com/users/octocat/orgs",
…
```

Data fetch REST example

The client need to turn to the server multiple times

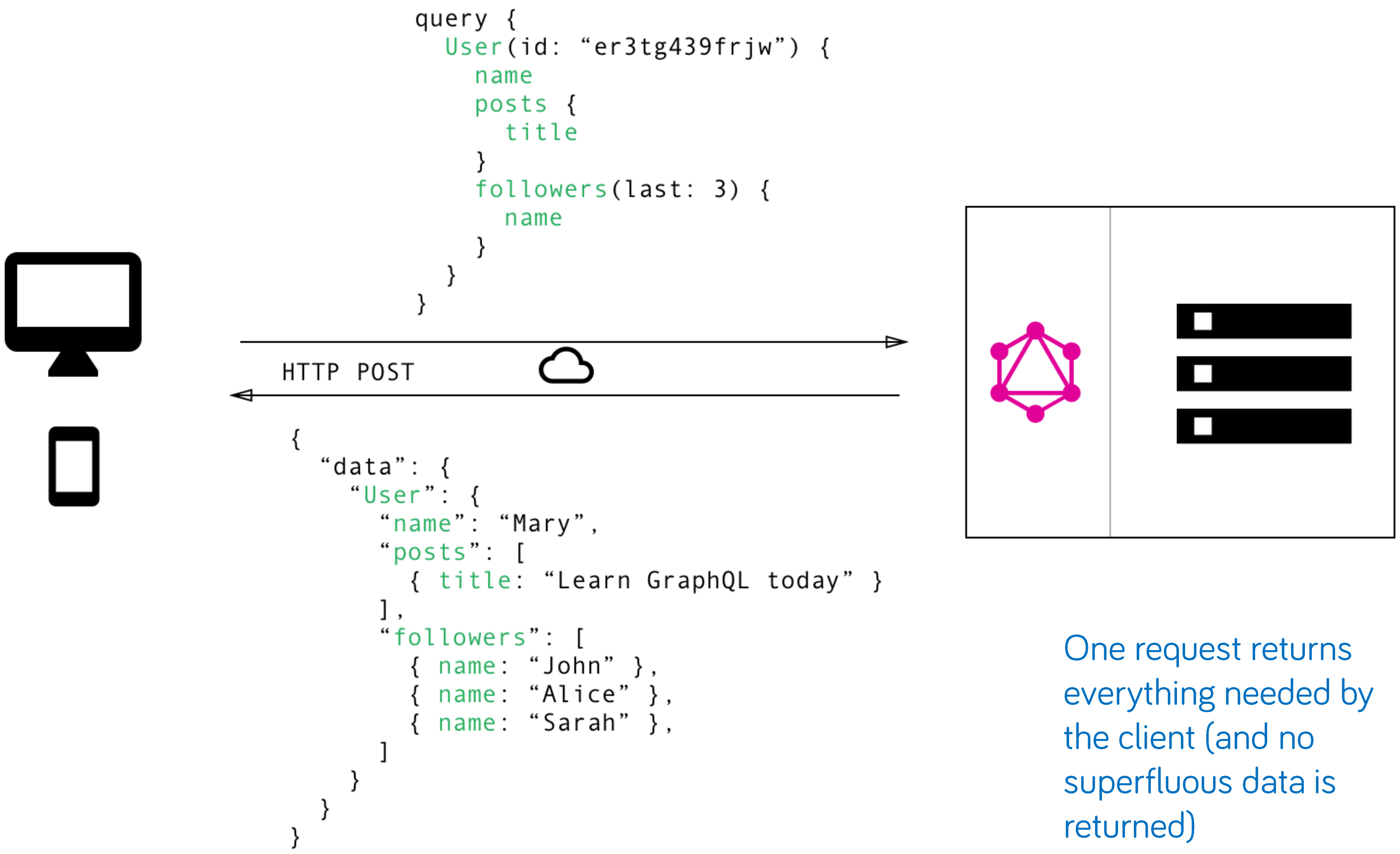


```
{
  “user”: {
    “id”: “er3tg439frjw”
    “name”: “Mary”,
    “address”: { … },
    “birthday”: “July 26, 1982”
  }
}
```

```
{
  “posts”: [{
    “id”: “ncwon3ce89hs”
    “title”: “Learn GraphQL today”,
    “content”: “Lorem ipsum … ”,
    “comments”: [ … ],
  }]
}
```

```
{
  “followers”: [{
    “id”: “leo83h2dojsu”
    “name”: “John”,
    “address”: { … },
    “birthday”: “July 26, 1982”
  },
  …]
}
```

Data fetch GraphQL example



```
query {
  User(id: “er3tg439frjw”) {
    name
    posts {
      title
    }
    followers(last: 3) {
      name
    }
  }
}
```

```
{
  “data”: {
    “User”: {
      “name”: “Mary”,
      “posts”: [
        { title: “Learn GraphQL today” }
      ],
      “followers”: [
        { name: “John” },
        { name: “Alice” },
        { name: “Sarah” },
      ]
    }
  }
}
```

One request returns everything needed by the client (and no superfluous data is returned)

GraphQL – type system + schema

- GraphQL uses a strong type system to define the capabilities of an API. All the types that are exposed in an API are written down in a *schema* using the GraphQL Schema Definition Language (SDL). This schema serves as the contract between the client and the server to define how a client can access the data.
- Once the schema is defined, the teams working on frontend and backends can do their work without further communication since they both are aware of the definite structure of the data that's sent over the network.

Simple type definition examples

```
type Person {
  id: ID!
  name: String!
  age: Int!
}

type Post {
  title: String!
  author: Person!
}

type Person {
  name: String!
  age: Int!
  posts: [Post!]!
}

type Query {
  allPersons(last: Int): [Person!]!
}
```

- „!”: required field (it’s a non-null type, meaning the GraphQL service promises to give you a value whenever the client queries this field)
- []: array, multiple items
- The example actually defines a one-to-many relationship
- The Query keyword defines and API entry point for clients
- „allPersons” is a root field, our query has only one root field

Simple queries and responses

Given that we have the type definitions of the previous slide

```
{
  allPersons {
    name
  }
}
```

→

```
{
  "data": {
        "allPersons": [
          { "name": "Johnny" },
          { "name": "Sarah" },
          { "name": "Alice" }
        ]
  }
}
```

```
{
  allPersons {
    name
    age
  }
}
```

→

```
{
  "data": {
        "allPersons": [
            {
              "name": "Johnny",
              "age": 23
            },
            {
              "name": "Sarah",
              "age": 20
            },
            {
              "name": "Alice",
              "age": 20
            }
          ]
  }
}
```

Queries reflect the structure of the expected response.

The response is typically JSON!
(the platform may support other formats as well)

Query and response – embedded data

```
{
  allPersons {
    name
    age
    posts {
      title
    }
  }
}
```

```
{
   "data": {
      "allPersons": [
         {
           "name": "Johnny",
           "age": 23,
           "posts": [
             { "title": "GraphQL is awesome" },
             { "title": "Relay is a powerful GraphQL Client" }
           ]
         },
         {
           "name": "Sarah",
           "age": 20,
           "posts": [
             { "title": "How to get started with GraphQL" }
           ]
         },
         {
           "name": "Alice",
           "age": 20,
           "posts": []
         }
      ]
   }
}
```

One of the major advantages of GraphQL is that it allows for naturally querying nested information. For example, if you want to load all the posts that a Person has written, you could simply follow the structure of your types to request this information.

Query parameters

- Each field (even embedded ones) can have parameters
- These have to be defined with the types in the schema
- In our previous example we introduced one parameter:

```
type Query {
  allPersons(last: Int): [Person!]!
}
```

- Request and response example

```
{
  allPersons(last: 2) {
    name
  }
}
```

```
{
  "data": {
    "allPersons": [
      {
        "name": "Sarah"
      },
      {
        "name": "Alice"
      }
    ]
  }
}
```

GraphQL operation types

- GraphQL three operation types
  - Query
    - This is what we used so far in our examples. The „query” keyword is to be used when forming queries, but it can be omitted (as we did in our examples)
  - Mutation
    - For manipulating data, will cover them later
  - Subscription
    - When a client *subscribes* to an event, it will initiate and hold a steady connection to the server. Whenever that particular event then actually happens, the server pushes the corresponding data to the client.

Changing data

- Called mutation in the context of GraphQL
- Has three categories
  - creating new data
  - updating existing data
  - deleting existing data
- Mutation types have to be defined with the Mutation keyword in the schema

```
type Mutation {
  createPerson(name: String!, age: Int!): Person!
}
```

- Requests have to be started with the mutation keyword

→

Changing data

- Request and response example for mutation

```
mutation {
  createPerson(name: "Alice", age: 36) {
    id
  }
}
```

⇩

```
{
  "data": {
    "createPerson": {
      "id": "ck3cn8xe70vq3016806asxw6g"
    }
  }
}
```

When forming the requet besides providing the parameter values we also describe the format of the expected resonse (in exactly the same form as for queries). In our example we ask only for the „id" of the newly created Person object.

GraphQL evaluation - pros

- Advantages mostly have been covered, a few other ones:
  - The complete schema is accessible/downloadable from the server for clients. This is called introspection.
  - Frontend development – assuming having good library support – is greatly simplified.

GraphQL evaluation - cons

- Drawbacks
  - Although it greatly simplifies frontend development, the tradeoff is more complex logic is required on the backend side.
  - Since clients have the possibility to craft very complex queries, our servers must be ready to handle them properly. These queries may be abusive queries from evil clients, or may simply be very large queries used by legitimate clients. In both of these cases, the client can potentially take your GraphQL server down. We have to introduce some strategies to mitigate such risks.
  - Need good library suppor for both client and server (Apollo, HotChocolate etc)

GraphQL evaluation

- GraphQL is transport layer agnostic
  - Typically above HTTP (JSON)
  - Completely hidden by libraries
- First made big impact in case of mobile and thin web client applications
- More practical than REST in a lot of scenarios, but not always, need to make a decision for each case. These days REST is still (far) more popular.
- GraphQL has great momentum (has better and better library support as time goes on).

GraphQL: further not covered topics

- Server side resolvers
- Data subscriptions
- Platform specific libraries
- Client caching, normalization, …
- Alias, fragment, variable
- Server overload protection (Timeout, Maximum Query Depth, Query Complexity, Throttling)
- Introspection