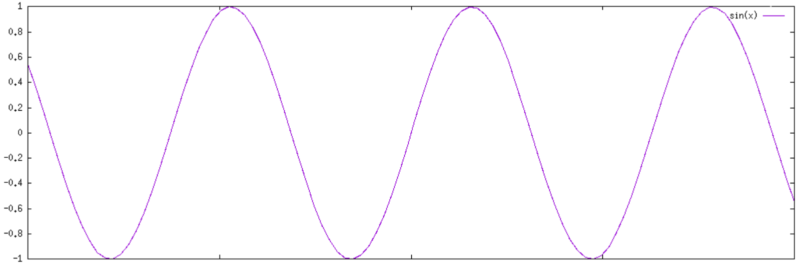
WHAT COMES AFTER REST?

Waldemar Korłub

Mobile services on the Internet
KASK ETI Gdańsk University of Technology

IT industry evolution



- Alternating periods of:
  - (r)evolution
    - Looking for new solutions
      - …to the problems we encountered with the previous standard
    - Period of creativity
    - Multiple approaches are proposed
      - Some of them may be standardized later on
      - Many will be soon forgotten
  - Standardization
    - Allows widespread adoption
    - Allows tooling support
    - Period of productivity (established solutions + tools = productivity)
    - Standards lock you into a particular solution
    - They become outdated

Web Services evolution

- CORBA (199x)
  - Standard for RPC
- XML (late 1990s, early 2000s)
  - Ubiquitous data format
  - Swamp of communication protocols
- SOAP (2002+)
  - Standard for XML-based document exchange and RPC

Web Services evolution

- JSON+HTTP (2005+)
  - Web developers have been using AJAX for quite a while
  - No ubiquitous conventions for URIs structure, operation semantics
- RESTful Web Services (2009+)
  - Not really a *proper* standard
  - …but a popular convention nonetheless
- Swagger, RAML, API Blueprints (2013+)
  - "standards" for REST API description
  - No love for WADL?
    - Submitted in 2009 to W3C, but never standardized

Web Services evolution

- GraphQL, Falcor (2015)
  - Solutions for consumer-driven contract definition
  - HTTP as a transport layer
    - Driving away from REST
- What comes next?

REST is not the end of the road!

Criticism of REST APIs

- Fetching of complicated object graphs require multiple HTTP requests
  - Responsive data capabilities are coarse-grained and often does not offer adequate flexibility
- Data contract usually driven by server-side application
  - New data fields added to reflect new functionalities of the REST API
  - Payloads grow over time
    - Even if clients do not require additional data
  - API versioning could solve *this* issue
    - …but introduces a lot of other problems at the same time

Criticism of REST APIs

- Usually weakly-typed
  - Not designed for tooling support
  - Client’s behaviours based on documentation (often outdated) instead of strongly-typed contracts reflecting current server-side endpoints
  - But what about Open API (Swagger), RAML, API Blueprints?

Consumer-driven contract

- Single version of data will not suit all clients
  - Required denormalized representations traversing different complex sub-resources
- Let clients decide what representation of data they need
- Responsive data – analogy to responsive websites
  - Different views of the same website depending on the characteristics of the client device
- Multiple clients of the API
  - Different apps for different mobile OSes
    - Separate apps for smartphones/tablets
  - Different versions of the same mobile app
  - 3rd party clients (e.g. websites)

Responsive data

- Getting a representation of data useful for the client (simple approach):
  - `http://example.com/users/1234?expand=group,private-messages,friends`
  - `http://example.com/users/1234?expand=group,messages,private-count`
- Might be *just-enough*-expressive for all clients
- Many APIs do it this way!
  - A common approach
  - …but not a *standard*
    - Think SQL – a *standard* for querying different databases

GraphQL

- API query language
- Developed by Facebook
  - Utilized in Facebook mobile apps
  - Publicly available since 2015
- Focus on types and fields not endpoints
- Allows to obtain many resources in a single request
  - Especially import for mobile clients
  - Product-centric

GraphQL

- Encourages API evolution instead of versioning
  - Facebook releases apps on a two week fixed cycle
  - Each release supported for at least 2 years
  - At least 52 versions per platform of client app needs to be supported
- Not limited to a specific storage engine
  - Uniform interface for many databases
  - Invokes arbitrary server-side code to fetch data from storage engines

GraphQL

- Application-layer protocol
  - Does not require any specific transport layer
- Strongly typed
  - Well standardized
  - Formalized client-server contract
  - Allows for tooling support

GraphQL: Data types

```
type Car {
    id: ID!
    brand: String!
    model: String!
    engineCapacity: Float
    regNumber: String!
}

enum Faculty {
    ETI
    ZIE
    FTIMS
}
```

```
type Employee {
    id: ID!
    name: String!
    principal: Employee
    employedAt: Faculty!
    cars: [Car]
    issuedEntryCards: Int
}
```

GraphQL: Entry point

```
schema {
    query: Query //entry point
}

type Query {
    employee(id: ID!): Employee
}
```

GraphQL: Queries

Query:

```
query {
       employee(id: 123)  {
         name
              employedAt
               cars {
           regNumber
               }
       }
}
```

Response:

```
{
  data: {
     employee: {
        name:  "Waldemar Korłub",
        employedAt:  "ETI",
        cars:  [
         {
             regNumber:  "ABC 1234"
         }
        ]
     }
  }
}
```

Tooling for GraphQL

- Server-side libraries for:
  - JavaScript
  - Ruby
  - Python
  - Scala
  - Java
- Client-side libraries for:
  - JavaScript
    - Including environments like React, React Native, Angular 2 and plain JavaScript
  - Swift / iOS

REST Issues: Lack of verbs

- When you only know 4 verbs it is hard to communicate
  - GET, POST, PUT, DELETE
  - Imagine talking to another person while only using 4 verbs
    - e.g. to have, to want, to eat, to sleep
- Some business domains can be mapped to HTTP verbs and resources quite easily
  - …for others this kind of mapping is counter-intuitive

Easy example: products in your fridge

- Create new product
  - POST /products
- Read information about product
  - GET /products/17
- Update information about product
  - PUT /products/17
- Delete product from the fridge
  - DELETE /products/17
- CRUD!

REST as CRUD

- It is easy to use REST when you just need a CRUD:
  - Create → POST
  - Read → GET
  - Update → PUT
  - Delete → DELETE
- Many business requirements go beyond simple CRUD capabilities
  - Otherwise we would be out of jobs for devs
    - CRUD can be easily generated by automated tools

REST Architectural Constraints

- client-server
- stateless
- cacheable
- layered
- uniform interface:
  - identification of resources
  - manipulation through representations
  - self-descriptive messages, e.g. format, cacheability
  - HATEOAS

Case study: Change the amount

- Requirement: change the amount of a product though operations like:
  - Increase by x
  - Decrease by x
- Can we PUT request?
  - PUT /products/17/amount
    - delta: -3
  - We are not providing a representation of the amount resource
  - Against the PUT semantics – PUT should be idempotent

Case study: Change the amount

- How about PATCH (WebDAV)?
  - PATCH /products/17/amount
    - delta: -3
  - We are trying to fix REST by introducing additional verbs
    - …which only reaffirms that 4 verbs if not enough
- How about POST?
  - POST /products/17/amount/deltas
    - delta: +6
  - Seems RESTful
  - But we are introducing a new sub-resource to make up for the lack of verbs

Case study: RESTful authentication

- Requirement: design a RESTful endpoint for client authentication
- Assume access control scheme involving Access Tokens
  - User credentials (login, password) are exchanged for an Access Token
- Endpoint should follow verbs semantics defined in the HTTP protocol specification

Case study: RESTful authentication

- POST /login
  {"login": "stawrul", "password": "p@ssw0rd"}
  - Are we creating a new resource here?
  - /login is not even a real resource, it is an operation
  - In a RESTfull service a URI should identify resources (nouns) and not operations (verbs)

Case study: RESTful authentication

- If we want to obtain an access token than how about:
  GET /users/stawrul/token
  {"password": "p@ssw0rd"}
  - GET operation should be *safe* and *idempotent*
  - Server needs to crate an AT, so it is not *safe*
  - There might be multiple tokens for a single user
    - Single URI should represent a single resource

Case study: RESTful authentication

- If we want an AT to be create than why don’t we use POST?

```
POST /users/stawrul/tokens
 {"password": "p@ssw0rd"}
```

  - The Body of the request obviously does not represent a token
    - Against *manipulation by representations* principal
  - It is not the client who creates the token – the server creates the token on client’s request

Case study: RESTful authentication

- So lets create a request for an AT:

  ```
  POST /token_requests
  { "login": "stawrul",
    "password": "p@ssw0rd",
    "token_type": "access_token" }
  ```

- The response could look like this:

  ```
  201 Created
  {"access_token": "br4k2ew43reobx723"}
  ```

- Is this RESTful?

Case study: RESTful authentication

- POST /token_requests
- Is this RESTfull?
  - Is /token_requests an actual resource in our application?
    - It seems like a superfluous entity created only to conform to RESTful conventions
    - We create additional resources to make up for the lack of verbs

RESTful vs RPC

- Those use cases can be easily expressed using RPC model:

```
AccessToken t = authService.login(login, pass);
```

- Is there anything wrong with RPC model?
  - CORBA was based on RPC model
  - Remote EJBs are based on RPC
  - .NET Remoting is based on RPC
  - SOAP has both RPC and document exchange models
  - These are all outdated…
  - …but there is nothing wrong with RPC model itself

RESTful vs RPC

- When you write code you think about method invocations
  - Complex business domains can be express through objects and their methods
    - …and interactions between them
  - If we use RPC we don't have to translate our internal business logic based on method invocations to the resources model of RESTful web services
    - …and than back again to the method invocation model on the other side of the service
- We just need a modern ubiquitous standard for RPC

RESTful vs RPC

- So why do we use REST?
  - Lack of modern ubiquitous tools for RPC
  - It might not be *the best* standard but it is still *a* standard
    - Easier consumption by client applications
    - Easier interoperability
  - Little to no requirements regarding tooling support
    - HTTP client is just enough
  - …so we constrain ourselves to the resources model of REST

REST Issues: Performance with REST

- JSON is better than XML in terms of:
  - Processing power required to parse data
  - Time required to parse data
  - Data size *on-the-wire*
- …but it is still a text-based format
  - Nowhere near the performance and conciseness of binary data formats

Interoperability (is not an issue in REST)

- Text-based formats are good for interoperability
  - Data decoupled from its binary representation in any particular programming language/platform/OS
- Text-based formats are good for humans
  - Human-readable
    - Easy debugging and inspection of data
- HTTP is good for interoperability
  - Text-based, ubiquitous
- REST is good for interoperability
  - Easy consumption of services
    - …even without dedicated tooling
    - Possible in every popular programming language

Interoperability

- Do we always need that level of interoperability?
- Yes, we do for:
  - Publicly available APIs
  - APIs meant for consumption by 3rd party developers
- But if we have control over the server and the client we might not need that level of interoperability
  - So we can optimize our services!
    - Make them more efficient
    - …at the cost of being harder to consume for outsiders

The need for performance

- Why would we need better performance?
  - Isn’t REST just *fast enough* for most use cases?
  - It is fine from the point of view of a single mobile device
- Servers handling millions of mobile clients
  - Reduction of cost and time of computations

The need for performance

- Microservices architecture
  - Microservices can allow for better scaling of applications
  - But the gain from scaling can be lost on communication overhead
    - Data serialization/deserialization between microservices
  - An advent of binary protocols
- There is a lot of hype around microservices
  - Solutions proposed for microservices can quickly became industry standards
    - Leading to fast adoption also in other areas

Questions?