Microservices & REST

CCS3341 Cloud Computing

Dr S. Veloudis

SOA vs Monolithic

- We saw the benefits brought about when replacing a monolithic application tier with a SOA

But are SOAs really delivering the benefits they promise?

- It turns out that SOA services are exhibiting after all a **high degree of coupling…**

And how is this?

- SOA services are architected around the tenet that they should **share resources**

Remember: The ESB is part of SOA. The ESB is required for simplifying service interaction. Without the ESB, a service would have to implement a multitude of bindings, one for each service that it interacts with; moreover, it would have to update these bindings each time a new service is added, or an existing one is changed…

Main shared resource is the ESB!

SOA vs Monolithic

- Nevertheless, resource sharing leads to **tight coupling**
  - With the ESB

Dependence on the ESB harms **portability** and thus **reusability**

In other words, SOA services are developed with the principle in mind that they will ultimately communicate with the ESB…

- Moreover, the **large size** of services impedes reusability

Hard to find use cases in which coarse-grained SOA services that offer a multitude of functionalities can be reused

It also impedes **scaling**

- Even without the ESB, for any two services to communicate with each other they need to exactly specify their APIs

E.g., when two services interact in the context of the **SOA triangle**

- This leads too to **tight coupling** harming **portability** and **reusability**

Recall the detailed interface descriptions provided through WSDL!

But isn't it reasonable to expect that, for a service A to interact with a service B, A is fully aware of B's exact communication requirements?

Microservices

- Microservices are an evolution of SOAs
- They comprise more granular services, each **focusing on a single purpose**
- Microservices are truly decoupled from each other and can function independently
- Their limited functionality typically gives rise to:

Simpler communication requirements

Allows communication through lightweight protocols (**REST**)

Fine-grained scalability

Note: A microservice typically offers a single functionality (recall: separation of concerns…)

Note: Microservices are **truly autonomous** in the sense that they are developed having the principle in mind that they should be able to offer their service independently of any other services (e.g., independently of the existence of an ESB…)

Microservices are architected around the tenet that they should share as little as possible!

Microservices

- This decoupling also brings about:

Reusability, portability, maintenance	**Agility**
• Services readily reusable in any context (with or without an ESB!) • Changes in one service do not affect other services • Best of breed!	• Microservices-based applications lend themselves to CI/CD for they inherently facilitate the gradual update and deployment of services through rolling updates and without downtime • Their true modularity and small size facilitate the kind of updates promoted through CI/CD

REST

- Stands for **Re**presentational **S**tate **T**ransfer
- An architectural style based on the HTTP protocol
  - First described by Roy Fielding in 2000
- In a REST-based architecture:
  - **Everything is a resource!**
  - There is typically a **REST controller** providing access to (back-end) resources
  - There is typically a **REST client** which requests access to the (back-end) resources

REST is not a protocol, it is an **architectural style**, or **paradigm**, that draws upon another protocol, namely HTTP. REST developed from the question: "how do we take the insights that made HTTP so successful for the web and use them for distributed computing"? REST imposes a style that simplifies distributed interfaces so that distributed resources can refer to each other just like distributed HTML pages refer to each other (via HTTP links).

A REST resource is:
- Identified by a global id – **a URI**
- Accessed via the interface that it exposes

Based on standard HTTP methods!

HTTP

- HTTP is an application-layer protocol for transmitting hypermedia documents
- HTTP does not include provisions for associating a request to a response

Associations can only be achieved through the order in which responses arrive over a (TCP) connection

The rationale behind this is that it reduces the amount of data transferred

HTTP is stateless!

- Neither the server, nor the client, keeps any information that could associate two or more requests as being part of a session
- In other words, each request must be able to be served without any knowledge of prior requests and responses

Such information would include, for example, a session id as well as the ids of the two communicating parties

- HTTP offers **request methods**

Frequently used methods in REST:
GET
POST
PUT
DELETE

HTTP methods reflect the semantics of a request i.e., what a request sets out to achieve – e.g., ***retrieve*** *an existing web resource*, ***create*** *a new one*, ***update/delete*** *an existing one*, etc.

HTTP Requests and Responses

Request

HTTP method followed by destination URI

Request line

```
Host: www.example.com
User-Agent: MyCustomUserAgent
Accept-Encoding: gzip, deflate
Accept-Language: en-US, en;q=0.5
Connection: keep-alive
Content-Type: application/json
Origin: http://www.example-origin.com
```

Empty line

Request body

⋮

E.g.

Mozilla / 5.0 (iPhone; CPU iPhone OS 8_0 like Mac OS X) AppleWebKit / 600.1.3 (KHTML, like Gecko) Version / 8.0 Mobile / 12A4345d Safari / 600.1.4

Response

1XX INFORMATIONAL
2XX SUCCESS
3XX REDIRECTION
4XX CLIENT ERROR
5XX SERVER ERROR

Status line

```
HTTP/1.1 200 OK
Server: ExampleServer/1.0
Content-Length: 1234
Content-Type: text/html; charset=UTF-8
Set-Cookie: sessionId=ABC123; Path=/; HttpOnly; Secure
Access-Control-Allow-Origin: *
```

Empty line

Request body

⋮

GET Method

- Reads or retrieves the representation of the resource identified by the URI specified in the request line

If the resource is a data-producing process, it is the produced data that is returned as response, not the source text of the process!

- A safe and idempotent operation for it never alters a resource

Note: An operation is **idempotent** iff multiple applications of the operation yield the same result as its initial application. All safe operations are by definition idempotent

Note: The body of the request is **not** significant

Request data (parameters) are piggybacked on the URI: `<uri>?key-value&key-value…`

```
GET http://127.0.0.1:5500/styles/navigation.css HTTP/1.1
```

```
HOST: 127.0.0.1:5500
Accept: text/css,*/*;q=0.1
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate, br
User-Agent: Mozilla/5.0 (Windows NT 10.0;
Win64; x64; rv:102.0) Gecko/20100101 Firefox/102.0
Connection: keep-alive
<CRLF>
```

POST Method

- Requests that the entity enclosed in the body of a request message, is accepted as input by the resource identified by the URI specified in the request line

The URI in the request line identifies the (server-side) resource that will handle according to its own business logic and semantics the POSTed entity (i.e. the entity enclosed in the body of the request). That resource is a data-accepting process (an application!) that handles the POSTed entity according to its business logic.

This handling is in fact 'responsible' for the non-idempotency of POST requests

Note: The POSTed entity might be:

- A message for a bulletin board, newsgroup, mailing list, comment thread
- A block of data that is the result of submitting a web form to a data-handling process
- An item to add to a database
- …

```
POST / HTTP/1.1
Host: localhost:8000
User-Agent: Mozilla/5.0 (Macintosh;… )… Firefox/51.0
Accept: text/html,application/xhtml+xml,…,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Content-Type: multipart/form-data; boundary=-12656974
Content-Length: 345

-12656974
(more data)
```

- Neither a safe nor an idempotent operation

PUT Method

- Requests that the entity in the body of the request message be stored under the URI supplied in the header of the request message

Note: If the URI points to an existing resource, the content of the resource is replaced by the entity enclosed in the body of the request message.

Note: If the URI does not point to an existing resource, a new resource is created with that URI

- An unsafe but idempotent operation

In other words, PUT is used to overwrite (update) a resource at a particular URI that is known to the client process issuing the request

Observe how a server-side URI is known by the client that issues the request

Main difference with **POST**: the request line URI identifies the **actual entity** to be updated or created, whereas in POST it identifies the entity to handle the request

So PUT can theoretically be used for creating a new server-side resource, but this would mean that the server gives to its clients freedom to control the creation of its own resources… not advisable…

DELETE Method

- Deletes the resource identified by the URI in the request line

  The request body is **insignificant**!

- An unsafe but idempotent operation

Describing REST Services

- We saw that WSDL is used to describe SOAP services
- But how can RESTful services be described?
- Common formalism:

One way is to use plain text documentation that describes the URIs that are to be used for accessing a service

OpenAPI (formerly Swagger): provides a standard way of describing RESTful APIs; typically written in YAML or JSON format.

Base URI

```yaml
openapi: 3.0.0
info:
  title: Example RESTful API
  version: 1.0.0
servers:
  - url: http://example.com/api/v1
```

GET operation & corresponding response

```yaml
paths:
  /users:
    get:
      summary: Get a list of users
      description: Returns a list of user objects
      responses:
        200:
          description: OK
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/User'
```

Describing REST Services

POST operation & corresponding response

```yaml
post:
  summary: Create a new user
  description: Creates a new user object
  requestBody:
    required: true
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/User'
  responses:
    201:
      description: Created
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/User'
```

```yaml
components:
  schemas:
    User:
      type: object
      properties:
        id:
          type: integer
        name:
          type: string
        email:
          type: string
```

Note: The POST operation is offered from the same path as the GET operation

Describing REST Services

- Common formalism (contd.):

RESTful API Modeling Language (**RAML**) is another specification for describing RESTful APIs in YAML or JSON format.

```
#%RAML 1.0
title: Example RESTful API
version: 1.0.0
baseUri: http://example.com/api/v1
mediaType: application/json
types:
  User:
    type: object
    properties:
      id: integer
      name: string
      email: string
```

Base URI & schema information

Note: The POST operation is offered from the same path as the GET operation

```
/users:
  get:
    description: Returns a list of user objects
    responses:
      200:
        body:
          application/json:
            type: User[]
  post:
    description: Creates a new user object
    body:
      application/json:
        type: User
    responses:
      201:
        body:
          application/json:
            type: User
```

GET operation & response

POST operation & response

Describing REST Services

- OK, but how such descriptions really differ from WSDL descriptions in SOAs?

In other words, how such descriptions lead to loosely coupled services that can communicate without the one hardcoding the communication requirements of the other?

- Well, they don't!

But then, what do we gain using REST? Why is REST considered to be ideal for developing loosely-coupled services that can communicate with each other?

Solution…

When a service A wants to consume a service B it contacts B and gets **dynamically i.e., on the fly,** information on how to access B's REST interface (i.e., which HTTP method to use, what parameters to pass, etc.)

I.e., instead of hardcoding this info into the consumer

Note: This is how browsers operate! Browsers do not hardcode the communication requirements of any of the websites that they visit. Instead, each time they visit a page they get dynamically from the web server, in the form of hyperlinks, the content of the page and the pages that can be navigated next

It is to be noted, however, that the dynamically provided hyperlinks that browsers get from web servers are navigated by **human users**…

Richardson's Maturity Levels

- Any use of REST that fails to comply with the above solution also fails to lead to generic services that can interact with each other without hardcoding their communication requirements
- The proposed solution imposes a **constraint** on how to use REST

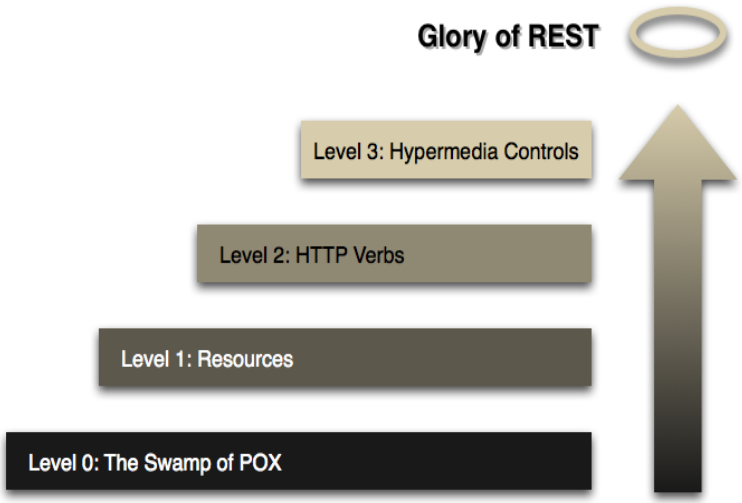
HATEOAS (Hypermedia as the Engine of Application State)

Richardson's maturity levels can be used to determine whether a RESTful interface abides by this constraint

see

https://martinfowler.com/articles/richardsonMaturityModel.html

Note: When consuming a service, the state of an application is defined as the set of interactions that it can engage in with the consumed service; these interactions are driven by hypermedia controls provided on the fly by the consumed service



HATEOAS

In the HATEOAS **architectural style**, HTTP responses from a server include **hypermedia links** that provide information about the next set of actions that can be taken by the client

The idea behind HATEOAS is that clients should be able to navigate through an application by following links provided in the response from the server, rather than relying on out-of-band (hardcoded) knowledge of the available resources and their URIs. This allows the server to evolve its API over time, adding or changing resources or actions without breaking existing clients

Note: These links typically include URIs that identify resources that can be accessed next, along with other metadata that describe the available actions and the state of the application

```
{
    "title": "My Blog",
    "description": "A simple blog API using HATEOAS",
    "links": [
        {
            "rel": "self",
            "href": "https://example.com/api/blog"
        },
        {
            "rel": "next",
            "href": "https://example.com/api/blog/posts",
            "title": "List of Posts",
            "type": "application/json",
            "methods": ["GET", "POST"],
            "auth": "required"
        }
    ]
}
```

How to contact link

HATEOAS

Note: The metadata that the server associates with each link could be even more specific allowing a client to post a blog…

So, when a client wants to create a new post, it follows the "create-post" link and retrieves the metadata for the resource. The client can then present a form to the user that includes input fields for the required parameters, based on the "fields" metadata provided by the server. When the user submits the form, the client constructs a POST request that includes the required parameters in the request body, and sends it to the server.

```
{
    "rel": "create-post",
    "href": "https://example.com/api/blog/posts",
    "title": "Create a New Post",
    "type": "application/json",
    "methods": ["POST"],
    "auth": "required",
    "fields": [
        {
            "name": "title",
            "type": "text",
            "required": true,
            "description": "The title of the post"
        },
        {
            "name": "content",
            "type": "textarea",
            "required": true,
            "description": "The content of the post"
        }
    ]
}
```

HATEOAS Benefits

- Web applications can alter their APIs without breaking their clients
  - They can change the URIs used
  - They can change how each resource is accessed
  - New capabilities/functionalities can be added on the fly

see

https://www.youtube.com/watch?v=nVEmfmdDUXU

Allows the development of generic client applications that can interact with any web app! Absolves with the need for prefab client-side SW that needs to be installed for contacting a server!

Is SOAP Dead?

- **No!**
- SOAP does have advantages
  - Not tied to HTTP
  - Built-in error support
  - Standards support
    - SOAP is associated with a multitude of WS-* Standards to support more sophisticated requirements
    - Such requirements are harder to satisfy in REST ("roll your own" approach)

WS-Security - adds enterprise security features (not supported by SSL):

- True end-to-end protection (SSL protects messages in transit)
- Partial encryption/signing

E.g. just encrypt the part of a message that carries a credit card number

WS-AtomicTransaction:

Provides support for ACID transactions

WS-ReliableMessaging

Provides successful/retry logic

SOAP is designed to provide a formal contract between the client and the server, using WSDL and XML schema. This can be useful when you need to **enforce a specific set** of rules for how data is exchanged between the client and server, such as in a **corporate environment** or when integrating with a **legacy system**

Appendix: YAML vs JSON

- **YAML** has a more expressive syntax that allows for more complex data structures, such as nested maps and lists
  - However, this can make it harder to read and understand
- **JSON** has a simpler syntax that is easy to read and understand and only supports arrays and key-value pairs

This makes it easier to parse and use in different programming languages

YAML allows for more flexibility in terms of indentation and formatting.

YAML offers support for nesting and references and allows for comments

YAML is more human readable

YAML is a superset of JSON!

Appendix: YAML vs JSON

```yaml
# YAML example
fruits:
  - apple
  - banana
  - strawberry
  - orange
  - grapes
  - kiwi

# JSON equivalent
{
  "fruits": [
    "apple",
    "banana",
    "strawberry",
    "orange",
    "grapes",
    "kiwi"
  ]
}
```

Nested data structure in YAML that is not supported in JSON:

a list of fruits represented as a YAML map with a key of "fruits" and a value of a nested list of fruits

Appendix: YAML vs JSON

```yaml
# YAML example with references
employees:
  - name: Alice
    id: 1
  - name: Bob
    id: 2
  - name: Charlie
    id: 3
manager:
  name: Dave
  id: 4
  employees:
    - *id-1   # reference to employee with id 1
    - *id-2   # reference to employee with id 2
    - *id-3   # reference to employee with id 3
```

References in YAML

Appendix: YAML vs JSON

```
# JSON equivalent without references
{
  "employees": [
    {
      "name": "Alice",
      "id": 1
    },
    {
      "name": "Bob",
      "id": 2
    },
    {
      "name": "Charlie",
      "id": 3
    }
  ],
```

```
  "manager": {
    "name": "Dave",
    "id": 4,
    "employees": [
      {
        "name": "Alice",
        "id": 1
      },
      {
        "name": "Bob",
        "id": 2
      },
      {
        "name": "Charlie",
        "id": 3
      }
    ]
  }
}
```

JSON equivalent…