Medii de proiectare și programare

2024-2025
Curs 10

Conținut

- Servicii Web
- REST
- Arhitectura REST
- REST Spring

Servicii Web

- Definiții:

1. Un **serviciu web** este o aplicație/componentă soft disponibilă pe internet și care folosește un sistem stardardizat de transmitere a mesajelor în format XML. XML este folosit pentru comunicarea datelor către/de la un serviciu web.
   - Un client apelează un serviciu web trimițând un mesaj în format XML și așteaptă un răspuns în format XML.
   - Comunicarea are loc folosind XML, serviciile web nu sunt dependente de un anumit sistem de operare sau de un anumit limbaj de programare.
2. **Serviciile web** sunt aplicații *self-contained*, modulare și distribuite care pot fi descrise, publicate, localizate și apelate prin rețea pentru a crea produse, procese, etc. Aceste aplicații pot fi locale, distribuite sau pe web. Serviciile web sunt dezvoltate folosind standarde ca TCP/IP, HTTP și XML.
3. **Serviciile web** sunt sisteme de schimbare a mesajelor bazate pe XML care folosesc Internetul pentru interacțiunea dintre aplicații. Aceste sisteme pot fi: programe, obiecte, mesaje, documente, etc.
4. Un **serviciu web** este o colecție de protocoale și standarde folosite pentru transmiterea datelor între aplicații sau sisteme. Aplicații dezvoltate în diferite limbaje de programare și rulând pe diferite platforme (sisteme de operare) pot folosi serviciile web pentru a schimba date în rețea (ex. Internet) într-o manieră similară comunicării între procese pe același calculator. Interoperabilitatea acestora este posibilă datorită folosirii standardelor.

Servicii Web

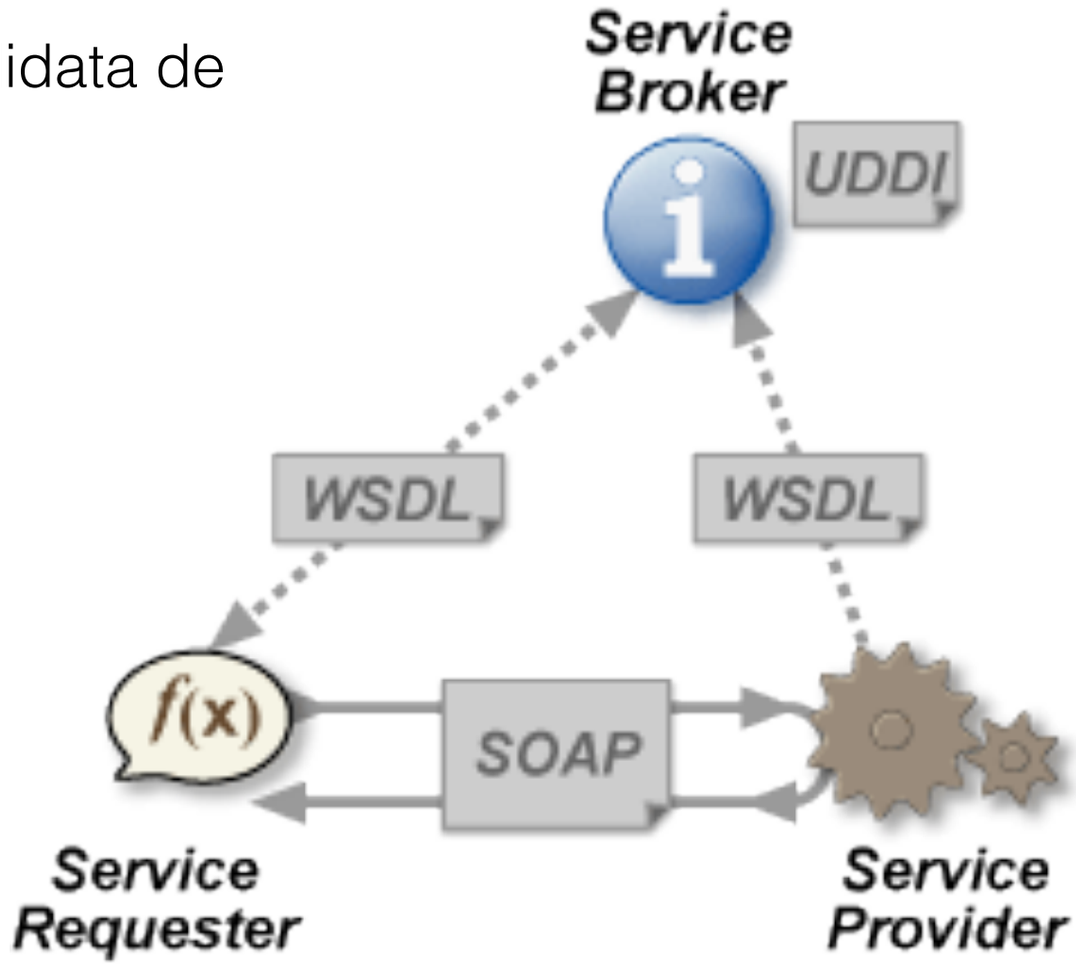
- Un serviciu web este un serviciu:
  - Disponibil pe Internet sau într-o rețea privată.
  - Folosește un sistem standardizat de schimbare a mesajelor bazat pe XML.
  - Nu este dependent de un anumit limbaj de programare sau de un anumit sistem de operare.
  - Este self-describing folosind o gramatică XML (DTD, XML Schema).
  - Poate fi descoperit folosind mecanisme simple.
- Componentele: structura de baza pentru serviciile web este XML + HTTP. Toate serviciile web standard funcționează folosind componentele:
  - XML pentru marcarea informației
  - SOAP (Simple Object Access Protocol) pentru transmiterea mesajelor
  - UDDI (Universal Description, Discovery and Integration)
  - WSDL (Web Services Description Language) pentru a descrie disponibilitatea serviciului

Exemplu WSDL

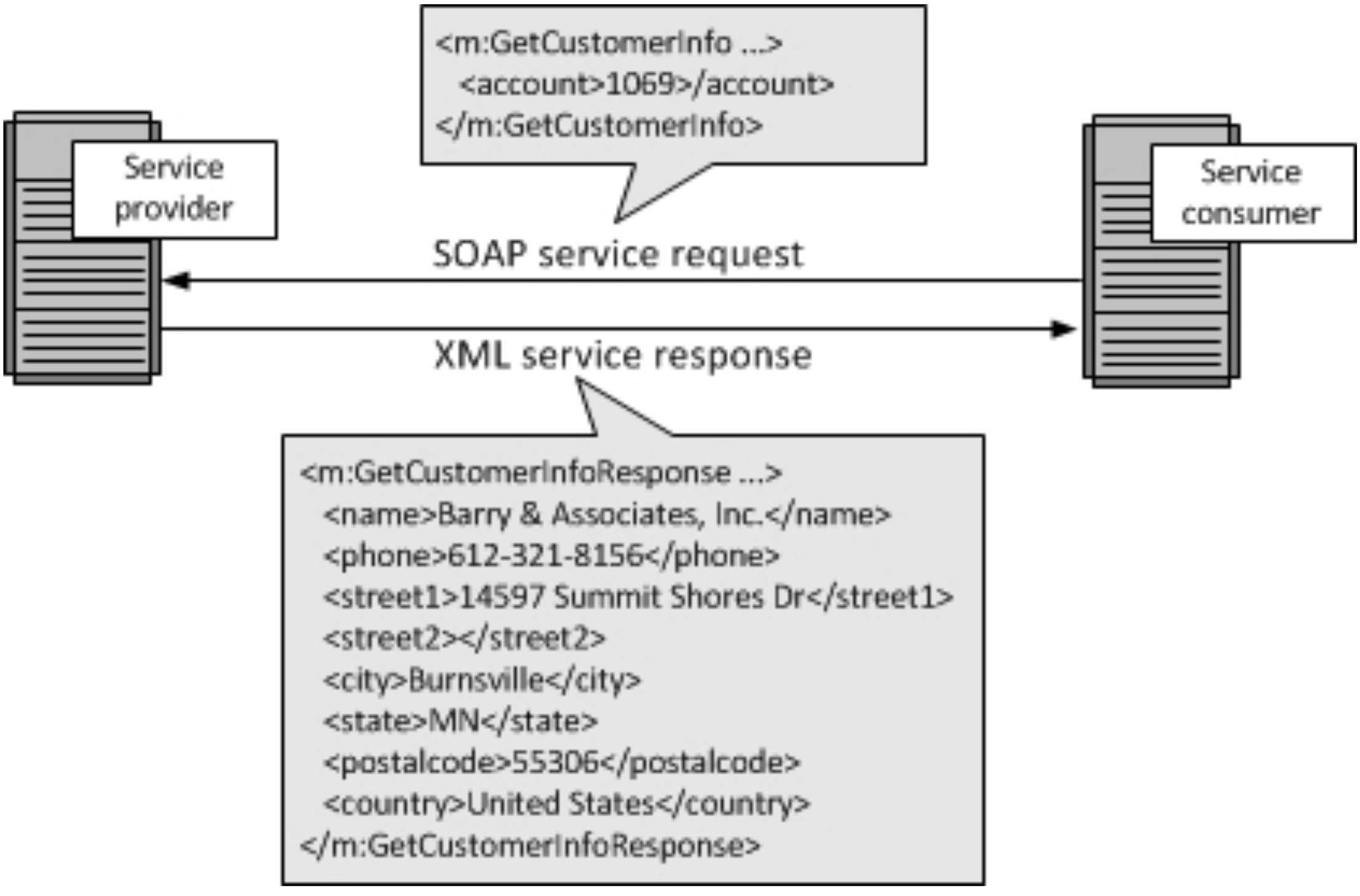
```
<definitions name = "HelloService"
   targetNamespace = "http://www.examples.com/wsdl/HelloService.wsdl"
   xmlns = "http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap = "http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns = "http://www.examples.com/wsdl/HelloService.wsdl"
   xmlns:xsd = "http://www.w3.org/2001/XMLSchema">

   <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>
```

Servicii Web - Arhitectura

- Furnizorul serviciului trimite un fișier WSDL serviciului UDDI.
- Clientul serviciului (consumatorul) contactează UDDI pentru a descoperi cine este furnizorul datelor de care are nevoie, iar apoi contacteaza furnizorul serviciului folosind protocolul SOAP.
- Furnizorul serviciului validează cererea și trimite datele cerute în format XML folosind protocolul SOAP.
- Structura datelor în format XML ar trebui validata de client folosind un fișier XSD (XML Schema).



Servicii Web - Arhitectura



REST

- REpresentational State Transfer (REST).
- Este un stil arhitectural pentru sisteme distribuite hipermedia.
- A fost propus de Roy T. Fielding în teza sa de doctorat "*Architectural Styles and the Design of Network-based Software Architectures*" (2000).
- Conține 6 constrângeri ce trebuie satisfăcute pentru a fi numit sistem REST veritabil:

1. Client–server
2. Fără stare (eng. *stateless*)
3. Cacheable
4. Interfață uniformă (eng. *uniform interface*)
5. Sistem stratificat (eng. *layered system*)
6. Cod la cerere (eng. *code on demand*) -opțională

REST - Resursa

- Conceptul de bază în REST este *resursa* (eng. *resource*).
- Orice informație care poate fi numită poate fi **resursă**: un document, o imagine, o colecție de alte resurse, un obiect real (ex. persoană, mașină), etc.
- REST folosește identificatorul resursei pentru a identifica resursa în momentul interacțiunii între componente.
- Starea unei resurse la un anumit moment de timp este numită ***reprezentarea resursei***.
- Reprezentarea constă din date, metadata care descriu datele și legături hipermedia care ajută clienții în tranziția la următoarea stare dorită.
- Formatul datelor dintr-o reprezentare este numit ***tip media*** (eng. *media type*).
- Tipul media identifică o specificație care definește modul în care reprezentarea va fi procesată.
- Un API RESTful veritabil seamănă cu hypertext. Fiecare informație ce poate fi obținută are o *adresă explicită* (prin legături și atribute id) sau *implicită* (obținută din definiția tipului media și structura reprezentării).
- Reprezentările resurselor trebuie să se descrie pe ele (eng. *self-descriptive*): clientul nu trebuie să știe că resursa este persoană sau mașină. Datele trebuie să poate fi procesate pe baza tipului media asociat resursei.

REST - metode asupra resursei

- Un lucru important asociat cu REST sunt metodele/operațiile ce pot fi aplicate resursei pentru a efectua tranziția dorită.
- Toată informația necesară modificării stării resursei face parte din API-ul cererii pentru acea resursă (inclusiv operațiile/metodele și modul în care vor afecta reprezentarea resursei).
- Este recomandat ca rezultatele interogărilor să fie reprezentate folosind o listă cu legături conținând informații sumare, și nu tablouri conținând toate informațiile corespunzătoare resursei, deoarece interogarea nu este un substitut pentru identificarea resurselor.
- Adesea, metodele/operațiile asupra resursei sunt confundate cu metodele HTTP GET/PUT/POST/DELETE.

REST și HTTP

- REST și HTTP nu reprezintă același lucru.
- Dacă un sistem satisface constrângerile REST, se poate spune ca sistemul este RESTful (modelul de maturitate Richardson, 2009).
- În stilul arhitectural REST, datele și funcționalitățile sunt considerate resurse ce pot fi accesate folosind *Uniform Resource Identifiers* (URIs).
- Resursele sunt prelucrate (adăugate, șterse, modificate, etc) folosind un set simplu, bine definit de operații.
- Clienții și serverele interschimbă (schimbă între ele) reprezentări ale resurselor folosind o interfață standardizată și un protocol standardizat (adesea HTTP).
- Resursele sunt decuplate de reprezentarea lor. Conținutul lor poate fi accesat folosind diferite formate: XML, text, PDF, JSON, HTML, etc.
- Metadata despre resurse este folosită pentru detecția erorilor, controlul caching-ului, negocierea celui mai convenabil format pentru reprezentare, autentificare și controlul accesului.
- Fiecare interacțiune este stateless.
- Aceste principii fac aplicațiile RESTful simple, lightweight și rapide.

Constrângerile arhitecturale REST

- REST este un stil arhitectural ce permite proiectarea unor aplicații slab cuplate (loosely coupled) folosind HTTP.
- Este adesea folosit pentru dezvoltarea serviciilor web.
- REST nu impune nici o regulă despre cum ar trebui implementat la nivelele inferioare, doar impune constrângeri la nivelul de proiectare și lasă implementarea dezvoltatorului.
- REST definește 6 constrângeri arhitecturale a căror satisfacere fac orice serviciu web un API RESTful veritabil:

1. *Uniform interface*
2. *Client–server*
3. *Stateless*
4. *Cacheable*
5. *Layered system*
6. *Code on demand* (opțional)

REST - *Uniform interface*

- Aplicând principiul generalității interfeței componentelor care interacționează, arhitectura întregului sistem este simplificată și interacțiunea între componente este îmbunătățită.
- Pentru a obține o interfață uniformă, se folosesc mai multe constrângeri arhitecturale pentru a ghida execuția componentelor.
- REST este definit de 4 constrângeri ale interfeței:
  - *Identificarea resurselor*
  - *Manipularea resurselor folosind reprezentările*
  - *Mesaje ce se descriu pe sine* (*self-descriptive*)
    - *Hypermedia ca și motor al stării aplicației (eng. hypermedia as the engine of application state - HATEOAS).*
- Dezvoltatorii trebuie **să decidă interfața** pentru resursele sistemului făcute publice clienților și **să folosească întotdeauna această interfață**.
- O resursă din sistem trebuie să aibă un singur URI, și trebuie să ofere o modalitate de a obține date adiționale sau asociate.
- O resursă nu ar trebui să fie foarte mare și ar trebui să conțină legături către alte informații adiționale (URI relativ) care permit obținerea acelor informații.
- Reprezentările resurselor trebuie să respecte anumite recomandări (convenții de nume, formatul legăturilor, formatul datelor -xml și/sau json).
- Toate resursele ar trebui să fie accesibile folosind o abordare comună (ex. folosind HTTP GET) și, în mod similar, ar trebui să poată fi modificate folosind o abordare consistentă.

REST - *Client–server*

- Aplicațiile client și server trebuie să poate evolua independent fără dependențe între ele.
- Clientul ar trebui să știe doar de URI-ul resursei.
- Prin separarea interfeței clientului de modul de păstrare a resurselor, se îmbunătățește portabilitatea interfeței cu utilizatorul pe platforme diferite și se îmbunătățește scalabilitatea prin simplificarea componentelor de pe server.
- "*Servers and clients may also be replaced and developed independently, as long as the interface between them is not altered*."

REST - *Stateless*

- Stilul REST a fost inspirat din HTTP.
- Toate interacțiunile client-server trebuie să fie ***stateless***.
- Serverul nu va păstra informații despre ultima cerere HTTP de la client.
- Fiecare cerere va fi tratată ca și cum ar fi o cerere nouă (fără sesiune, fără istoric).
- Fiecare cerere trimisă de la client la server trebuie să conțină toate informațiile necesare pentru a înțelege cererea și nu poate folosi informații legate de context păstrate pe server.
- Starea sesiunii este păstrată în întregime la client.
- Dacă o aplicație trebuie să păstreze starea pentru client (clientul se autentifică iar apoi efectuează operații), fiecare cerere de la client trebuie să conțină toate datele necesare pentru îndeplinirea cererii (inclusiv detalii legate de autentificare și autorizare).
- "*No client context shall be stored on the server between requests. Client is responsible for managing the state of application.*"

Constrângeri REST

- *Cacheable*: constrângerea cere ca datele dintr-un răspuns la o cerere să fie ***implicit sau explicit marcate ca cacheable sau non-cacheable***. Dacă răspunsul este cacheable, atunci cache-ul de pe client are permisiunea de a refolosi datele ulterior (echivalând cu obținerea lui folosind cereri către server).
- Caching îmbunătățește performanța clientului și îmbunătățește scalabilitatea serverului deoarece numărul de cereri se reduce.
- Caching trebuie aplicat tuturor resurselor ce se declară cacheable. Caching poate fi implementat atât pe server cât și pe client.
- "*Well-managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance.*"

- *Sistem stratificat:*
  - REST permite folosirea unei arhitecturi stratificate a sistemului: API public (URIs) este pe serverul A, datele sunt păstrate pe server B, iar cererile de autentificare se fac pe serverul C.
  - Un client nu știe dacă este conectat direct cu serverul destinație sau cu un intermediar.
- Arhitectura stratificată permite ca sistemul final să fie compus din nivele ierarhice care permit unui nivel comunicarea doar cu nivelul inferior (superior)

Constrângeri REST

- *Code on demand*
- Constrângerea este ***opțională.*** Adesea, serverele vor trimite reprezentarea statică a resurselor în format XML sau JSON.
- Serverul poate trimite și cod executabil, pentru a suporta o parte a aplicației (ex. cod pentru afișarea unei componente grafice)
- REST permite extinderea funcționalității clienților prin descărcarea și executarea codului (sub forma unor applet-uri sau a unor scripturi).
- Poate simplifica clienții prin reducerea numărului de funcționalități ce trebuie implementate ulterior.

Recomandări resurse REST

- Se recomandă folosirea consecventă a unei strategii pentru denumirea resurselor.
- API REST folosește URIs pentru accesarea resurselor. URI-urile folosite ar trebui să sugereze clienților modelul resurselor.
- Dacă resursele sunt denumite sugestiv, API-ul corespunzător va fi ușor de folosit și intuitiv. Altfel poate fi dificil de înțeles și utilizat.
- O resursă poate fi *singulară* sau *o colecție de alte resurse*.

  Exemplu:

  - “**`articles`**” - colecție de resurse
  - “**`article`**” - singular.
- Identificarea resurselor de tip colecție se poate face folosind substantivul la plural:

  Resursa “**`articles`**” : URI “**`/articles`**”.
- Identificarea unei resurse singulare se poate face folosind identificatorul:

  Resursa “**`article`**” are URI “**`/articles/{articleID}`**”.
- O resursă poate conține o colecție de alte resurse (subcolecție).

  ex. autorii unui articol pot fi identificați prin:

  “**`/articles/{articleID}/authors`**”.
- Identificarea unei resurse singulare dintr-o subcolecție:

  “**`/articles/{articleID}/authors/{authorID}`**”.

REST Recomandări

- *Folosirea substantivelor pentru reprezentarea resurselor*
- **URI RESTful trebuie să refere resursa, nu să se refere la o acțiune.** Substantivele au proprietăți, și asemănător resursele au atribute.

http://api.example.com/conference/articles

~~**http://api.example.com/conference/getArticles**~~ **- nu este URI corespunzator REST**

http://api.example.com/conference/articles/{id}/title

http://api.example.com/user-management/users

http://api.example.com/user-management/users/{id}

http://api.example.com/user-management/users/{id}/name

- *Folosirea consecventă a convențiilor pentru denumirea resurselor și a modului de formare a URI-ului pentru a reduce ambiguitățile și îmbunătățirea înțelegerii și a întreținerii*:
  - Folosirea caracterului / pentru a indica relații ierarhice
  - A NU se folosi caracterul / la finalul URI-ului

~~http://api.example.com/device-management/managed-devices/~~

http://api.example.com/device-management/managed-devices

REST Recomandări

- *Folosirea cratimei (-) pentru îmbunătățirea înțelegerii URI*

http://api.example.com/inventory-management/managed-entities/{id}/install-script-location

~~http://api.example.com/inventory-management/managedEntities/{id}/installScriptLocation~~

- *Folosirea literelor mici în URI*

http://api.example.org/my-folder/my-doc

~~http://api.example.org/My-Folder/my-doc~~

- *A nu se folosi extensii de fișier:* Nu aduc informații suplimentare clientului și pot induce în eroare dezvoltatorii.
- Pentru a preciza tipul media corespunzător unei reprezentări se folosesc antetele *Accept* și *Content-Type*.
- Acestea sunt folosite și pentru a determina modul de procesare a conținutului.

~~http://api.example.com/device-management/managed-devices.xml~~

http://api.example.com/device-management/managed-devices

REST Recomandări

- ***A nu se folosi nume de funcții CRUD (get, add/save, delete, update/modify) în URI-uri***
- URI-urile nu ar trebui folosite pentru a preciza operația/operațiile efectuate.
- URI-urile ar trebui folosite pentru a identifica în mod unic resursele, nu pentru a le manipula.
- Metodele HTTP ar trebui folosite pentru a indica operația CRUD ce trebuie efectuată.

```
//obținerea tuturor articolelor (echivalentul lui getAll/findAll)
HTTP GET si URI http://api.example.com/conference/articles
   //Crearea unui nou articol (echivalentul lui save/add)
HTTP POST si URI http://api.example.com/conference/articles
   //Obținerea articolului cu id-ul dat (echivalentul lui findOne)
HTTP GET si URI http://api.example.com/conference/articles/{id}
  //Actualizarea articolului cu id-ul dat (echivalentul lui update/modify)
HTTP PUT si URI http://api.example.com/conference/articles/{id}
  //Stergerea articolului cu id-ul dat (echivalentul lui delete)
HTTP DELETE si URI http://api.example.com/conference/articles/{id}
```

REST Recomandări

- *Folosirea parametrilor interogării HTTP pentru a filtra o colecție.*
  - Sortarea elementelor din colecție
  - Filtrarea elementelor din colecție
  - Limitarea numărului de elemente returnate (paginarea)
  - Nu se creează URI-uri noi, ci se folosesc parametrii de tip query:

```
http://api.example.com/conference/articles

http://api.example.com/conference/articles?domain=AI

http://api.example.com/conference/articles?domain=AI&subdomain=Genetic

http://api.example.com/conference/articles?
domain=AI&subdomain=Genetic&sort=submission-date

//gresit

http://api.example.com/conference/getArticlesSortedBy?
domain=AI&subdomain=Genetic&sort=submission-date
```

REST API - Proiectare

- *Pașii proiectării serviciilor REST*
  1. Identificarea modelului obiectual (a resurselor)
  2. Crearea modelului pentru URI-uri
  3. Determinarea reprezentărilor
  4. Atribuirea/asocierea metodelor HTTP
  5. Alte acțiuni (Logging, Security, Discovery)
- Identificarea modelului obiectual
- Identificarea obiectelor care vor fi prezentate ca și resurse:
  - User
  - Message,
  - Article,
  - Author,
  - Person
  - etc.

REST API Proiectare

- Crearea modelului URI-urilor
  - Deciderea URI-urilor pentru resursele identificate anterior.
  - Axarea pe relațiile dintre resurse și sub-resurse.
  - Aceste URI-uri asociate resurselor vor fi folosite ca și endpoint-uri pentru serviciile REST.
  - **URI-urile nu conțin verbe sau operații!!!**
  - **URI-urile ar trebui să conțină doar substantive!**

```
/articles
/articles/{id}

/authors
/authors/{id}

/articles/{id}/authors
/articles/{id}/authors/{aid}
```

REST API - Proiectare

- *Determinarea reprezentării resurselor*
  - Majoritatea reprezentărilor sunt definite în format XML sau JSON.

```
<author>
  <name> Pop Ion </name>
  <affiliation> ZY Lab </affiliation>
</author>
```

```
{
   “name”:”Pop Ion”
   “affiliation”: “ZY Lab”
}
```

- *Asocierea metodelor HTTP*
  - Obținerea tuturor articolelor sau autorilor

HTTP GET si URI **/articles**

HTTP GET si URI **/authors**

  - Paginarea, dacă rezultatul este prea mare.

HTTP GET si URI **/articles?startIndex=0&size=20**

HTTP GET si URI **/authors?startIndex=0&size=20**

  - Obținerea tuturor autorilor unui articol (subcolecție)

HTTP GET si URI **/articles/{id}/authors**

REST API -Proiectare

- Obținerea unui singur articol/autor

`HTTP GET` si URI **`/articles/{id}`**

`HTTP GET` si URI **`/authors/{id}`**

- Obținerea unui singur autor (subcolecție)

`HTTP GET` si URI **`/articles/{id}/authors/{id}`**

Reprezentarea subresursei poate fi diferită.

- Crearea unui articol/autor - metoda HTTP POST

`HTTP POST` si URI **`/articles`**

`HTTP POST` si URI **`/authors`**

IMPORTANT!

- **De obicei, la crearea unei resurse, corpul cererii NU conține informații legate de id, deoarece serverul este responsabil de determinarea acestuia.**
- **Exceptie: id-ul este informatie relevanta pentru resursa (username, CNP)**

REST API - Proiectare

- Actualizarea unui articol/autor - metoda HTTP PUT

HTTP PUT si URI **/articles/{id}**

HTTP PUT si URI **/authors/{id}**

- Ștergerea unui articol/autor - metoda HTTP DELETE

HTTP DELETE si URI **/articles/{id}**

HTTP DELETE si URI **/authors/{id}**

- Un răspuns ar trebui să returneze:
  - 202 (Accepted) dacă resursa a fost pusă într-o coadă și urmează a fi ștearsă (asincron).
  - 200 (OK) / 204 (No Content) dacă resursă a fost ștearsă permanent (sincron).
- Adăugarea unui autor la articol - Metoda HTTP PUT

  HTTP PUT si URI **/articles/{id}/authors**

- Ștergerea unui autor de la articol - Metoda HTTP DELETE

HTTP DELETE si URI **/articles/{articleId}/authors/{authorId}**

HTTP Response Status Code

- **2XX -Succes** acțiunea cerută de client a fost primită, înțeleasă, acceptată și procesată cu succes.
  - **200 OK**: Răspunsul standard pentru cereri HTTP executate cu succes.
  - **201 Created**: Cererea a fost îndeplinită și a fost creată o nouă resursă.
  - **202 Accepted**: Cererea a fost acceptată pentru procesare, dar procesarea nu a fost încă efectuată.
  - **204 No Content**: Serverul a procesat cererea cu succes, dar procesarea nu a returnat date.
  - **205 Reset Content:** Serverul a procesat cererea cu succes, dar nu a returnat date. Spre deosebire de răspunsul 204, acest răspuns necesită ca clientul să reafișeze vederea/informatia.
  - etc.

HTTP Response Status Code

- **4XX** Erori client:
  - **400 Bad Request:** Serverul nu poate sau nu va procesa cererea din cauza unei erori din partea clientului (ex. sintaxa cererii este greșită, conținutul cererii este prea mare, etc.)
  - **401 Unauthorized**: Similar cu 403, dar se folosește când este necesară autentificarea, care încă nu s-a efectuat sau a eșuat.
  - **403 Forbidden**: Cererea este validă, dar serverul refuză procesarea. (Utilizatorul nu are drepturile necesare pentru a accesa resursa respectivă).
  - **404 Not Found**: Resursa ceruta nu a fost găsită, dar poate deveni disponibilă în viitor. Sunt permise cereri ulterioare din partea clientului.
  - **405 Method Not Allowed**: Metoda cerută nu este disponibilă pentru resursa cerută. Ex. o cerere GET pentru un formular(eng. *form*) care necesită ca datele să fie transmise folosind POST, sau o cerere PUT pentru o resursă care poate fi doar citită.
  - **406 Not Acceptable**: Reprezentările disponibile pentru resursa ceruta nu sunt acceptate de client (folosind antetul *Accept*).
  - etc.
- **5XX** Erori la implementarea serviciilor

HTTP Media Types

- Antetele HTTP ***Accept*** și ***Content-Type*** pot fi folosite pentru a descrie reprezentarea folosită pentru schimbarea informației (reprezentarea resurselor).
- Clientul trimite o cerere în care în antetul ***Accept*** specifică lista tipurilor de reprezentare acceptate (separate prin virgulă).

  Exemplu : **`Accept: application/json, application/xml`**
- Serverul trimite răspunsul folosind ca și reprezentare primul tip mime înțeles din lista precizată (dacă înțelege unul sau mai multe). Tipul folosit este precizat în antetul ***Content-Type***.

  Exemplu: **`Content-Type: application/xml`**

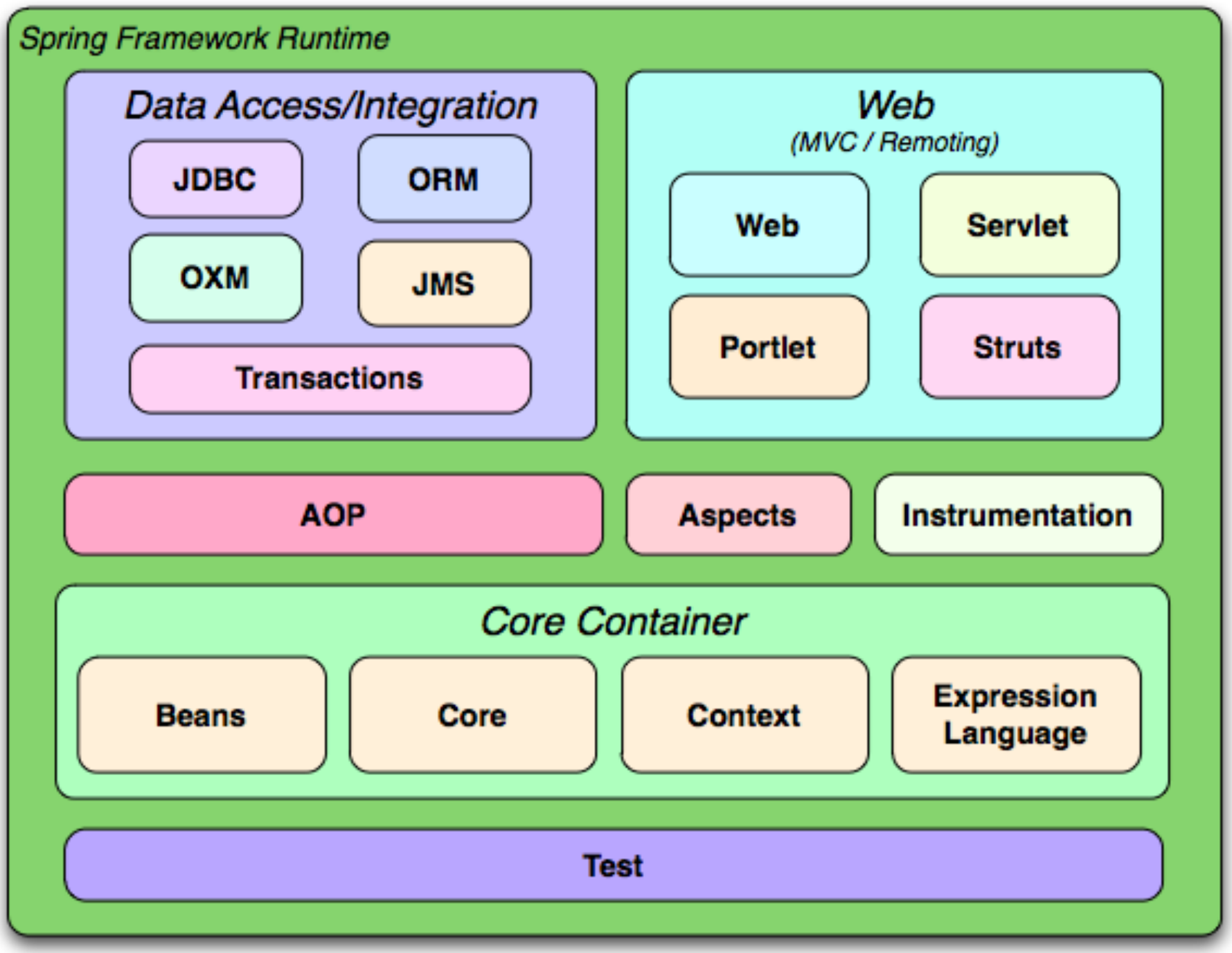
Referințe

- Roy Fielding, PhD Thesis, Chapter 5. Representational State Transfer (REST)

https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

- http://restfulapi.net/
- http://www.service-architecture.com/articles/web-services/
- https://martinfowler.com/articles/richardsonMaturityModel.html
- Alte tutoriale

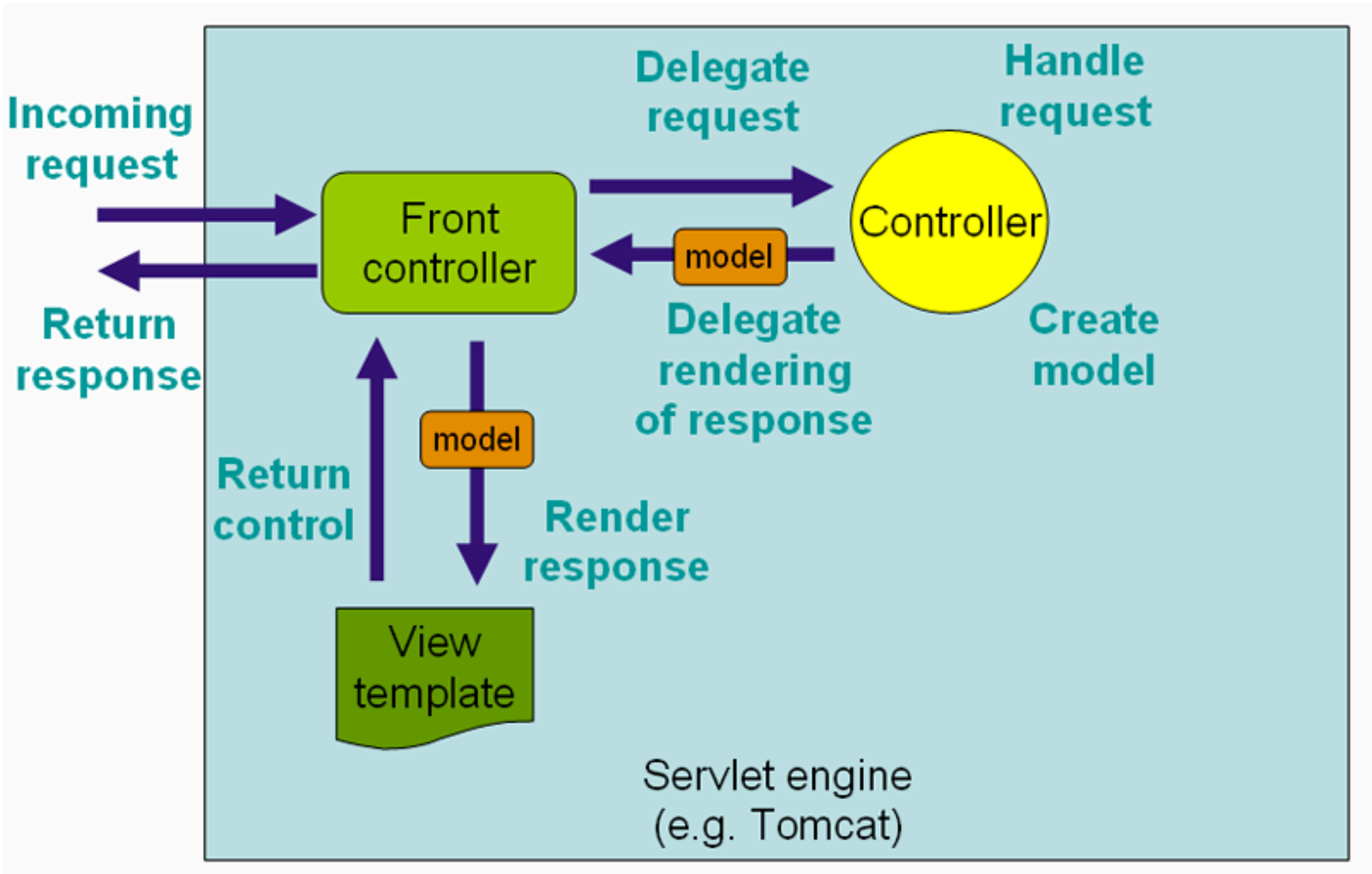
Spring Web MVC



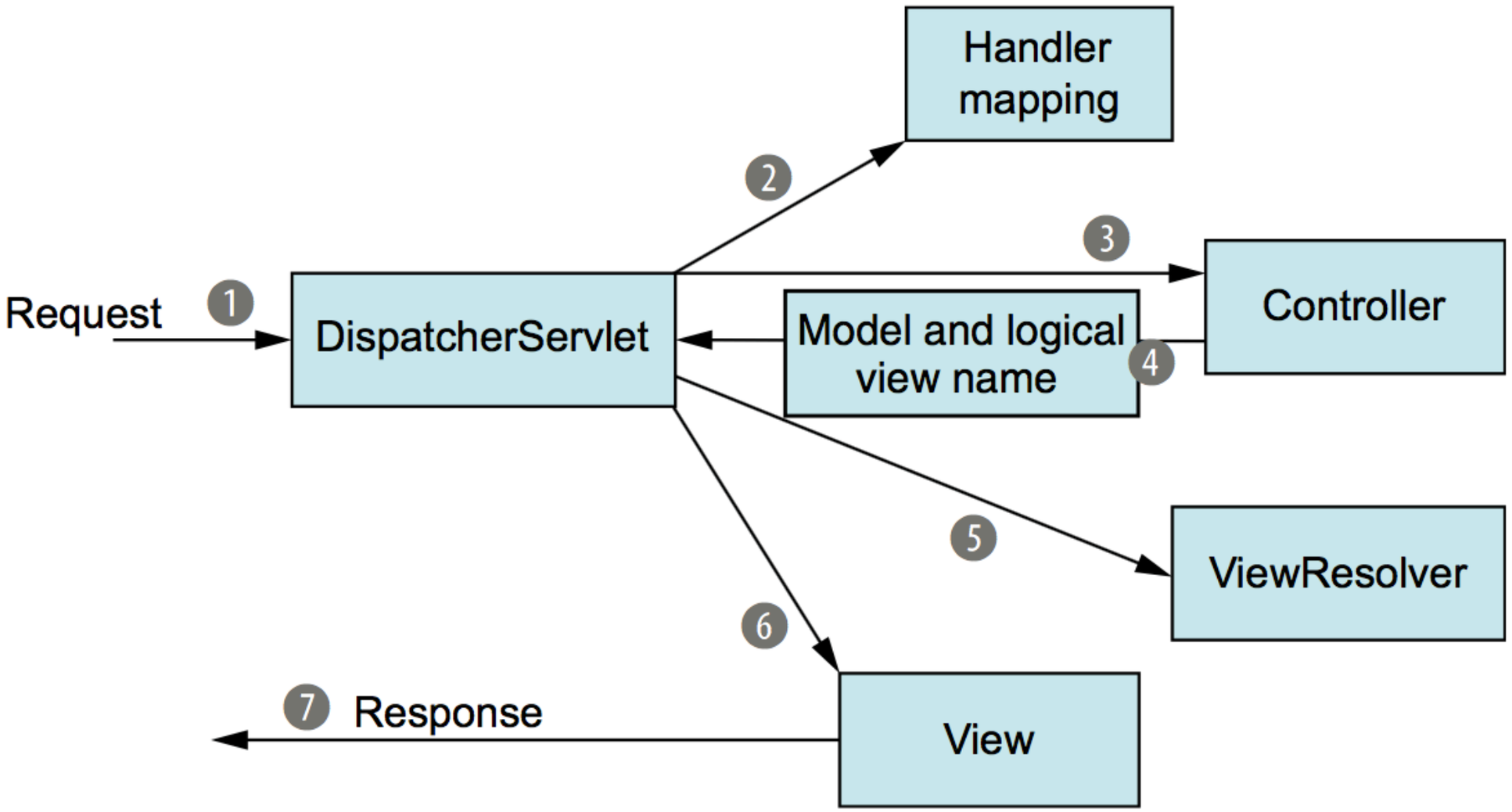
Spring Web MVC

- Frameworkul Spring Web model-view-controller (MVC) este proiectat în jurul unui *DispatcherServlet* care trimite cererile handler-elor, cu mapări configurabile, soluționarea vederilor (eng. *view*), a temei, a datelor și suport pentru încărcarea fișierelor.
- Handler-ul implicit folosește adnotările @Controller și @RequestMapping ce permit folosirea diferitor metode pentru procesarea cererilor.
- Orice obiect poate fi folosit ca și comandă sau pentru procesarea unei forme.
- Soluționarea vederilor este foarte flexibilă.
- Un controller este responsabil cu pregătirea modelului ce conține datele ce urmează a fi afișate și cu selectarea vederii folosită pentru afișare (folosind un nume pentru vedere).
- Soluționarea numelui asociat unei vederi se poate face prin extensii de fișiere, folosind antetul Accept, fișiere de proprietăți, bean-uri, etc.
- Modelul care păstrează datele este o interfață Map, care permite decuplarea de tehnologia folosită pentru afișarea acestora.

Spring Web MVC



Spring Web MVC



Spring Web MVC - RestController

- Clase adnotate cu **@RestController**
- Adnotarea **@RestController**: metodele adnotate din controller sunt considerate ca fiind HTTP endpoints și folosesc metadata furnizată de adnotarea **@RequestMapping** (**@GetMapping**, **@PostMapping**, etc) folosită la fiecare metodă/clasă pentru identificarea metodei ce trebuie să proceseze o cerere.
- O metodă va fi folosită pentru procesarea unei cereri HTTP daca cererea satisface condițiile precizate folosind adnotarea @RequestMapping/@GetMapping/… asociată metodei.
- Adnotarea @RestController furnizează și valori implicite (informații metadata) pentru toate metodele din clasa adnotată.
- Fiecare metodă poate decide suprascrierea adnotărilor specificate la nivel de clasă, dar anumite informații depind de context.

Spring Web MVC - RestController

```
@RestController

@RequestMapping(“/articles/{articleId}/authors”)

public class AuthorRestController {

     @RequestMapping(method = RequestMethod.GET) //@GetMapping

     Collection<Author> readAuthors(@PathVariable String articleId,

                @RequestParam(value=“sorted”, defaultValue=“name”)String sortCrit) { …   }

      @RequestMapping(method = RequestMethod.POST) //@PostMapping

     ResponseEntity<?> add(@PathVariable String articleId, @RequestBody Author input) { … }


     @RequestMapping(value = “/{authorId}”, method = RequestMethod.GET)

     Author readAuthor(@PathVariable String articleId, @PathVariable Long authorId) { …}


     private void validateUser(String userId) { ….   }

}
```

Spring Web MVC - RestController

- `{articleId}` și `{authorId}` sunt ***variabile de cale*** (eng. *path variables*). Sunt asemănătoare caracterelor speciale (eng. *wildcards*).
- Spring MVC va extrage acele părți din URI și le va face disponibile ca și parametrii ai metodelor ce procesează cererea. ***Parametrii vor avea același nume cu variabila de cale și sunt adnotați cu adnotarea @PathVariable.***

Exemplu: O cerere HTTP GET cu URI `/articles/art123/authors/4234`

  - Parametrul `@PathVariable String articleId` va avea valoarea `art123`,
  - Parametrul `@PathVariable Long authorId` va avea valoare `4234.`

- `@RequestParam` indică ***parametrii cererii*** (eng. *request query parameters*).

Exemplu: O cerere HTTP GET cu URI /`art123/authors/4234?sorted=affiliation`

  - `@RequestParam String sortCrit` va avea valoarea `affiliation`

Tipul parametrilor /returnat

- Datele returnate după procesarea cererii pot avea orice tip.
  - Tip primitiv/ obiect: informația va fi convertită la reprezentarea negociată de client și server (json, xml, text, etc).
  - @RequestBody: Datele transmise de client vor fi convertite la tipul precizat.
  - RequestEntity<E>: Este un wrapper pentru cererea primită de la client. Conține reprezentarea informației și datele adiționale din cererea HTTP (antetele, dimensiunea, etc).
- @ResponseBody: Obiectul returnat va fi convertit la reprezentarea negociată.
- ResponseEntity<E>: Este un wrapper pentru obiectul de tipul E și opțional poate seta tipul de răspuns (HTTP Response Status Code) și alte antete HTTP.

Tipul returnat

```
@RequestMapping(method = RequestMethod.GET)
String greeting(@RequestParam(value="name" defaultValue="") String name ){
        return "Hello "+name+"!";
 }

@RequestMapping(method = RequestMethod.GET)
@ResponseBody String greeting(@RequestParam(value="name" defaultValue="") String name ){
        return "Hello "+name+"!";
 }
@RequestMapping(method = RequestMethod.GET)
ResponseEntity<String> greeting(@RequestParam(value="name" defaultValue="") String name )
{
        return new ResponseEntity<>("Hello "+name+"!");
 }
@RequestMapping(method = RequestMethod.GET)
ResponseEntity<String> greeting(RequestEntity<String> request, @RequestParam(value="name"
defaultValue="") String name ){ …}
```

Convertirea mesajelor

- Convertirea mesajelor este o modalitate de a transforma datele obținute de controller/de la controller într-o reprezentare ce va fi transmisă clientului.
- Spring conține diferite clase pentru convertirea mesajelor, pentru a putea trata cele mai întâlnite conversii obiect-reprezentare.
  - **`BufferedImageHttpMessageConverter`**: Obiect **`BufferedImage`** din/într-o imagine (păstrată ca și un tablou de octeți).
  - **`MappingJackson2HttpMessageConverter`**: Obiecte simple și HashMaps (String, <? >) în format JSON.
  - **`MarshallingHttpMessageConverter`** (XML)
  - **`StringHttpMessageConverter`** (String, text/plain)

Tratarea excepțiilor

```
@RequestMapping(value = "/{id}", method = RequestMethod.GET)
public Article getById(@PathVariable String id){
    return articleRepository.findBy(id); //daca id-ul nu exista, returnează null
}
```

- Id-ul nu există, răspunsul va fi 200 (OK), iar conținutul va fi null.

Soluția 1

```
@RequestMapping(value = "/{id}", method = RequestMethod.GET)
public ResponseEntity<?> getById(@PathVariable String id){
    Article article=articleRepository.findBy(id);
    if (article==null)
        return new ResponseEntity<String>("Article not found",HttpStatus.NOT_FOUND);
    else
        return new ResponseEntity<Article>(article, HttpStatus.OK);
}
```

Tratarea excepțiilor

Soluția 2

```
@RequestMapping(value = "/{id}", method = RequestMethod.GET)
public ResponseEntity<Article> getById(@PathVariable String id){
    Article article=articleRepository.findBy(id);
    if (article==null)
       throw new RepositoryException("Article not found.");
    …
}
//2.a - in aceeași clasa cu metoda getById
@ExceptionHandler(RepositoryException.class)
@ResponseStatus(HttpStatus.NOT_FOUND)
public String repoException(RepositoryException e) { return e.getMessage(); }
//2.b
@ResponseStatus(HttpStatus.NOT_FOUND)
class RepositoryException extends RuntimeException {
    public RepositoryException(String message) {
        super(message);
    }
}
```

Pornirea aplicatiei

```
package conference;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class StartApplication {
    public static void main(String[] args) {
        SpringApplication.run(StartApplication.class, args);
    }
}
package conference;
@RestController
public class ArticleController{
  @Autowired private ArticleRepository articleRepository;
  // …
}
@Component
public class ArticleRepositoryJdbc implements ArticleRepository{ …}
```

Pornirea aplicatiei

- Adnotarea @SpringBootApplication specifică containerului Spring următoarele:
  - Adaugă adnotarea @Configuration clasei curente (specifică că poate conține definițiile unor bean-uri).
  - Adaugă adnotarea @EnableAutoConfiguration: specifică beanului Spring Boot să adauge beanuri în funcție de setările classpath-ului, a altor beanuri sau a altor proprietăți.
  - Adaugă adnotarea @EnableWebMvc necesară pentru a preciza ca este o aplicație SpringMVC.
  - Adaugă adnotarea @ComponentScan pentru căutarea altor bean-uri pornind de la pachetul "`conference`" (permite identificarea controllerelor REST). ("`conference`" reprezintă pachetul în care este definită clasă adnotată cu @SpringBootApplication)

- Metoda main() folosește SpringApplication.run() din Spring Boot pentru lansarea aplicației.
- **Serverul web (Tomcat) va fi inclus și pornit automat**.

Configurația Gradle

```
plugins {
    id 'org.springframework.boot' version '3.1.0'
    id 'io.spring.dependency-management' version '1.0.11.RELEASE'
    id 'java'
}

group 'Conference'
version '1.0'

repositories {
    mavenCentral()
}

dependencies {
     implementation project (':ConferenceModel')
     implementation project (':ConferencePersistence')

   implementation group: 'com.fasterxml.jackson.core', name: 'jackson-annotations', version: '2.13.1'

   implementation 'org.springframework.boot:spring-boot-starter-actuator'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    testImplementation('org.springframework.boot:spring-boot-starter-test')
    testImplementation('com.jayway.jsonpath:json-path')

}
```

Exemple

- ComputerShop
- Chat

Clienți REST

- API pentru apelarea unui serviciu REST:
  - Java
  - .NET
  - JavaScript
  - etc.
- Browser web
- Aplicații desktop (Extensii ale browserelor web):
  - POSTMAN
  - Advanced REST Client
  - RESTED
  - etc.

REST Client - RestTemplate (Spring)

- RestTemplate definește 36 de metode pentru interacțiunea cu resurse REST.

Method	Description
`delete()`	Performs an HTTP `DELETE` request on a resource at a specified URL
`exchange()`	Executes a specified HTTP method against a URL, returning a `ResponseEntity` containing an object mapped from the response body
`execute()`	Executes a specified HTTP method against a URL, returning an object mapped from the response body
`getForEntity()`	Sends an HTTP `GET` request, returning a `ResponseEntity` containing an object mapped from the response body
`getForObject()`	Sends an HTTP `GET` request, returning an object mapped from a response body
`headForHeaders()`	Sends an HTTP `HEAD` request, returning the HTTP headers for the specified resource URL
`optionsForAllow()`	Sends an HTTP `OPTIONS` request, returning the `Allow` header for the specified URL
`postForEntity()`	`POST`s data to a URL, returning a `ResponseEntity` containing an object mapped from the response body
`postForLocation()`	`POST`s data to a URL, returning the URL of the newly created resource
`postForObject()`	`POST`s data to a URL, returning an object mapped from the response body
`put()`	`PUT`s resource data to the specified URL

Spring REST Client

```
public class StartRestClient {

    public static void main(String[] args) {
        RestTemplate restTemplate=new RestTemplate();

     //Adding an article
        Article article=new Article(“AB CB","AI","Genetic");
        try{
        String articleId= restTemplate.postForObject("http://localhost:8080/
conference/articles",article, String.class);

        // Updating  article ...
        article.setTitle("New title");
        restTemplate.put("http://localhost:8080/conference/articles/"+articleId,
article);

       }catch(RestClientException ex){
            System.out.println("Exception ... "+ex.getMessage());
        }

    }
}
```

REST Client - RestClient (Spring >=v6.1)

- RestClient - permite apeluri sincrone către Endpoint-uri REST
- Usor de configurat și apelat

```
RestClient defaultClient = RestClient.create();  //configurare default
```

sau

```
//configurari specifice
RestClient customClient = RestClient.builder()
    .baseUrl(“https://example.com")
    .defaultHeader("My-Header", "Foo")
    .requestInterceptor(myCustomInterceptor)
  .build();
//apel
int articleId=123;
Article article = restClient.get()  //post(), delete(), put()
    .uri("https://conference.com/articles/{id}", articleId)
    .accept(APPLICATION_JSON)   //.contentType(APPLICATION_JSON).body(article)
    .retrieve()
    .body(Article.class);    //.toBodilessEntity();
```

.NET REST Client

- Clasa **`System.Net.Http.HttpClient`**
  - Trebuie adăugat (folosind NuGet) pachetul ***System.Net.Http***
  - ***HttpClient ar trebui instanțiat o singură dată într-o aplicație*** și refolosit pe parcursul rulării aplicației.
    Crearea unei noi instanțe pentru fiecare cerere, poate genera erori de tip SocketException (se atinge numărul maxim de conexiuni permise).

```
class MainClass{
  static HttpClient client = new HttpClient();
  public static void Main(string[] args){
     client.DefaultRequestHeaders.Accept.Clear();
      client.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("application/json"));
       // Get an article …;
      Article result = await GetArticleAsync(“http://localhost:8080/conference/articles/art124");
}
   static async Task<Article> GetArticleAsync(string path){
               Article article = null;
               HttpResponseMessage response = await client.GetAsync(path);
               if (response.IsSuccessStatusCode)
               {
                    var responseString = await response.Content.ReadAsStringAsync();
                     article = JsonConvert.DeserializeObject<Article>(responseString);
               }
               return article;
  }
}
```

Referințe

- https://learn.microsoft.com/en-us/dotnet/fundamentals/networking/http/httpclient
- Building REST services with Spring:
  https://spring.io/guides/tutorials/rest
- http://www.service-architecture.com/articles/web-services/
- Craig Walls, Spring in Action, 6th Edition, Ed. Manning, 2022
- Alte tutoriale