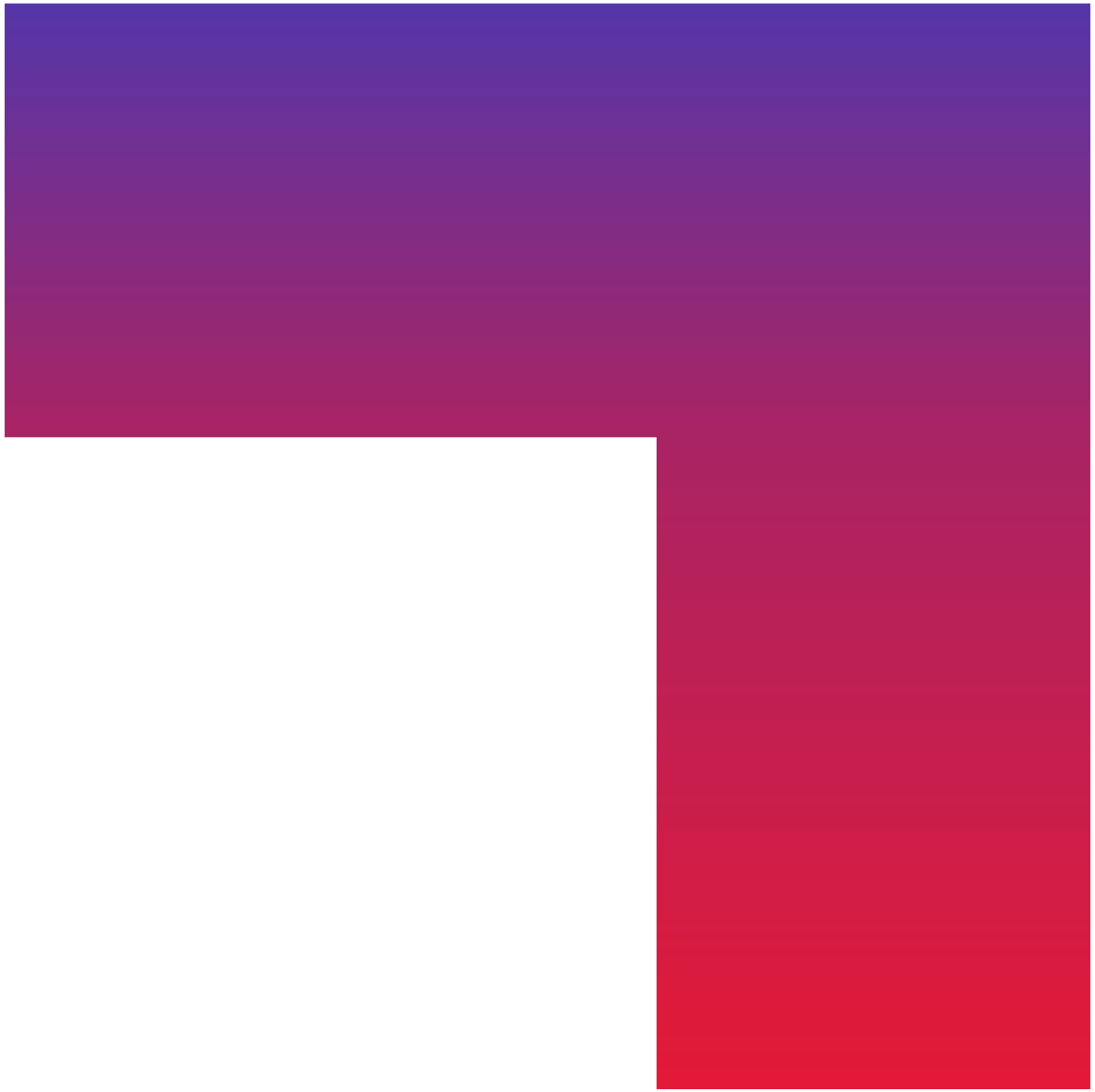
Spring REST

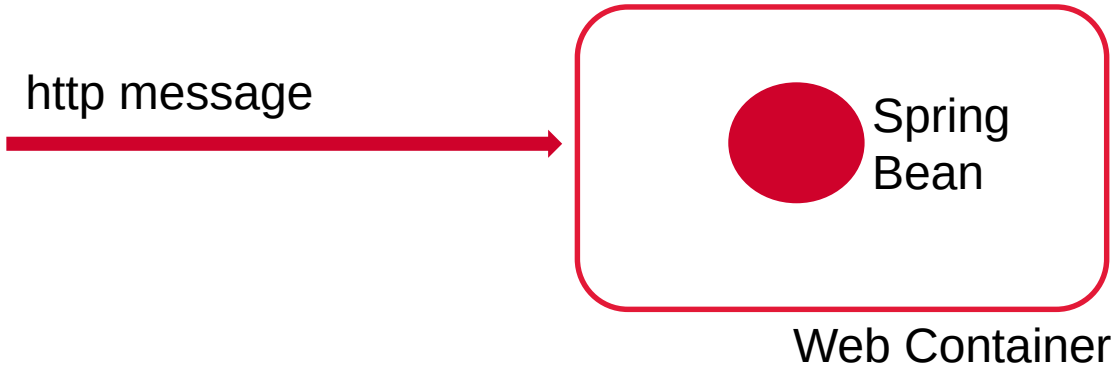


Sébastien Chassande-Barrioz
sebastien.chassande-barrioz@cgi.com

CGI

Spring REST, c’est quoi ?

- Librairie de Spring
- Exposition de service REST dans une container Web
- Repose sur les HttpServlet
- Une méthode du bean Spring traite une requête HTTP
- Conversion Java ⇔ JSON des objets paramètres et résultat



Un exemple

```
package hello;
import java.util.concurrent.atomic.AtomicLong;
import org.springframework.web.bind.annotation.*;

@RestController
public class GreetingController {

    @GetMapping("/greeting")
    public Greeting greeting(
            @RequestParam(value="name", defaultValue="World") String
name) {
        return new Greeting(String.format("Hello, %s!", name));
    }
}
```

- La méthode ‘greeting’ est appelé sur une requête HTTP GET à l’URL /greeting?name=toto.
- Le paramètre ‘name’ a une valeur par défaut.

Configuration par Annotations

- Sur la classe de service
  - @**RestController**() sur un bean spring pour déclarer le service comme un controller web REST. En Spring 3 il faut utiliser @Controller() et @ResponseBody.
  - @**CrossOrigin**(origins,maxAge) pour définir les URL autorisées
- Sur une méthode du service
  - @**RequestMapping**(value,method,produces) pour definir le mapping de la requête Web vers la méthode Java

    Variante Spring 4: @GetMapping, @PostMapping, @DeleteMapping, @PutMapping
  - @**ResponseBody** pour indiquer que le retourne de méthode sera le body de la réponse. (Inutile si RestController indiqué au niveau classe)
- Sur les paramètres d'une méthode
  - @**RequestParam**(value,defaultValue) : pour indiquer le nom du paramètre dans l'url: ?varName1=varValue1&varName2=varValue2)
  - @**PathVariable("bookId")** pour récupérer une partie de l'url variabilisée "/book/{boodId}"
  - @**RequestHeader** pour récupérer un paramètre transmis dans le Header
  - @**SessionAttribute** pour récupérer un paramètre de session

Requête avec pagination (1/3)

Le code Java :

```
@RestController
public class StudentDirectoryRestController {

    @Autowired    private StudentService service;

    @GetMapping(value = "/student/get", params = { "page", "size" })
    public Page<Student> findPaginated(
        @RequestParam("page") int page,
        @RequestParam("size") int size) {
        Page<Student> resultPage = service.findPaginated(page, size);
        if (page > resultPage.getTotalPages()) {
            throw new MyResourceNotFoundException();
        }
        return resultPage;
    }
}
```

Requête avec pagination (2/3)

Exemple de résultat :


```
{
    "content":[
            {"studentId":"1","name":"Bryan","gender":"Male","age":20},
            {"studentId":"2","name":"Ben","gender":"Male","age":22},
            {"studentId":"3","name":"Lisa","gender":"Female","age":24},
            {"studentId":"4","name":"Sarah","gender":"Female","age":26},
            {"studentId":"5","name":"Jay","gender":"Male","age":20}    ],
    "last":false,
    "totalElements":20,
    "totalPages":4,
    "size":5,
    "number":0,
    "sort":null,
    "first":true,
    "numberOfElements":5
}
```

Requête avec pagination (3/3)

- La structure JSON :
  - last – set to true if its the last page otherwise false
  - first – set to true if it’s the first page otherwise false
  - totalElements – the total number of rows/records. In our example, we passed this to the ui-grid options $scope.gridOptions.totalItems to determine how many pages will be available
  - totalPages – the total number of pages which was derived from (totalElements / size)
  - size – the number of records per page, this was passed from the client via param size
  - number – the page number sent by the client, in our response the number is 0 because in our backend we are using an array of Students which is a zero-based index, so in our backend, we decrement the page number by 1
  - sort – the sorting parameter for the page
  - numberOfElements – the number of rows/records return for the page

Renvoyer un fichier

Voici un exemple de méthode récupérant un binaire de fichier en appelant un Bean métier et copiant le binaire dans la HttpServletResponse

```
@GetMapping(value = "/recupererPieceJointe", method = RequestMethod.GET)
public void getPieceJointe(
        @RequestParam("idFichier") final String idFichier,
        final HttpServletResponse response) {
    final PieceJointeBO pj = piecesJointesBS.getPieceJointe(idFichier);
    if (pieceJointe == null) {
        response.setStatus(HttpServletResponse.SC_NOT_FOUND);
    } else {
        final byte[] file = pj.getContenuFichier();
        response.setContentType("application/octet-stream");
        response.setHeader("Content-Disposition",
                "attachment;filename=\"" + pj.getNomFichier() + "\""));
        response.setContentLength(file.length);
        final InputStream inputStream = new ByteArrayInputStream(file);
        try {
            FileCopyUtils.copy(inputStream, response.getOutputStream());
        } catch (final IOException e) {
            response.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
            LOGGER.error(e.getMessage(), e);
        }
    }
}
```

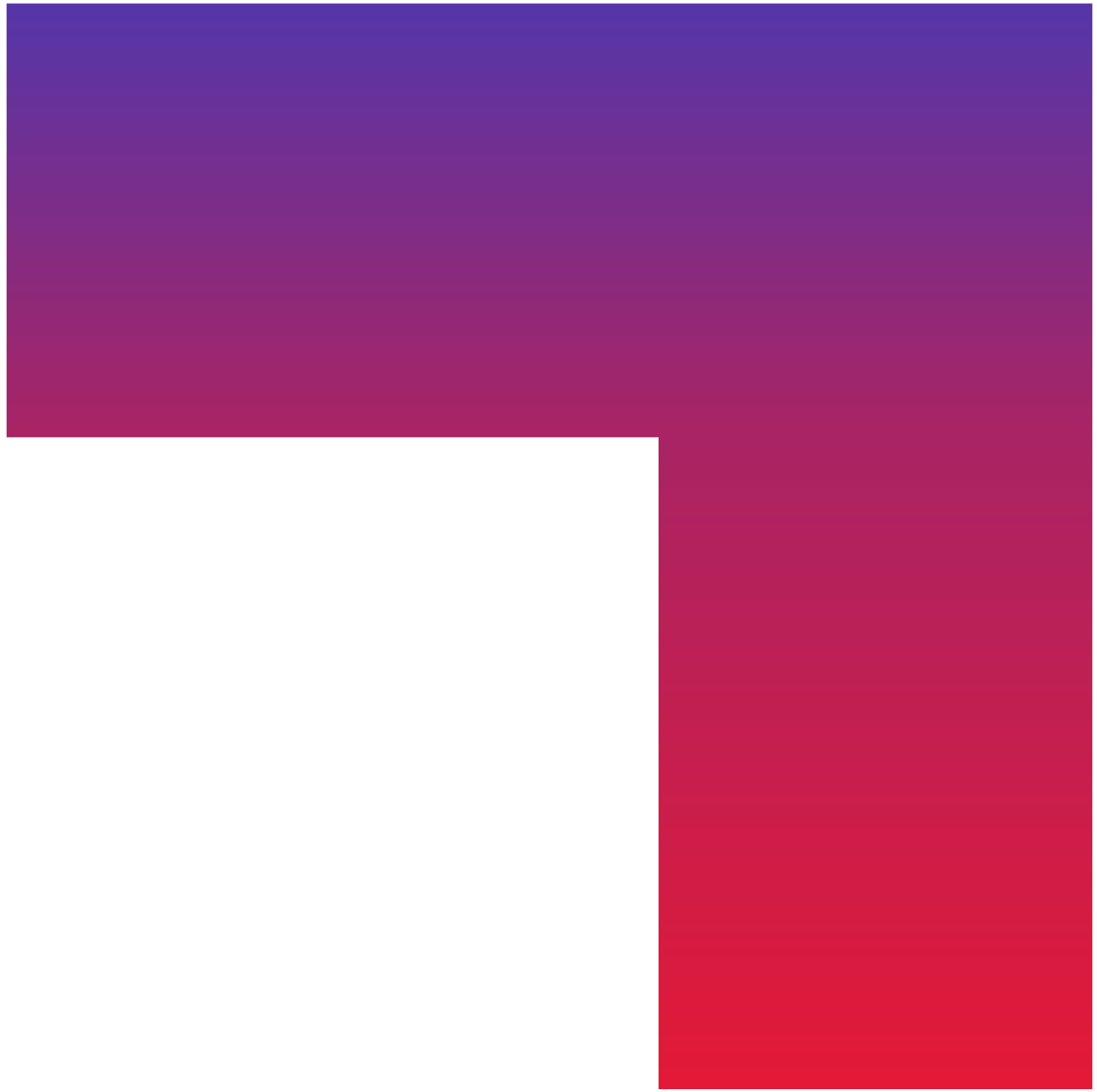
API Design

ou

Comment choisir

les URLS REST ?

Sébastien Chassande-Barrioz
sebastien.chassande-barrioz@cgi.com



Design d’API

- WOA : Web Oriented Architecture
- Orientation « Resource » ➔ REST
- Enjeux
  - respecte les standards HTTP,
  - s’inspire des bonnes pratiques des Géants du Web
  - Usage simple : KISS
  - Bonne affordance = capacité d’un objet à suggérer sa propre utilisation
    - ➔ DX Developer eXperience

REST : KISS

- Sémantique d'une API doit être **intuitive**,
  - URI, Payload ou données retournées
  - Exploiter l'API en ayant recours le moins possible à la documentation de l'API.
- Les termes utilisés doivent être **usuels** et **concrets** : *clients, orders, addresses, products,…* et non tirés d'un *"jargon fonctionnel"*.
- Il convient de ne pas proposer aux utilisateurs de pouvoir faire quelque chose de **plusieurs manières différentes**.
- L'API est "**designée" pour les clients**
  - Une erreur fréquemment rencontrée est de designer son API à partir d'un modèle de données existant, qui est souvent complexe.
- focaliser en premier lieu sur les *"uses-cases"* principaux,

REST : Granularité (1/2)

- théorie « une ressource = une URL »
  - ➔ fort découpage
- Exemple
  - Utilisateur
  - Son adresse
  - Son pays

```
CURL https://api.fakecompany.com/v1/users/1234
< 200 OK
< {"id":"1234", "name":"Antoine Jaby", "address":"https://api.fakecompany.com/v1/addresses/4567"}

CURL https://api.fakecompany.com/addresses/4567
< 200 OK
< {"id":"4567", "street":"sunset bd", "country": "http://api.fakecompany.com/v1/countries/98"}

CURL https://api.fakecompany.com/v1/countries/98
< 200 OK
< {"id":"98", "name":"France"}
```

- Mais accès systématique ➔ inefficacité / non performance
- Cas inverse : objet trop volumineux, information inutile

REST : Granularité (2/2)

Critères de choix

- Ne grouper que les ressources qui seront accédées à la suite de manière quasi systématique.

- Ne pas grouper les collections pouvant avoir de nombreux composants.
  - la liste des emplois courants sont limités (une personne ne peut pas cumuler beaucoup plus que 2 ou 3 emplois à temps partiel),
  - par contre, la liste des expériences professionnelles peut être très longue.

- Se limiter à deux niveaux d'imbrication :
  – /v1/users/addresses/countries

REST : Nommage de l’URI (1/2)

utiliser des noms concrets, pas de verbe

Approche RPC	Approche REST
getClient(1)	GET /clients/1
creerClient()	POST /clients
majSoldeCompte(1)	PATCH /accounts/1
ajoutetProduitDansCommande(1)	PUT /orders/1
effacerAdresse(1)	DELETE /addresses/1

Utiliser les methodes HTTP pour designer l’action :

- GET : Lire
- POST : Créer
- PUT : Modifier
- DEL : Supprimer
- PATCH : Mise à jour partielle

REST : Nommage de l'URI (2/2)

Convention : Usage du Pluriel

- Collection de ressources : /v1/users
- Instance d'une ressource : /v1/users/007

Casse des URI

- CamelCase
  - lowerCamelCase
  - UpperCamelCase
- snake_case
- spinal-case

	Google	Facebook	Twitter	Paypal	Amazon	dropbox	github
snake_case	x	x	x			x	x
spinal-case	x			x			
camelCase	x						

Choix ➔ spinal-case car RFC3986

- Exemple : POST /v1/specific-orders

REST : Versioning

- Versioning
  - Il y aura des évolutions
  - Coéxistance des versions
- 2 solutions :
  - Numéro de version sur un digit, au plus haut niveau du *path* de *l'uri*.
    - http://api.linkedin.com/v1/people/
  - Numéro de version dans un champ du header
- Changement de version majeure ➔ changement pour que l'API fonctionne
- Si API compatible (grâce à REST et Json en particulier) ALORS pas de changement de version
- Supporter au plus, 2 versions en parallèle

Résultat partiel, Filtre, Tri, Recherche

- Résultat Partiel :
  - Limiter le contenu de la ressource
  - ➔ Économie / allégement
  - *?fields=attribute1,attributeN*
- *Filtre :*
  - Limiter le nombre de ressource
  - https://api.fakecompany.com/v1/restaurants?type=japanese,chinese&rating=4
- Tri
  - Trier les ressources renvoyées
  - https://api.fakecompany.com/v1/restaurants?sort=rating,reviews,name
- Recherche
  - https://api.fakecompany.com/v1/search?q=running+paid

```
GET /clients/007?fields=firstname,name
200 OK
{
  "id":"007",
  "firstname":"James",
  "name":"Bond"
}
```

“Non-Resources” scenarios

- RESTful : Toute requête doit être vue et manipulée comme une ressource
- Et les pour les opérations comme la traduction, le calcul, la conversion, ou des services métiers parfois complexes inhérents à un SI ?

➔votre opération doit être représentée par un verbe

```
POST /calculator/sum
[1,2,3,5,8,13,21]
< 200 OK
< {"result" : "53"}
```

```
POST /convert?from=EURato=USD&amount=42
< 200 OK
< {"result" : "54"}
```

/! C’est un cas d’exception, à éviter

Gestion des erreurs

Utilisation du code HTTP de retour + code dans le body

```
{
  "error": "description_courte",
  "error_description": "description longue, Human-readable",
  "error_uri": "URI vers une description détaillée de l'erreur sur le portail developper"
}
```

Les codes d’erreurs HTTP classiques :

Code	Description
200	OK
201	Création (POST) ok
400	Bad request
401	Unauthorized : utilisateur inconnu
403	Fobidden : pas assez de droit
404	Not found : ressource introuvable
500	Server error : erreur technique/interne au serveur