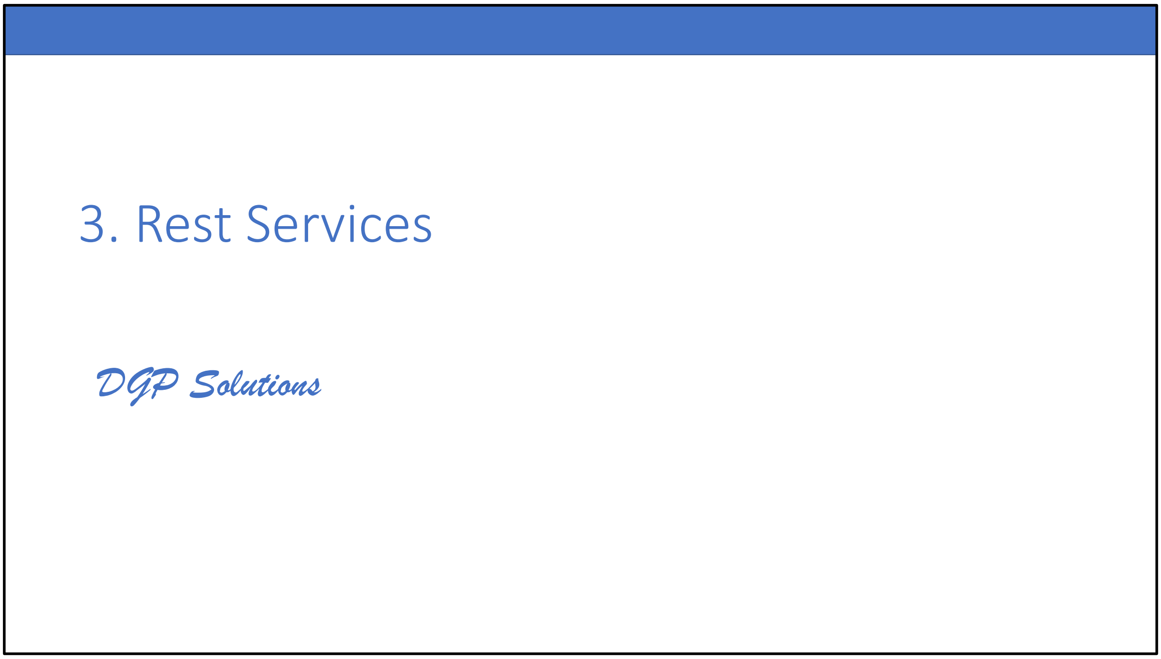
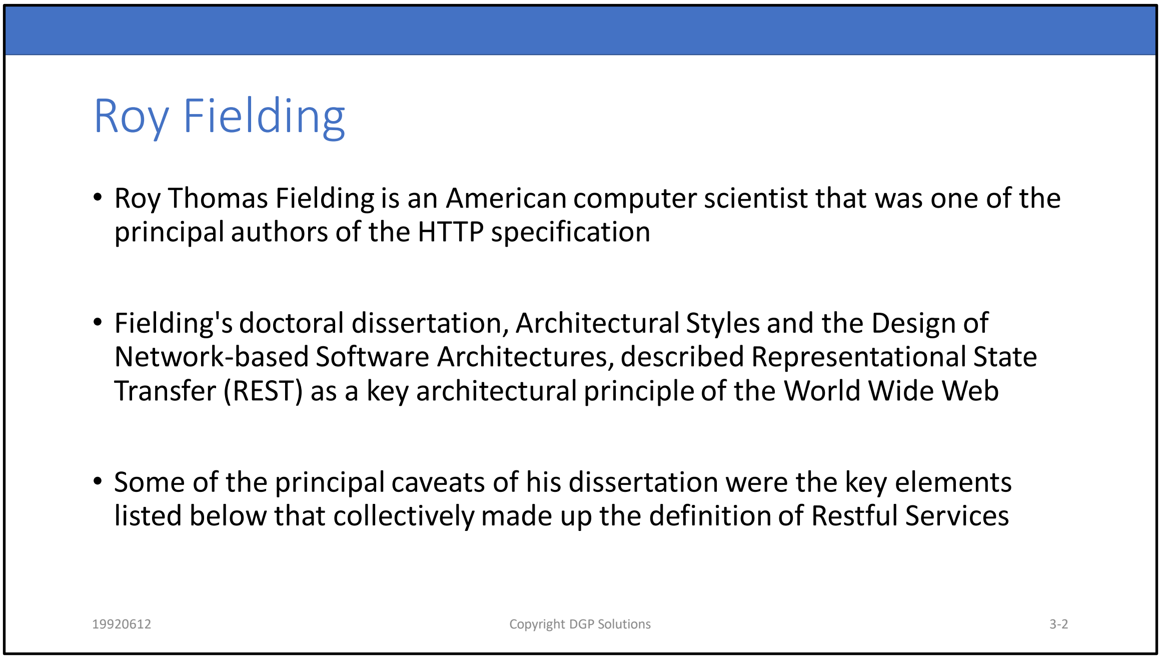


3. Rest Services

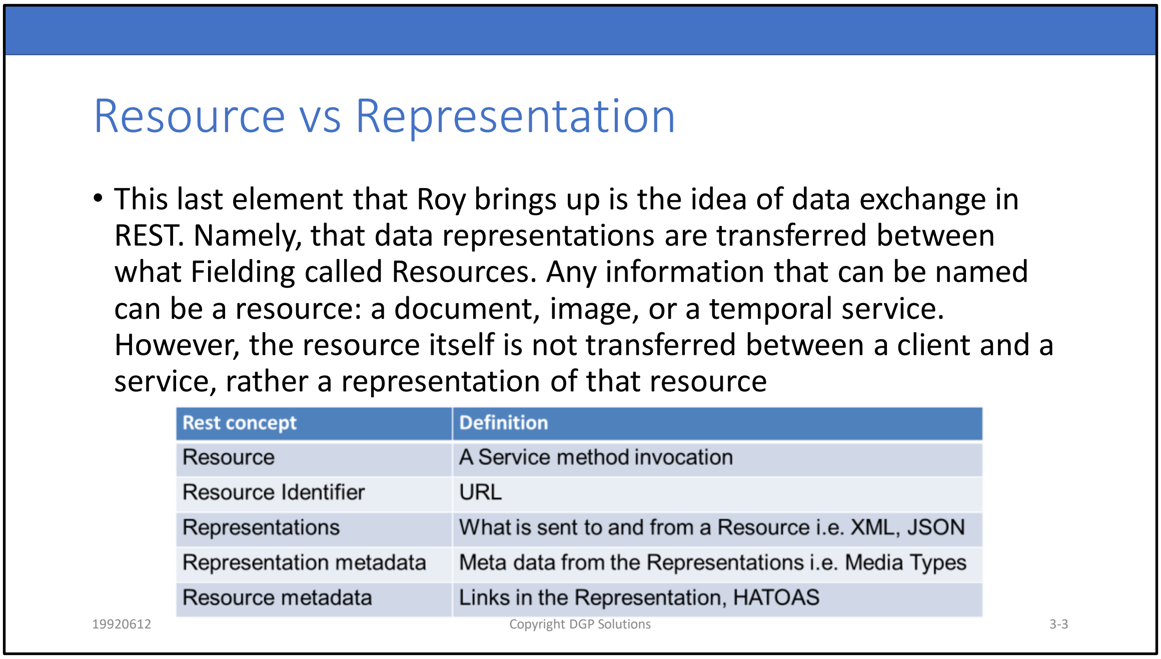
DGP Solutions



Roy Fielding

- Roy Thomas Fielding is an American computer scientist that was one of the principal authors of the HTTP specification
- Fielding's doctoral dissertation, Architectural Styles and the Design of Network-based Software Architectures, described Representational State Transfer (REST) as a key architectural principle of the World Wide Web
- Some of the principal caveats of his dissertation were the key elements listed below that collectively made up the definition of Restful Services

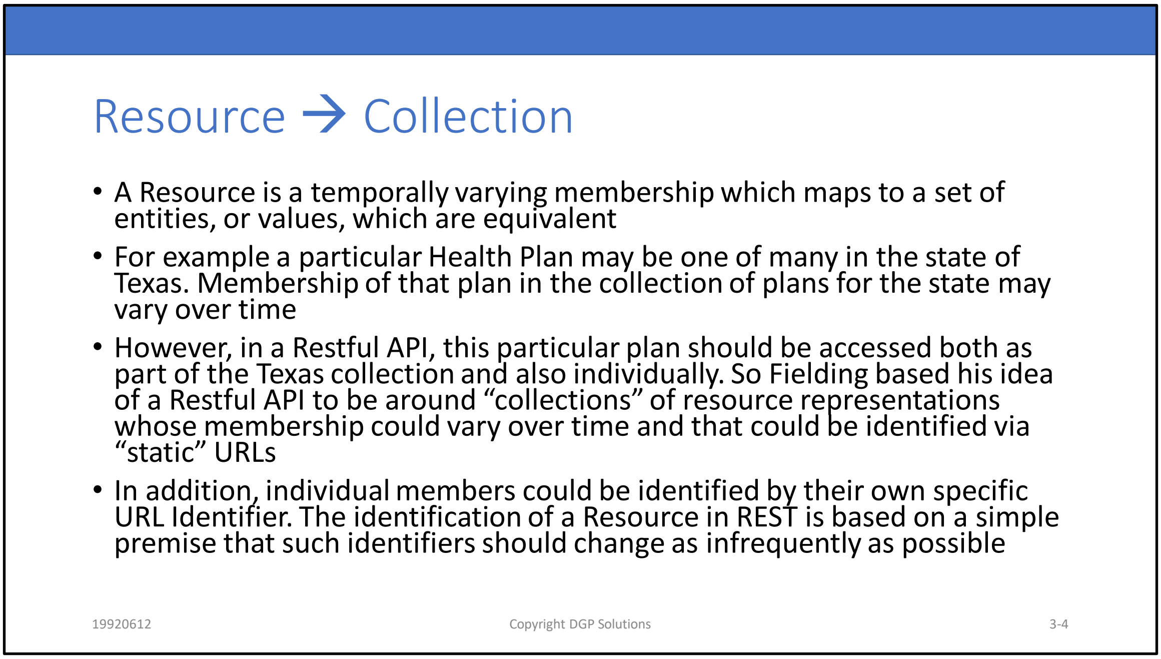
- The Separation of Concerns principal where the user interface is separated from data storage concerns
- Communication must be stateless in nature. Visibility is improved because a monitoring system does not have to look beyond a single request datum in order to determine the full nature of the uniform interface between components
- A layered system constraints architecture allows components to be composed of hierarchical layers where each component cannot "see" beyond the immediate layer with which they are interacting
- REST components communicate by transferring a representation of a resource in a format matching one of an evolving set of standard data types that are selected dynamically based on the capabilities or desires of the recipient



Resource vs Representation

- This last element that Roy brings up is the idea of data exchange in REST. Namely, that data representations are transferred between what Fielding called Resources. Any information that can be named can be a resource: a document, image, or a temporal service. However, the resource itself is not transferred between a client and a service, rather a representation of that resource

Rest concept	Definition
Resource	A Service method invocation
Resource Identifier	URL
Representations	What is sent to and from a Resource i.e. XML, JSON
Representation metadata	Meta data from the Representations i.e. Media Types
Resource metadata	Links in the Representation, HATOAS

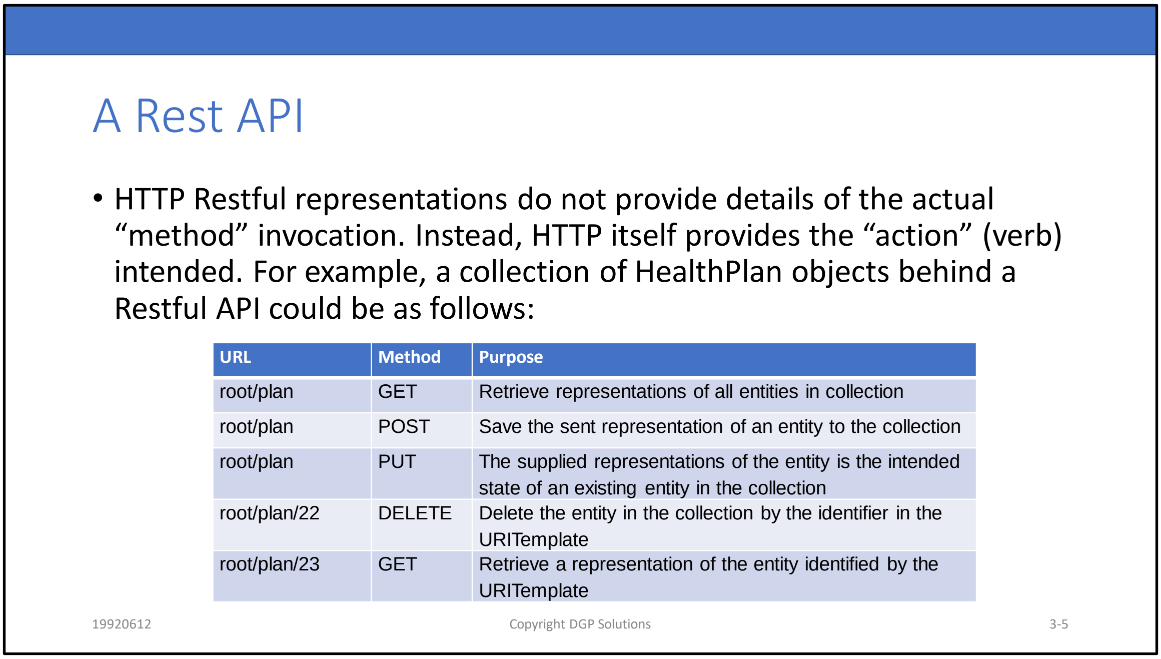


Resource → Collection

- A Resource is a temporally varying membership which maps to a set of entities, or values, which are equivalent
- For example a particular Health Plan may be one of many in the state of Texas. Membership of that plan in the collection of plans for the state may vary over time
- However, in a Restful API, this particular plan should be accessed both as part of the Texas collection and also individually. So Fielding based his idea of a Restful API to be around “collections” of resource representations whose membership could vary over time and that could be identified via “static” URLs
- In addition, individual members could be identified by their own specific URL Identifier. The identification of a Resource in REST is based on a simple premise that such identifiers should change as infrequently as possible

19920612 Copyright DGP Solutions 3-4

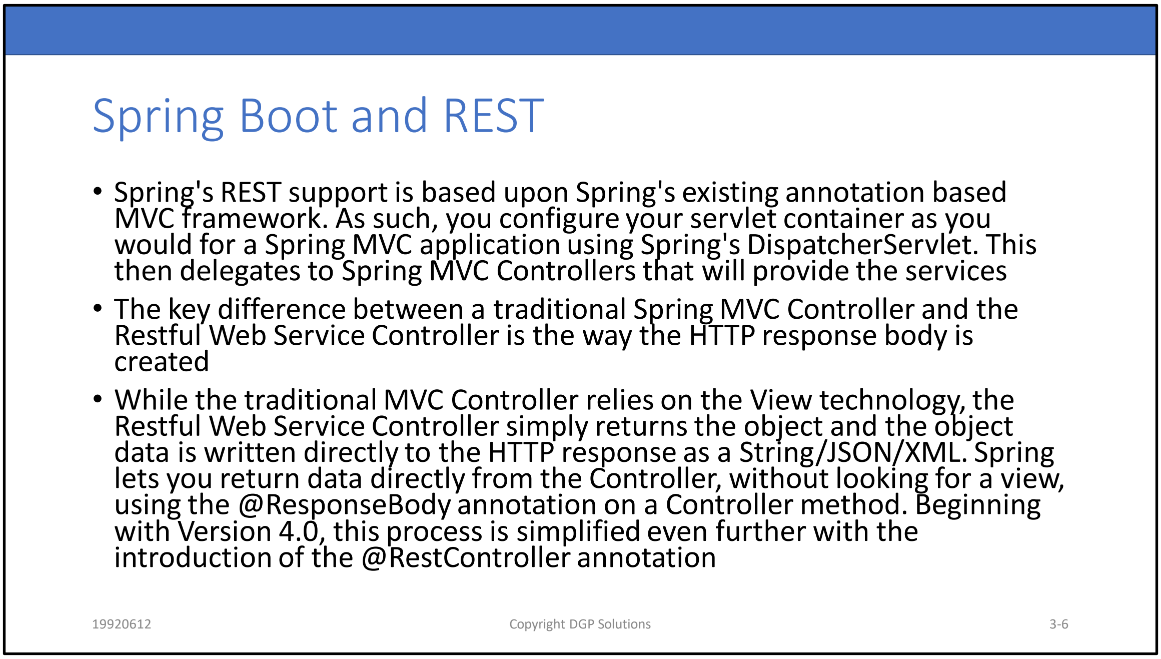
REST components perform actions on a resource by using a representation to capture the current or intended state of that Resource and transfer that representation between components. A representation is a sequence of bytes, plus its metadata to describe those bytes such as XML and JSON



A Rest API

- HTTP Restful representations do not provide details of the actual “method” invocation. Instead, HTTP itself provides the “action” (verb) intended. For example, a collection of HealthPlan objects behind a Restful API could be as follows:

URL	Method	Purpose
root/plan	GET	Retrieve representations of all entities in collection
root/plan	POST	Save the sent representation of an entity to the collection
root/plan	PUT	The supplied representations of the entity is the intended state of an existing entity in the collection
root/plan/22	DELETE	Delete the entity in the collection by the identifier in the URITemplate
root/plan/23	GET	Retrieve a representation of the entity identified by the URITemplate



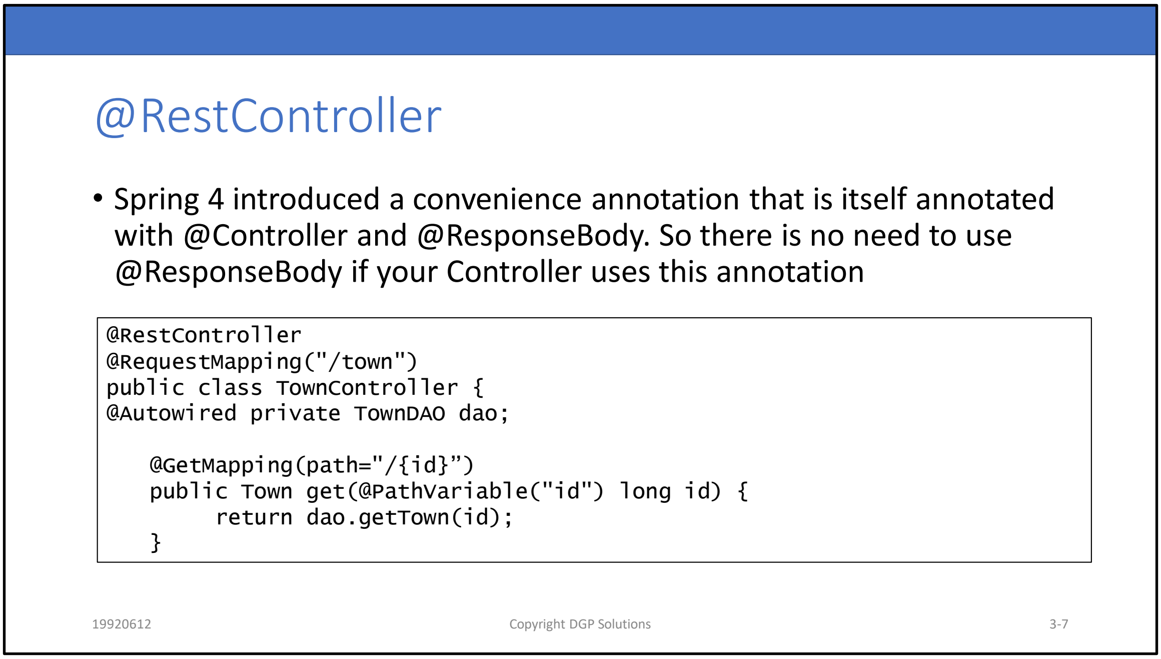
Spring Boot and REST

- Spring's REST support is based upon Spring's existing annotation based MVC framework. As such, you configure your servlet container as you would for a Spring MVC application using Spring's DispatcherServlet. This then delegates to Spring MVC Controllers that will provide the services
- The key difference between a traditional Spring MVC Controller and the Restful Web Service Controller is the way the HTTP response body is created
- While the traditional MVC Controller relies on the View technology, the Restful Web Service Controller simply returns the object and the object data is written directly to the HTTP response as a String/JSON/XML. Spring lets you return data directly from the Controller, without looking for a view, using the @ResponseBody annotation on a Controller method. Beginning with Version 4.0, this process is simplified even further with the introduction of the @RestController annotation

19920612 Copyright DGP Solutions 3-6

- Restful services use URIs to link to methods in Controllers. By applying the @ResponseBody annotation to the methods return type, the returned object is being written directly to the Response by way if an associated converter to supply a textural representation of the desired Resource

```
@GetMapping("/{id}")
public @ResponseBody Town get(@PathVariable("id") long
id) {
     return dao.getTown(id);
}
```

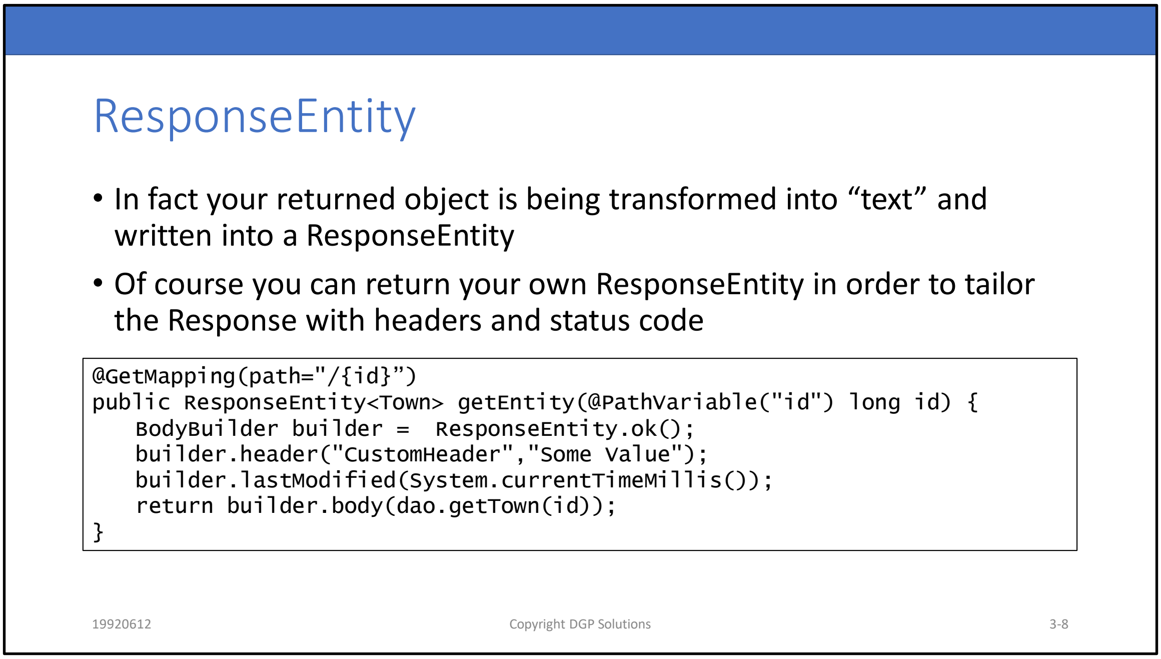


@RestController

- Spring 4 introduced a convenience annotation that is itself annotated with @Controller and @ResponseBody. So there is no need to use @ResponseBody if your Controller uses this annotation

```
@RestController
@RequestMapping("/town")
public class TownController {
@Autowired private TownDAO dao;

   @GetMapping(path="/{id}”)
   public Town get(@PathVariable("id") long id) {
        return dao.getTown(id);
   }
```

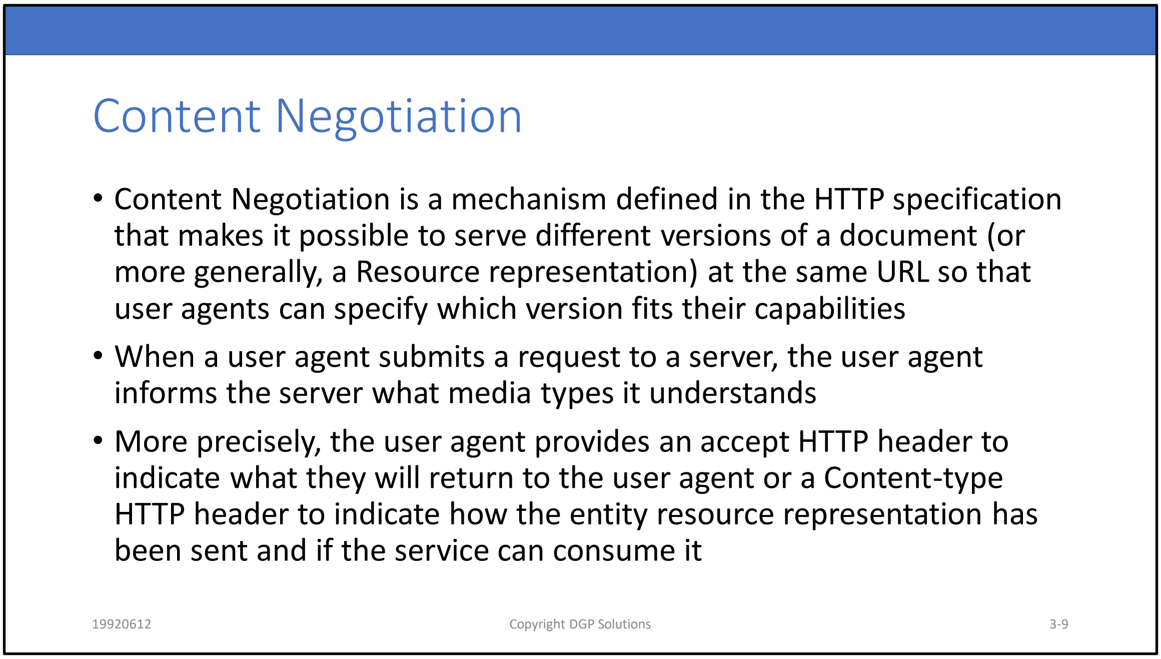
- @RestController indicates that every HTTP endpoint in this class will write its results directly into the HTTP response instead of using a view



ResponseEntity

- In fact your returned object is being transformed into “text” and written into a ResponseEntity
- Of course you can return your own ResponseEntity in order to tailor the Response with headers and status code

```
@GetMapping(path="/{id}”)
public ResponseEntity<Town> getEntity(@PathVariable("id") long id) {
   BodyBuilder builder =  ResponseEntity.ok();
   builder.header("CustomHeader","Some Value");
   builder.lastModified(System.currentTimeMillis());
   return builder.body(dao.getTown(id));
}
```

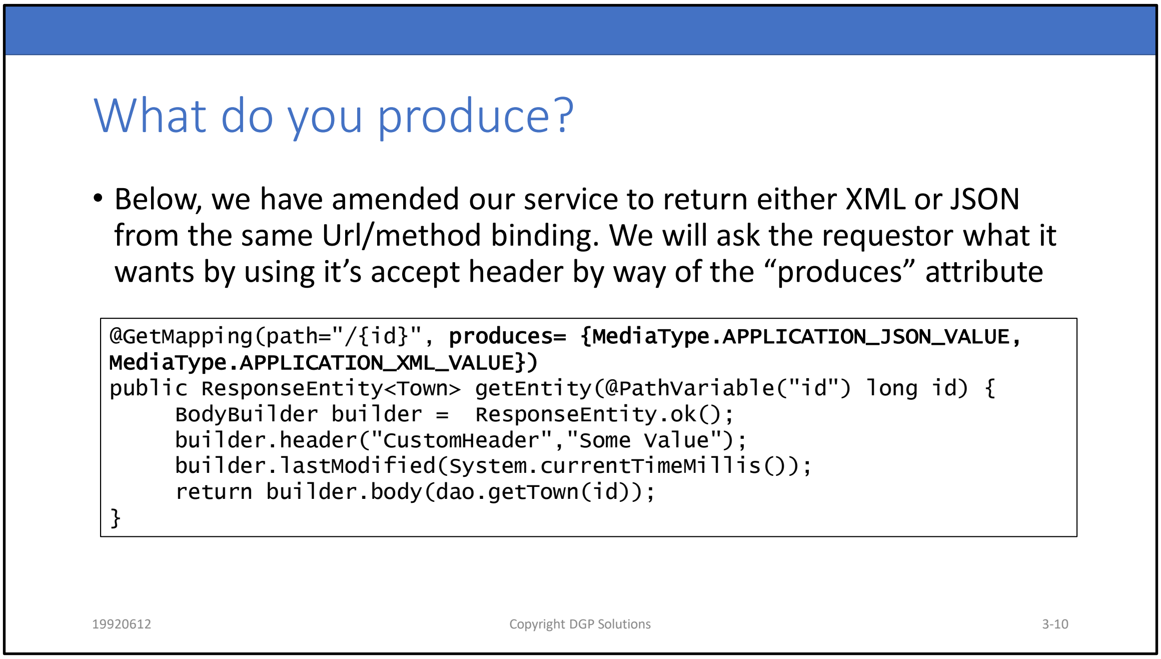


Content Negotiation

- Content Negotiation is a mechanism defined in the HTTP specification that makes it possible to serve different versions of a document (or more generally, a Resource representation) at the same URL so that user agents can specify which version fits their capabilities
- When a user agent submits a request to a server, the user agent informs the server what media types it understands
- More precisely, the user agent provides an accept HTTP header to indicate what they will return to the user agent or a Content-type HTTP header to indicate how the entity resource representation has been sent and if the service can consume it

19920612 Copyright DGP Solutions 3-9

- Roy Fielding called such negotiation “pre-emptive” if the server has to decide what representation to send the Resource back as or “reactive” if the user-agent has dictated what it wants



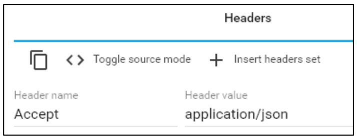
What do you produce?

- Below, we have amended our service to return either XML or JSON from the same Url/method binding. We will ask the requestor what it wants by using it’s accept header by way of the “produces” attribute

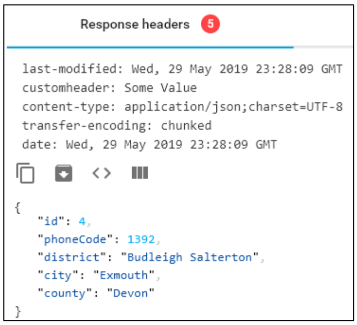
```
@GetMapping(path="/{id}", produces= {MediaType.APPLICATION_JSON_VALUE,
MediaType.APPLICATION_XML_VALUE})
public ResponseEntity<Town> getEntity(@PathVariable("id") long id) {
     BodyBuilder builder =  ResponseEntity.ok();
     builder.header("CustomHeader","Some Value");
     builder.lastModified(System.currentTimeMillis());
     return builder.body(dao.getTown(id));
}
```

Returning JSON

- Using Advanced REST client and the Chrome browser, we can set the Accept header for our request and ensure that our service returns an acceptable MIME type for our browser



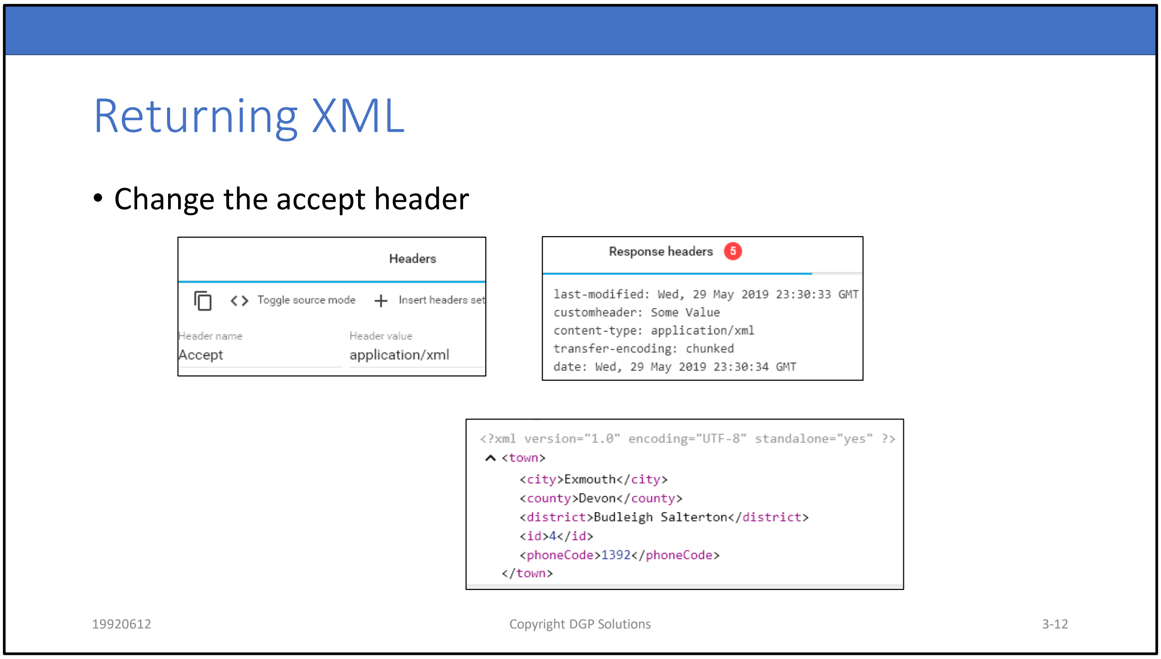
Header name	Header value
Accept	application/json



```
last-modified: Wed, 29 May 2019 23:28:09 GMT
customheader: Some Value
content-type: application/json;charset=UTF-8
transfer-encoding: chunked
date: Wed, 29 May 2019 23:28:09 GMT
```

```
{
    "id": 4,
    "phoneCode": 1392,
    "district": "Budleigh Salterton",
    "city": "Exmouth",
    "county": "Devon"
}
```

- Note the Content-Type response header confirming the MIME type



Returning XML

- Change the accept header

Headers

Toggle source mode Insert headers set

Header name: Accept

Header value: application/xml

Response headers 5

```
last-modified: Wed, 29 May 2019 23:30:33 GMT
customheader: Some Value
content-type: application/xml
transfer-encoding: chunked
date: Wed, 29 May 2019 23:30:34 GMT
```

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<town>
    <city>Exmouth</city>
    <county>Devon</county>
    <district>Budleigh Salterton</district>
    <id>4</id>
    <phoneCode>1392</phoneCode>
</town>
```

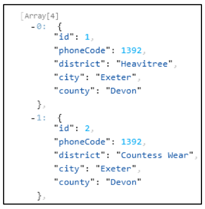
19920612 Copyright DGP Solutions 3-12

- Note your returned object must be annotated with @XmlRootElement for JAXB bindings

Returning Collections

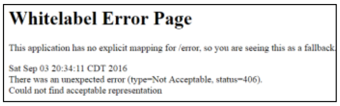
- When the return type is a Collection, our converters can marshal the Collection into a JSON array

```
@GetMapping(value="/location", produces=
MediaType.APPLICATION_JSON_VALUE)
public List<Town>  index(@RequestParam("code")
Optional<Integer> optional) {
    int code = optional.orElse(1392);
    return dao.getTowns().stream().filter(p-> p.getPhoneCode()
    == code).collect(Collectors.toList());
}
```



```
[Array[4]
  -0:  {
       "id": 1,
       "phoneCode": 1392,
       "district": "Heavitree",
       "city": "Exeter",
       "county": "Devon"
   },
  -1:  {
       "id": 2,
       "phoneCode": 1392,
       "district": "Countess Wear",
       "city": "Exeter",
       "county": "Devon"
   },
```

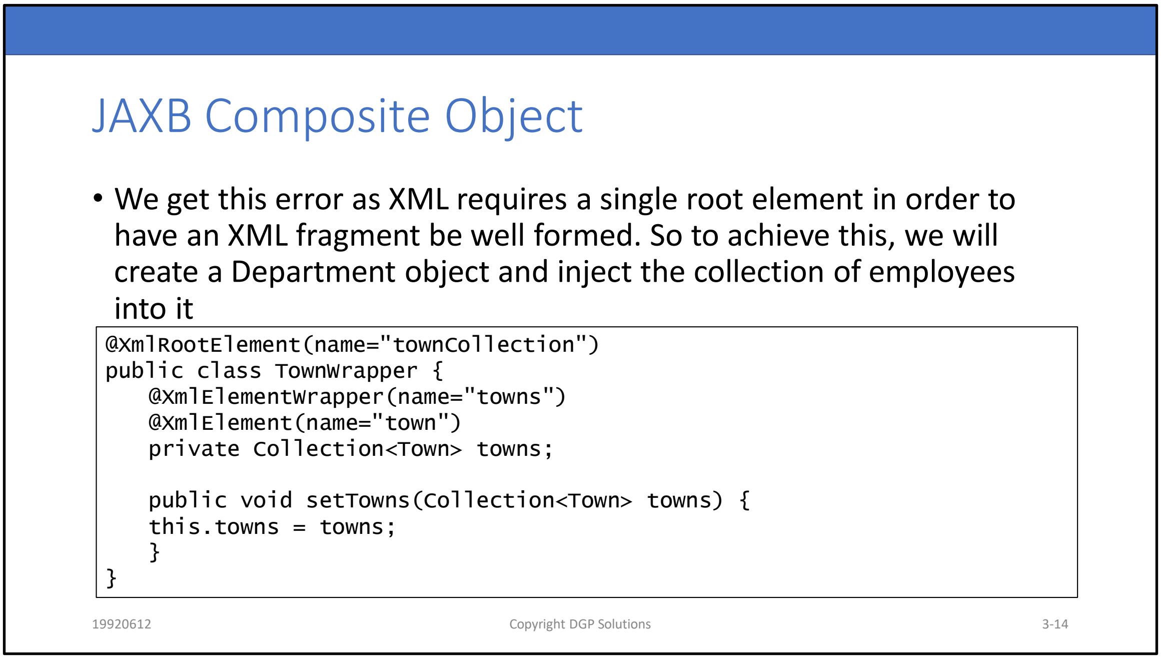
- However, changing this produces:
  - application/xml will cause problems



Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

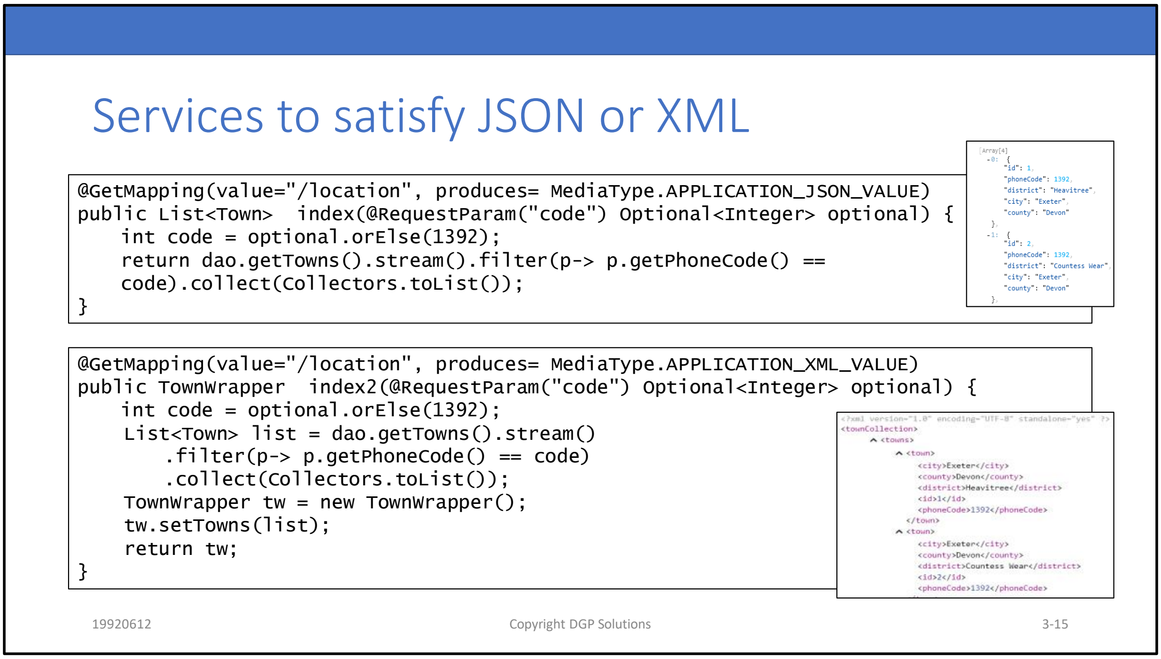
Sat Sep 03 20:34:11 CDT 2016
There was an unexpected error (type=Not Acceptable, status=406).
Could not find acceptable representation



JAXB Composite Object

- We get this error as XML requires a single root element in order to have an XML fragment be well formed. So to achieve this, we will create a Department object and inject the collection of employees into it

```
@XmlRootElement(name="townCollection")
public class TownWrapper {
   @XmlElementWrapper(name="towns")
   @XmlElement(name="town")
   private Collection<Town> towns;

   public void setTowns(Collection<Town> towns) {
   this.towns = towns;
   }
}
```



Services to satisfy JSON or XML

```
@GetMapping(value="/location", produces= MediaType.APPLICATION_JSON_VALUE)
public List<Town>  index(@RequestParam("code") Optional<Integer> optional) {
    int code = optional.orElse(1392);
    return dao.getTowns().stream().filter(p-> p.getPhoneCode() ==
    code).collect(Collectors.toList());
}
```

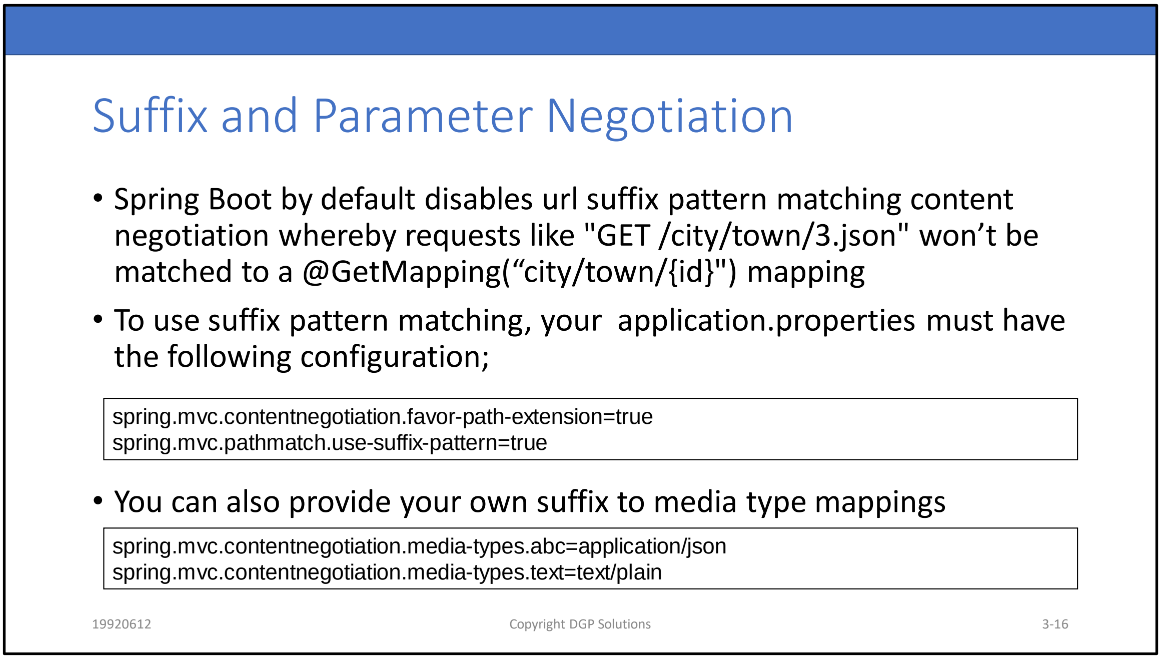
```
[Array[4]
  -0:  {
         "id": 1,
         "phoneCode": 1392,
         "district": "Heavitree",
         "city": "Exeter",
         "county": "Devon"
     },
  -1:  {
         "id": 2,
         "phoneCode": 1392,
         "district": "Countess Wear",
         "city": "Exeter",
         "county": "Devon"
     },
```

```
@GetMapping(value="/location", produces= MediaType.APPLICATION_XML_VALUE)
public TownWrapper  index2(@RequestParam("code") Optional<Integer> optional) {
    int code = optional.orElse(1392);
    List<Town> list = dao.getTowns().stream()
        .filter(p-> p.getPhoneCode() == code)
        .collect(Collectors.toList());
    TownWrapper tw = new TownWrapper();
    tw.setTowns(list);
    return tw;
}
```

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<townCollection>
    ^ <towns>
        ^ <town>
            <city>Exeter</city>
            <county>Devon</county>
            <district>Heavitree</district>
            <id>1</id>
            <phoneCode>1392</phoneCode>
          </town>
        ^ <town>
            <city>Exeter</city>
            <county>Devon</county>
            <district>Countess Wear</district>
            <id>2</id>
            <phoneCode>1392</phoneCode>
```

19920612 Copyright DGP Solutions 3-15

- Same url, same HTTP method but different produces



Suffix and Parameter Negotiation

- Spring Boot by default disables url suffix pattern matching content negotiation whereby requests like "GET /city/town/3.json" won’t be matched to a @GetMapping(“city/town/{id}") mapping
- To use suffix pattern matching, your application.properties must have the following configuration;

```
spring.mvc.contentnegotiation.favor-path-extension=true
spring.mvc.pathmatch.use-suffix-pattern=true
```

- You can also provide your own suffix to media type mappings

```
spring.mvc.contentnegotiation.media-types.abc=application/json
spring.mvc.contentnegotiation.media-types.text=text/plain
```

- Instead of using suffix matching, we can use a query parameter to ensure that requests like "GET /2?format=json will be matched to a @GetMapping(“/{id}") mapping
- In application.properties add an entry to include the means to access our services via a parameter called “format” (this is the default parameter name used for this negotiation technique)
- spring.mvc.contentnegotiation.favor-parameter=true

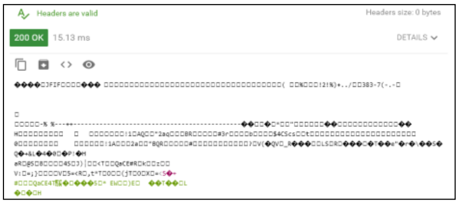
http://localhost:8082/city/town/2?format=json
http://localhost:8082/city/town/2?format=xml

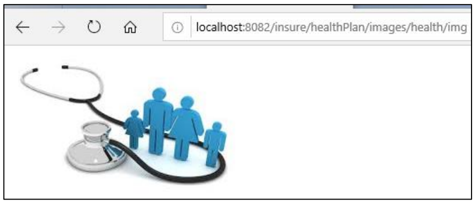
- You can change the parameter name via:
- spring.mvc.contentnegoation.parameter-name=foo

http://localhost:8082/city/town/2?foo=xml

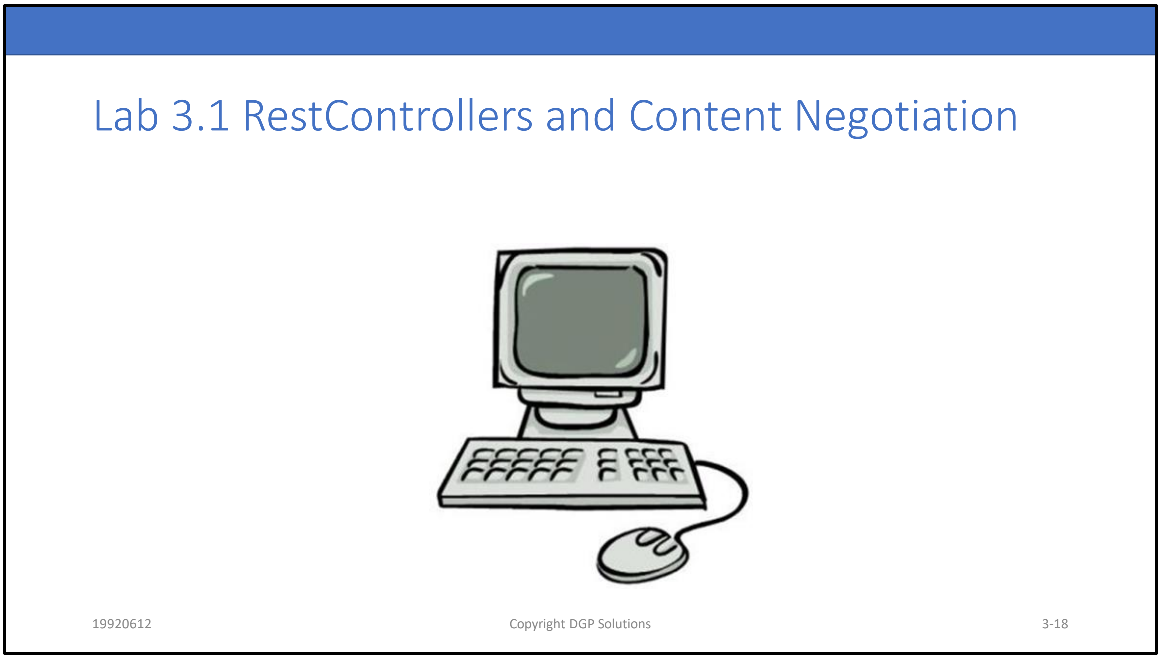
Returning Binary data

```
@GetMapping(path="/images/{code}/img", produces = MediaType.IMAGE_JPEG_VALUE)
public ResponseEntity getImage(@PathVariable("code") String code) throws
IOException {
    ClassPathResource r =
    new ClassPathResource("/static/images/" + code + ".jpeg");
    return
      ResponseEntity.ok().body(StreamUtils.copyToByteArray(r.getInputStream()
      ));
}
```

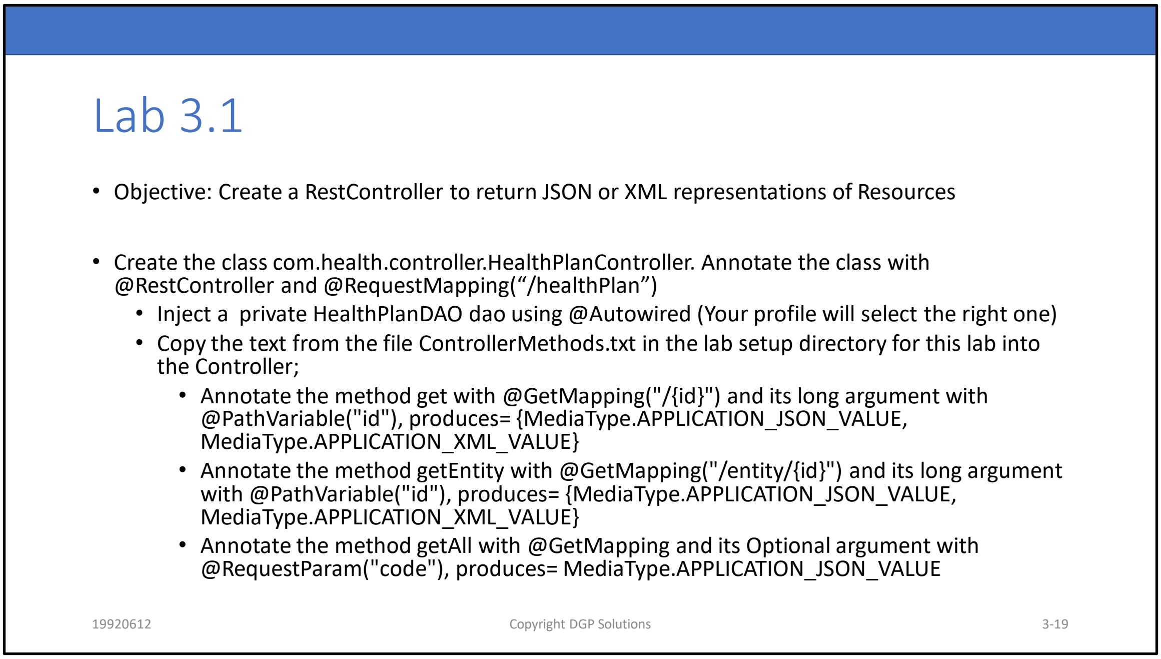




localhost:8082/insure/healthPlan/images/health/img



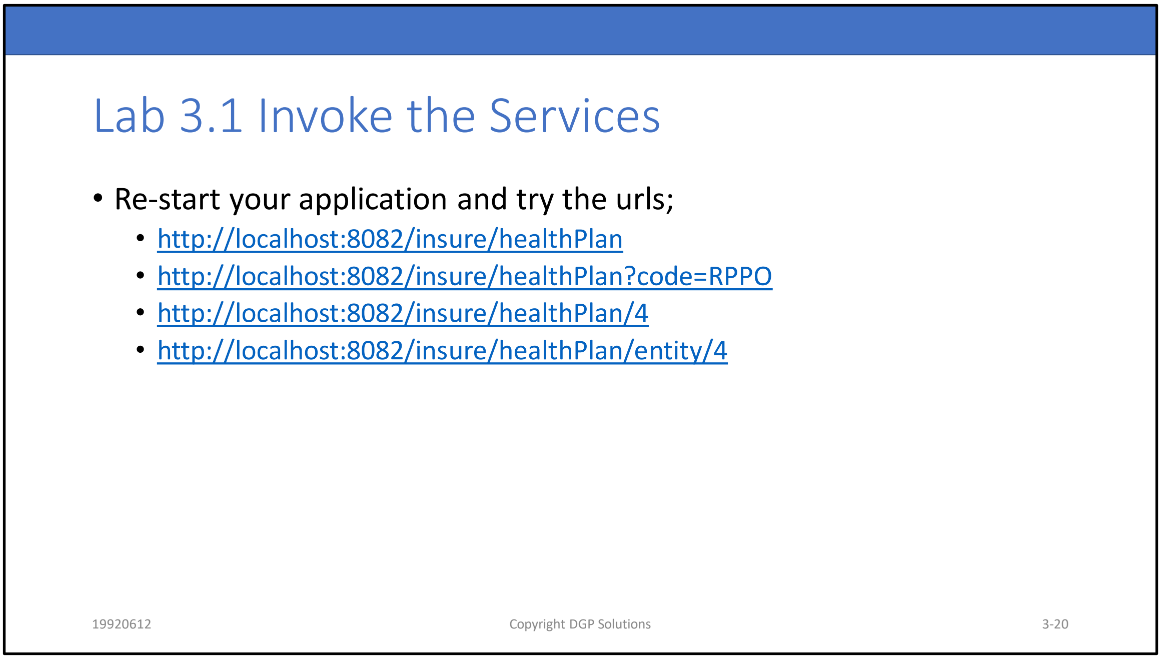
Lab 3.1 RestControllers and Content Negotiation



Lab 3.1

- Objective: Create a RestController to return JSON or XML representations of Resources

- Create the class com.health.controller.HealthPlanController. Annotate the class with @RestController and @RequestMapping(“/healthPlan”)
  - Inject a private HealthPlanDAO dao using @Autowired (Your profile will select the right one)
  - Copy the text from the file ControllerMethods.txt in the lab setup directory for this lab into the Controller;
    - Annotate the method get with @GetMapping("/{id}") and its long argument with @PathVariable("id"), produces= {MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE}
    - Annotate the method getEntity with @GetMapping("/entity/{id}") and its long argument with @PathVariable("id"), produces= {MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE}
    - Annotate the method getAll with @GetMapping and its Optional argument with @RequestParam("code"), produces= MediaType.APPLICATION_JSON_VALUE

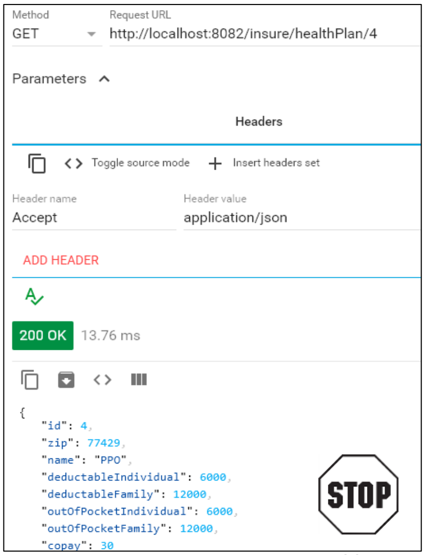


Lab 3.1 Invoke the Services

- Re-start your application and try the urls;
  - http://localhost:8082/insure/healthPlan
  - http://localhost:8082/insure/healthPlan?code=RPPO
  - http://localhost:8082/insure/healthPlan/4
  - http://localhost:8082/insure/healthPlan/entity/4

Lab 3.1 Content Negotiation

- Using a tool like Advanced Rest Client a Chrome “add on”, we can observe the headers being exchanged and negotiate what Response MediaType we want
- Add the following to the file application.properties to try suffix Content Negotiation
  - spring.mvc.contentnegotiation.favor-path-extension=true
  - spring.mvc.pathmatch.use-suffix-pattern=true
  - Try the urls
    - http://localhost:8082/insure/healthPlan/4.json
    - http://localhost:8082/insure/healthPlan/4.xml
    - http://localhost:8082/insure/healthPlan/4



Method: GET

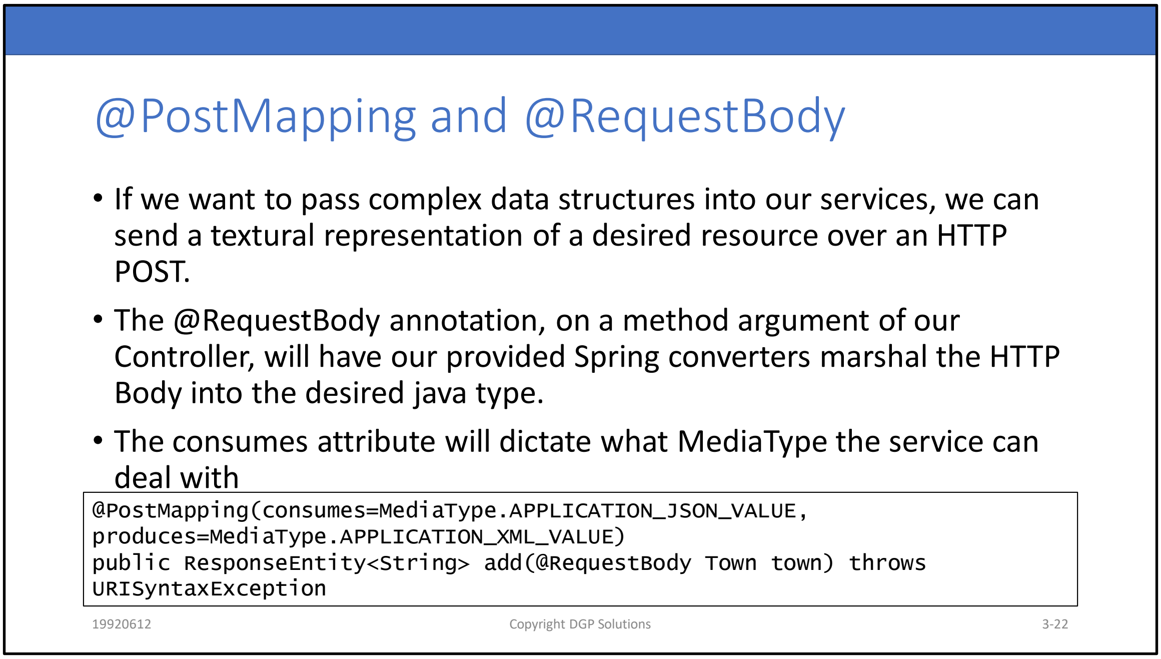
Request URL: http://localhost:8082/insure/healthPlan/4

Header name: Accept

Header value: application/json

200 OK 13.76 ms

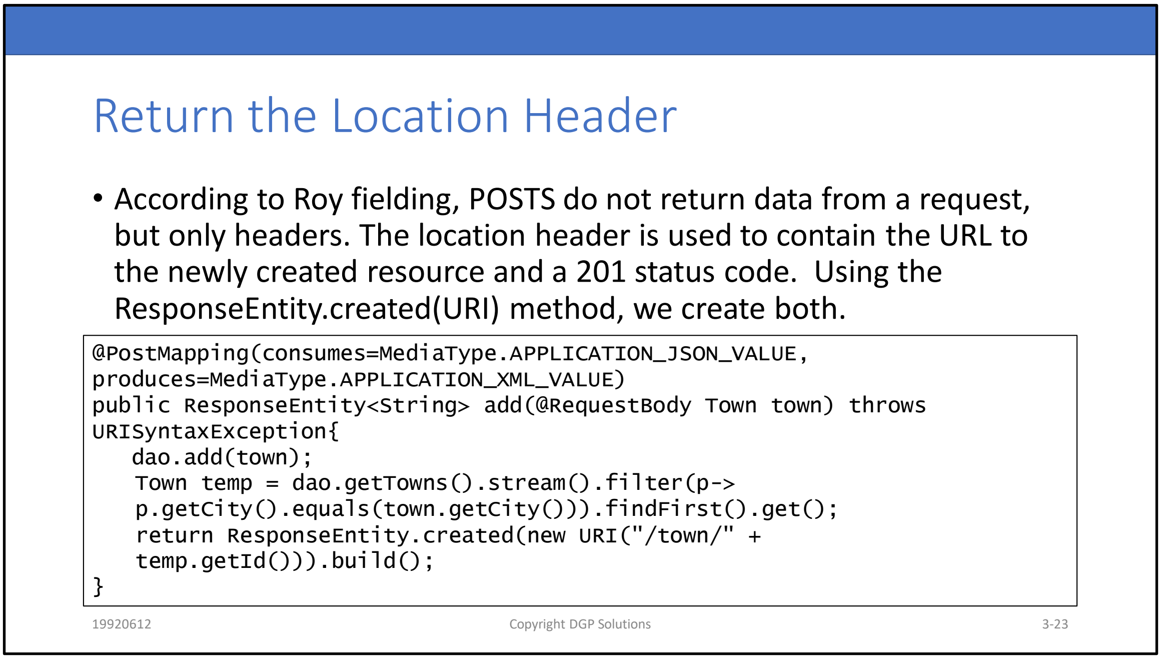
```
{
    "id": 4,
    "zip": 77429,
    "name": "PPO",
    "deductableIndividual": 6000,
    "deductableFamily": 12000,
    "outOfPocketIndividual": 6000,
    "outOfPocketFamily": 12000,
    "copay": 30
```



@PostMapping and @RequestBody

- If we want to pass complex data structures into our services, we can send a textural representation of a desired resource over an HTTP POST.
- The @RequestBody annotation, on a method argument of our Controller, will have our provided Spring converters marshal the HTTP Body into the desired java type.
- The consumes attribute will dictate what MediaType the service can deal with

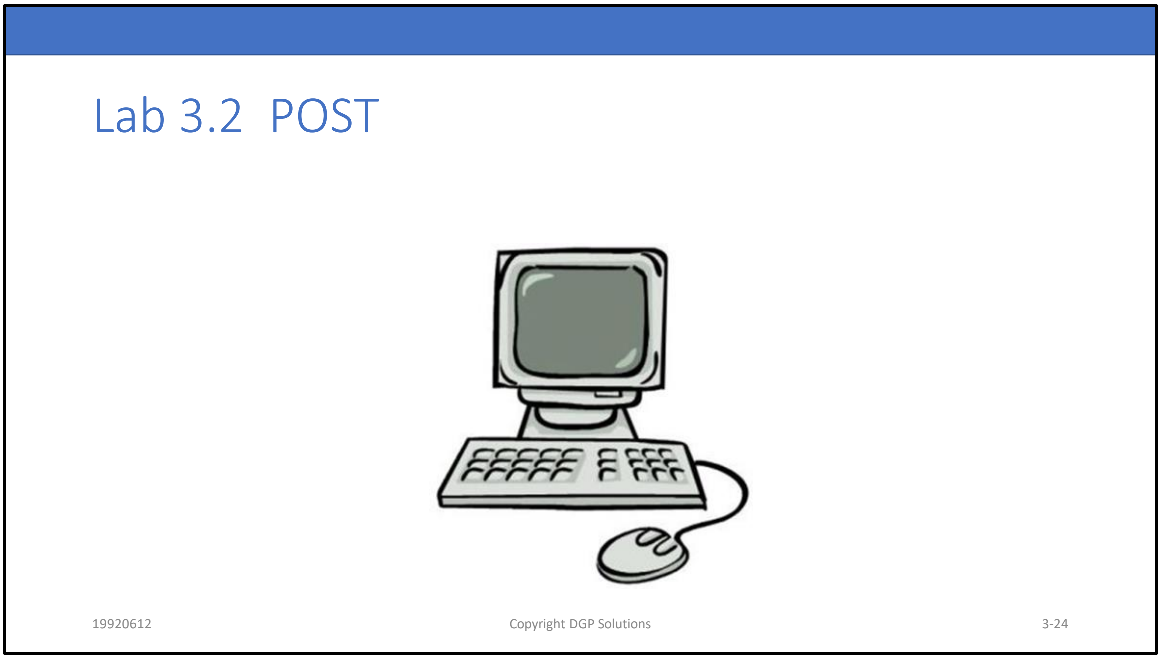
```
@PostMapping(consumes=MediaType.APPLICATION_JSON_VALUE,
produces=MediaType.APPLICATION_XML_VALUE)
public ResponseEntity<String> add(@RequestBody Town town) throws
URISyntaxException
```



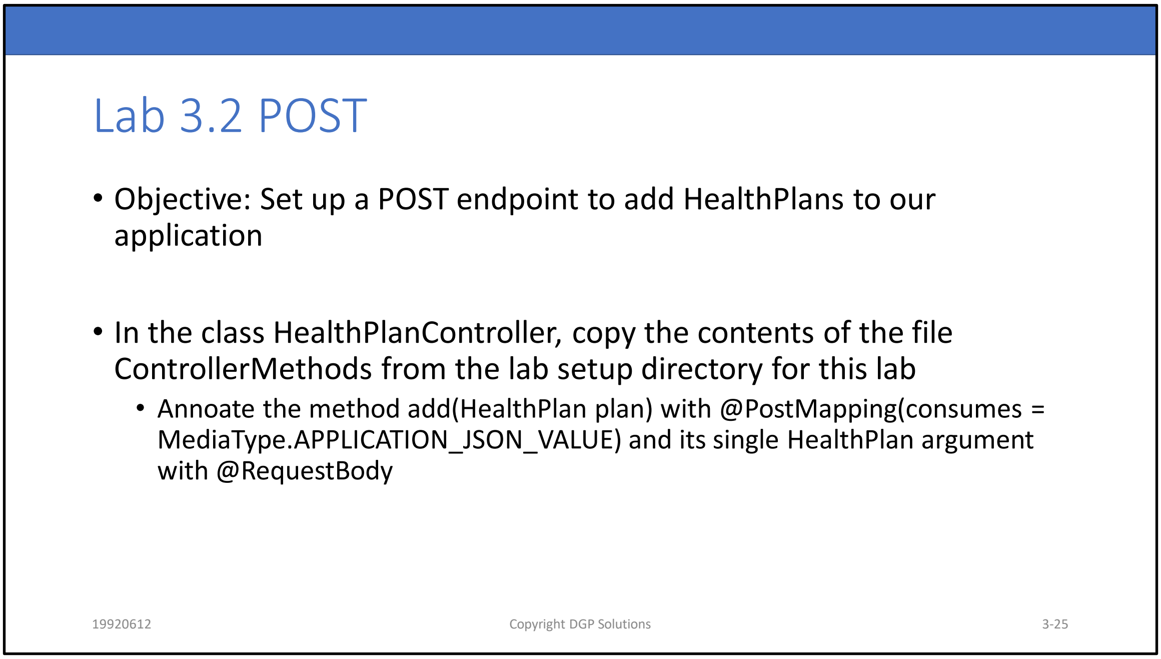
Return the Location Header

- According to Roy fielding, POSTS do not return data from a request, but only headers. The location header is used to contain the URL to the newly created resource and a 201 status code. Using the ResponseEntity.created(URI) method, we create both.

```
@PostMapping(consumes=MediaType.APPLICATION_JSON_VALUE,
produces=MediaType.APPLICATION_XML_VALUE)
public ResponseEntity<String> add(@RequestBody Town town) throws
URISyntaxException{
   dao.add(town);
   Town temp = dao.getTowns().stream().filter(p->
   p.getCity().equals(town.getCity())).findFirst().get();
   return ResponseEntity.created(new URI("/town/" +
   temp.getId())).build();
}
```



Lab 3.2 POST



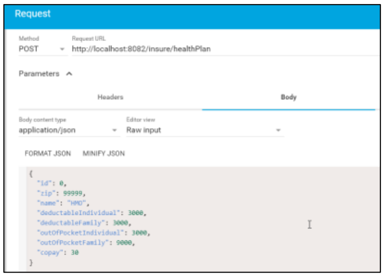
Lab 3.2 POST

- Objective: Set up a POST endpoint to add HealthPlans to our application

- In the class HealthPlanController, copy the contents of the file ControllerMethods from the lab setup directory for this lab
  - Annoate the method add(HealthPlan plan) with @PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE) and its single HealthPlan argument with @RequestBody

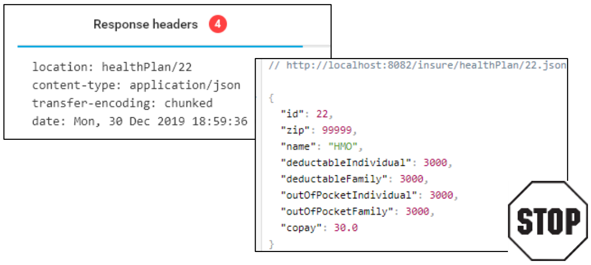
Lab 3.2 POST a Plan

- Using Advanced Rest Client we can POST a json representation of a HealthPlan, that you can cut and paste from the file json.text in the lab setup directory for this lab, into the body section of the request.
- Set the Content-type header to application/json and Submit



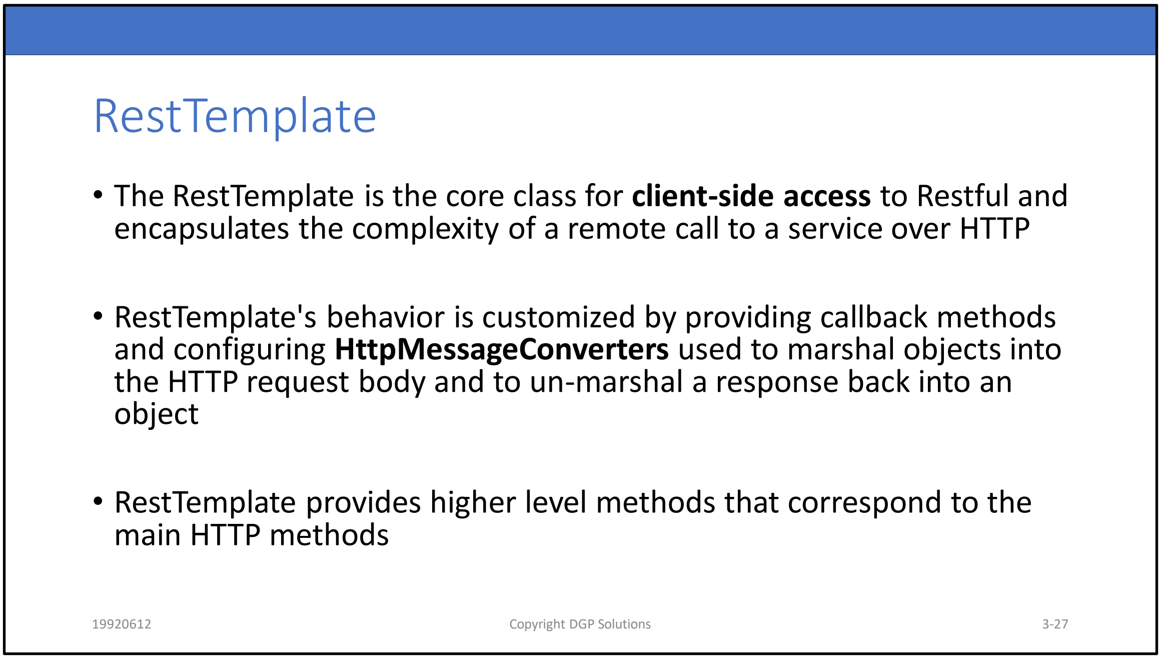
Method: POST
Request URL: http://localhost:8082/insure/healthPlan
Body content type: application/json
Editor view: Raw input

```
{
  "id": 0,
  "zip": 99999,
  "name": "HMO",
  "deductableIndividual": 3000,
  "deductableFamily": 3000,
  "outOfPocketIndividual": 3000,
  "outOfPocketFamily": 9000,
  "copay": 30
}
```



```
location: healthPlan/22
content-type: application/json
transfer-encoding: chunked
date: Mon, 30 Dec 2019 18:59:36
```

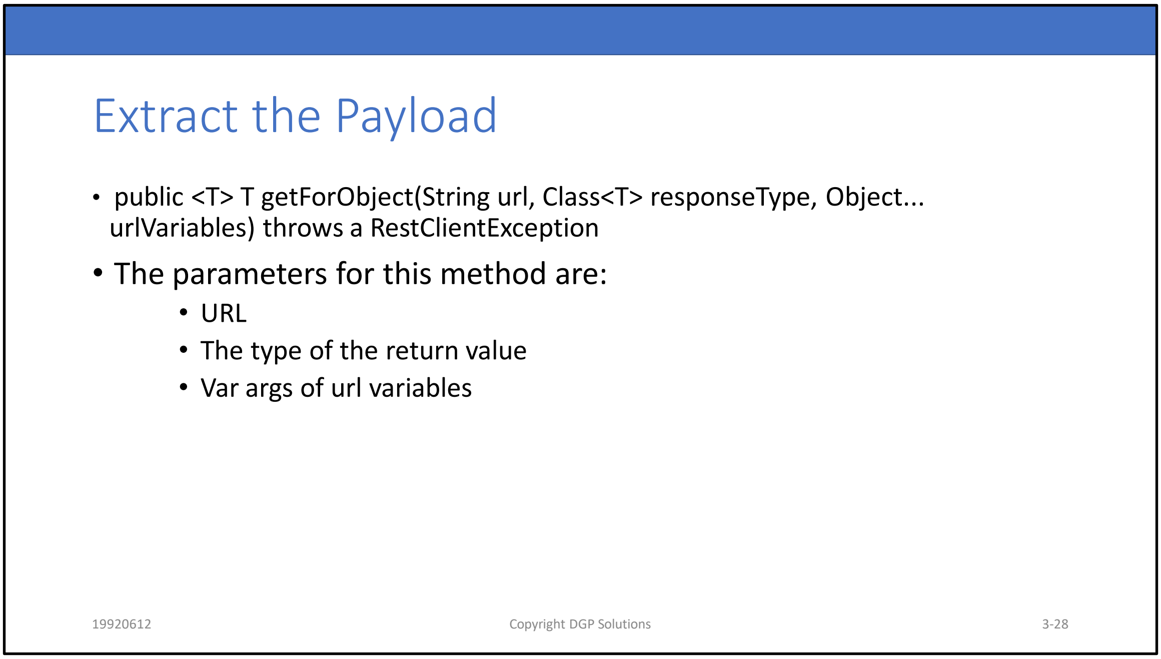
```
// http://localhost:8082/insure/healthPlan/22.json

{
  "id": 22,
  "zip": 99999,
  "name": "HMO",
  "deductableIndividual": 3000,
  "deductableFamily": 3000,
  "outOfPocketIndividual": 3000,
  "outOfPocketFamily": 3000,
  "copay": 30.0
}
```



RestTemplate

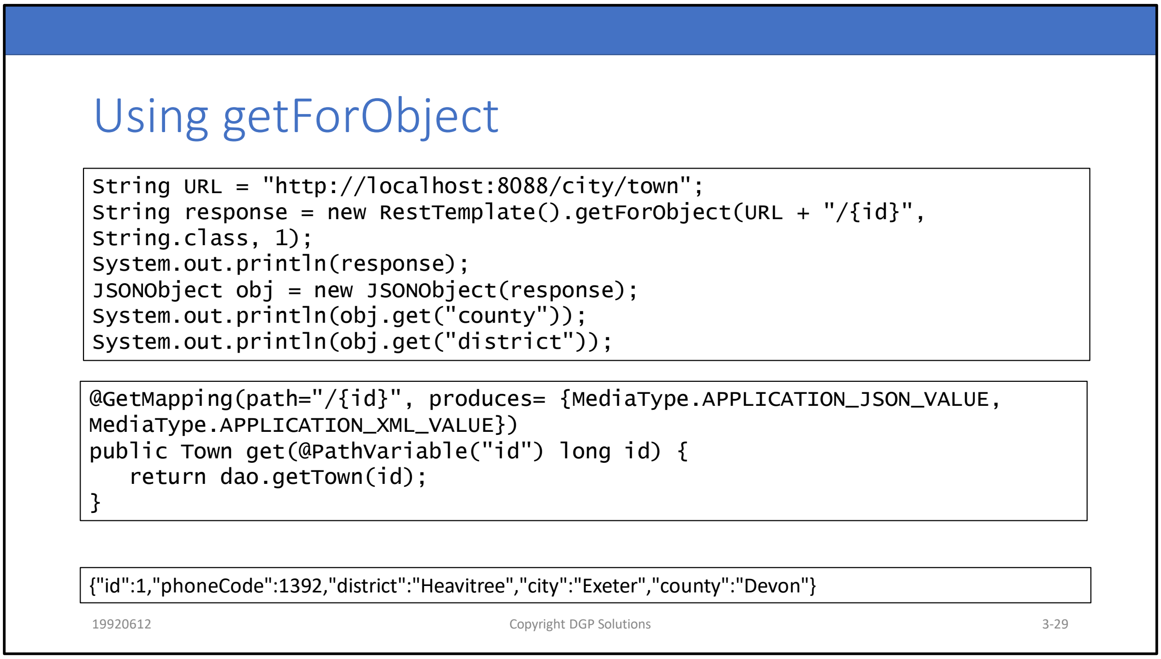
- The RestTemplate is the core class for **client-side access** to Restful and encapsulates the complexity of a remote call to a service over HTTP
- RestTemplate's behavior is customized by providing callback methods and configuring **HttpMessageConverters** used to marshal objects into the HTTP request body and to un-marshal a response back into an object
- RestTemplate provides higher level methods that correspond to the main HTTP methods

DELETE	delete(java.lang.String, java.lang.Object...)
GET	getForObject(java.lang.String, java.lang.Class<T>, java.lang.Object...)
	getForEntity (java.lang.String, java.lang.Class<T>, java.lang.Object...)
HEAD	headForHeaders(java.lang.String, java.lang.Object...)
OPTIONS	optionsForAllow(java.lang.String, java.lang.Object...)
POST	postForLocation(java.lang.String, java.lang.Object, java.lang.Object...)
	postForObject(java.lang.String, java.lang.Object, java.lang.Class<T>, java.lang.Object...)



Extract the Payload

- public <T> T getForObject(String url, Class<T> responseType, Object... urlVariables) throws a RestClientException
- The parameters for this method are:
  - URL
  - The type of the return value
  - Var args of url variables

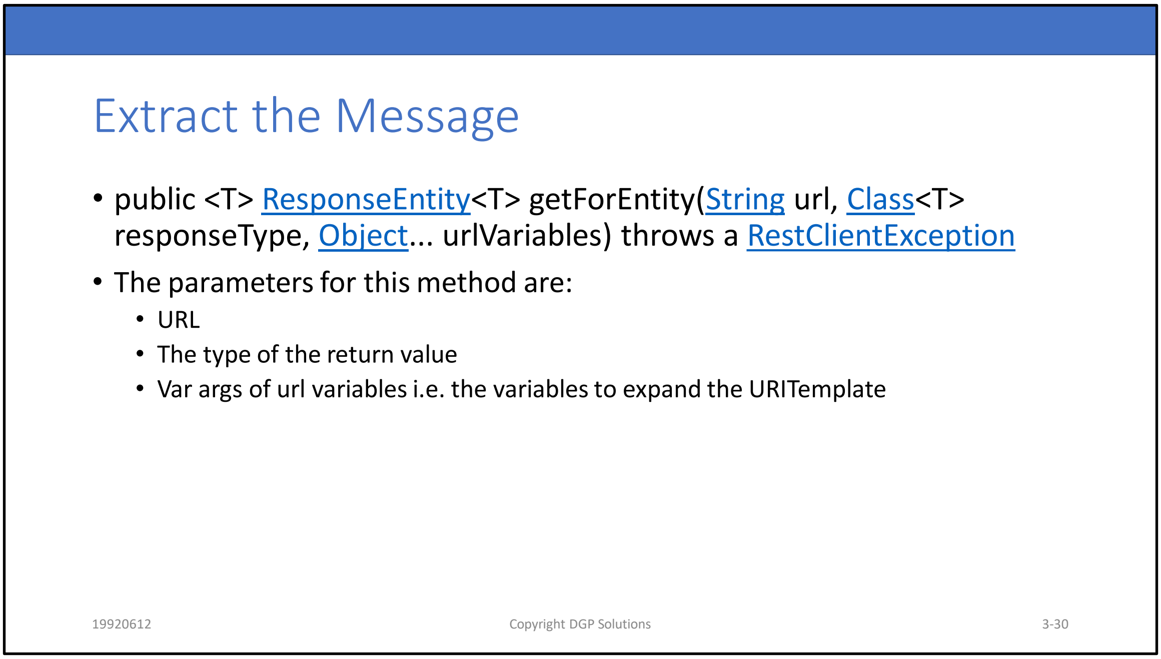


Using getForObject

```
String URL = "http://localhost:8088/city/town";
String response = new RestTemplate().getForObject(URL + "/{id}",
String.class, 1);
System.out.println(response);
JSONObject obj = new JSONObject(response);
System.out.println(obj.get("county"));
System.out.println(obj.get("district"));
```

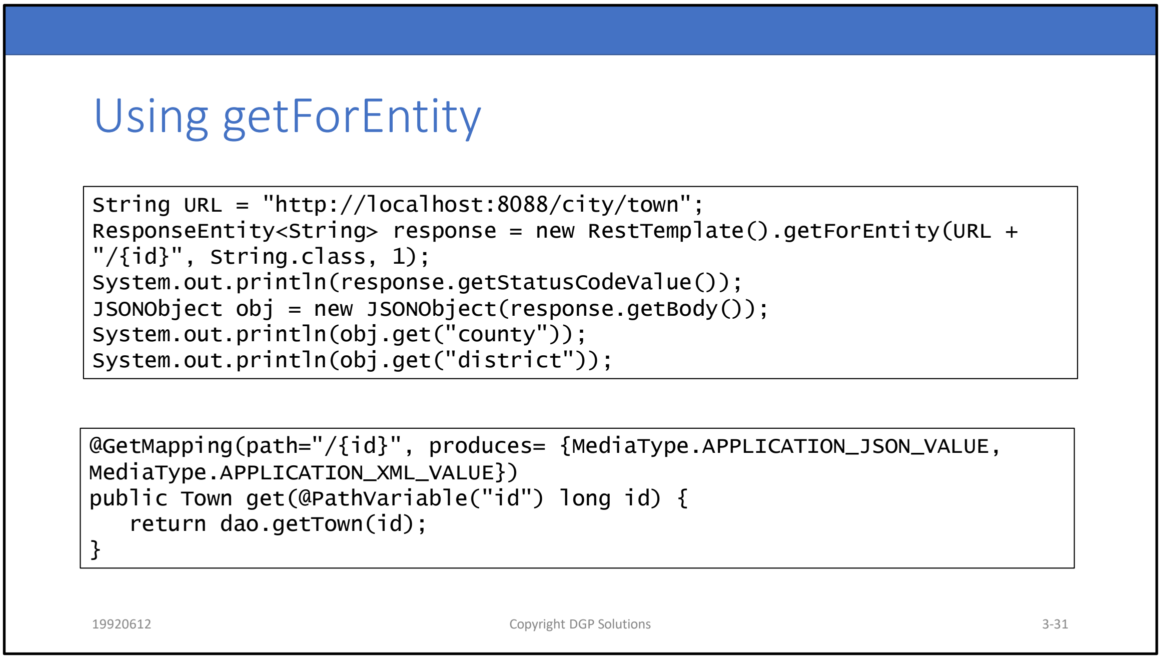
```
@GetMapping(path="/{id}", produces= {MediaType.APPLICATION_JSON_VALUE,
MediaType.APPLICATION_XML_VALUE})
public Town get(@PathVariable("id") long id) {
   return dao.getTown(id);
}
```

```
{"id":1,"phoneCode":1392,"district":"Heavitree","city":"Exeter","county":"Devon"}
```



Extract the Message

- public <T> ResponseEntity<T> getForEntity(String url, Class<T> responseType, Object... urlVariables) throws a RestClientException
- The parameters for this method are:
  - URL
  - The type of the return value
  - Var args of url variables i.e. the variables to expand the URITemplate

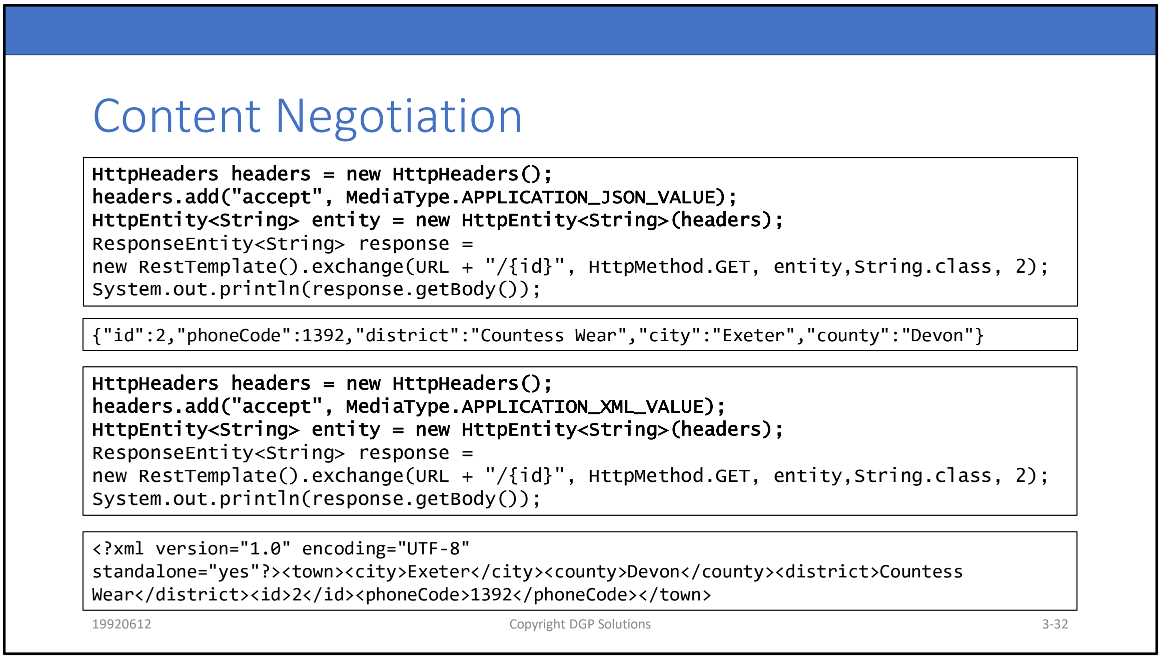


Using getForEntity

```
String URL = "http://localhost:8088/city/town";
ResponseEntity<String> response = new RestTemplate().getForEntity(URL +
"/{id}", String.class, 1);
System.out.println(response.getStatusCodeValue());
JSONObject obj = new JSONObject(response.getBody());
System.out.println(obj.get("county"));
System.out.println(obj.get("district"));
```

```
@GetMapping(path="/{id}", produces= {MediaType.APPLICATION_JSON_VALUE,
MediaType.APPLICATION_XML_VALUE})
public Town get(@PathVariable("id") long id) {
   return dao.getTown(id);
}
```

n



Content Negotiation

```
HttpHeaders headers = new HttpHeaders();
headers.add("accept", MediaType.APPLICATION_JSON_VALUE);
HttpEntity<String> entity = new HttpEntity<String>(headers);
ResponseEntity<String> response =
new RestTemplate().exchange(URL + "/{id}", HttpMethod.GET, entity,String.class, 2);
System.out.println(response.getBody());
```

```
{"id":2,"phoneCode":1392,"district":"Countess Wear","city":"Exeter","county":"Devon"}
```

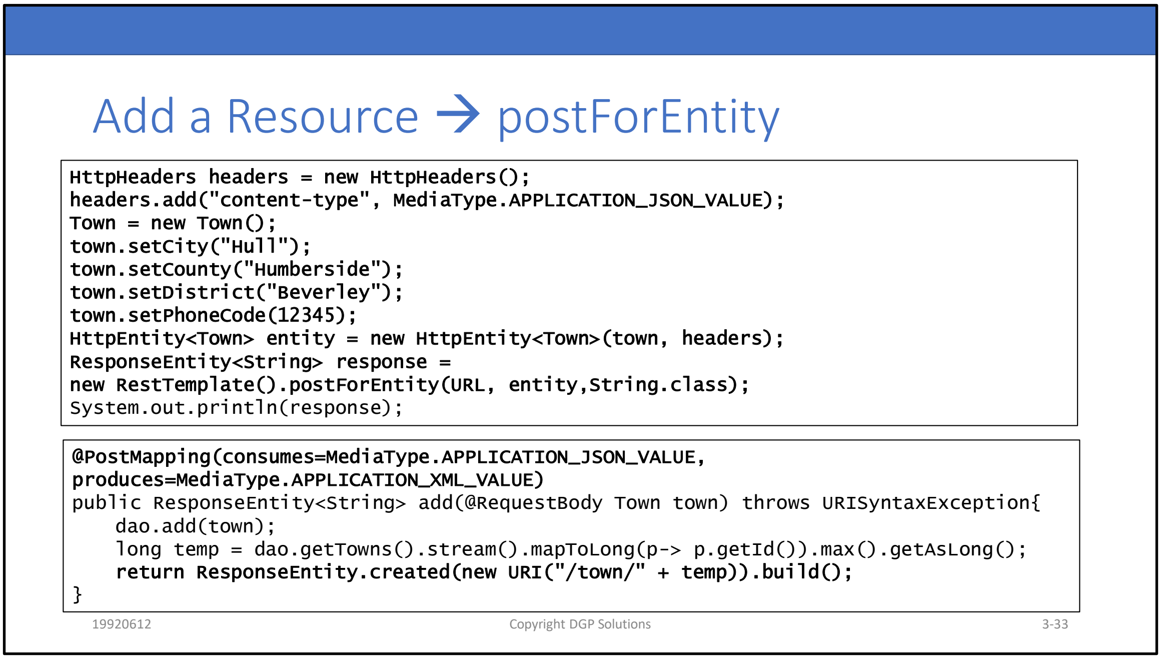
```
HttpHeaders headers = new HttpHeaders();
headers.add("accept", MediaType.APPLICATION_XML_VALUE);
HttpEntity<String> entity = new HttpEntity<String>(headers);
ResponseEntity<String> response =
new RestTemplate().exchange(URL + "/{id}", HttpMethod.GET, entity,String.class, 2);
System.out.println(response.getBody());
```

```
<?xml version="1.0" encoding="UTF-8"
standalone="yes"?><town><city>Exeter</city><county>Devon</county><district>Countess
Wear</district><id>2</id><phoneCode>1392</phoneCode></town>
```

19920612 Copyright DGP Solutions 3-32

- Converters work on the Template, unmarsalling into Objects

```
HttpHeaders headers = new HttpHeaders();
headers.add("accept", MediaType.APPLICATION_JSON_VALUE);
HttpEntity<String> entity = new HttpEntity<String>(headers);
ResponseEntity<Town> response =
new RestTemplate().exchange(URL + "/{id}", HttpMethod.GET, entity,Town.class, 2);
System.out.println(response.getBody());
```



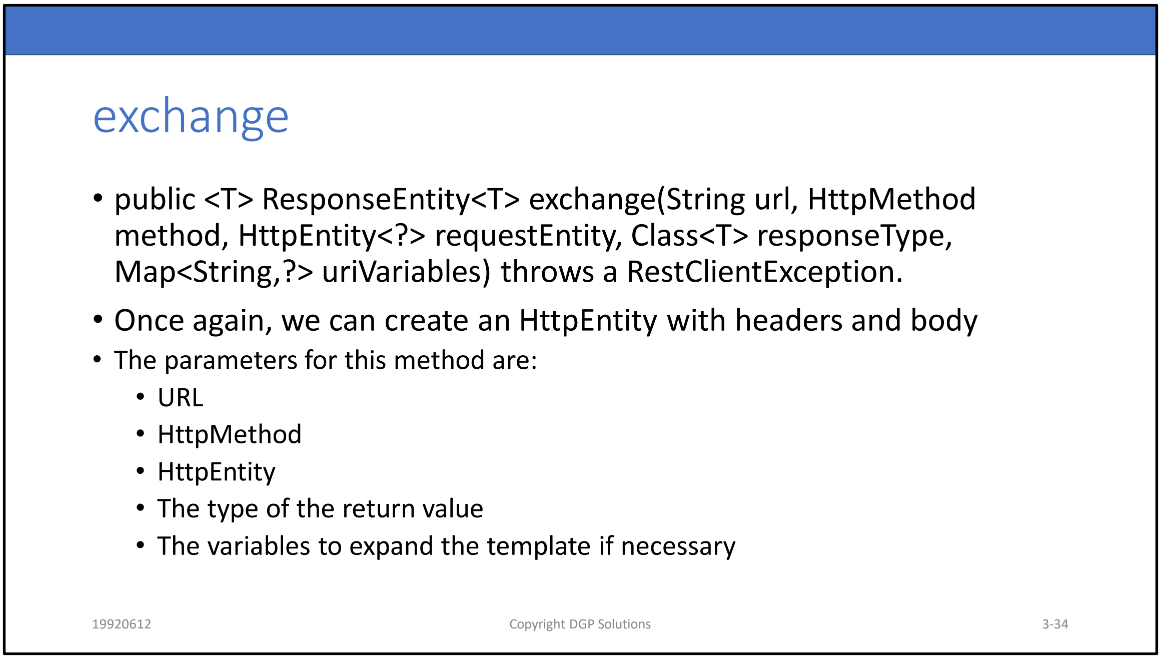
Add a Resource → postForEntity

```
HttpHeaders headers = new HttpHeaders();
headers.add("content-type", MediaType.APPLICATION_JSON_VALUE);
Town = new Town();
town.setCity("Hull");
town.setCounty("Humberside");
town.setDistrict("Beverley");
town.setPhoneCode(12345);
HttpEntity<Town> entity = new HttpEntity<Town>(town, headers);
ResponseEntity<String> response =
new RestTemplate().postForEntity(URL, entity,String.class);
System.out.println(response);
```

```
@PostMapping(consumes=MediaType.APPLICATION_JSON_VALUE,
produces=MediaType.APPLICATION_XML_VALUE)
public ResponseEntity<String> add(@RequestBody Town town) throws URISyntaxException{
    dao.add(town);
    long temp = dao.getTowns().stream().mapToLong(p-> p.getId()).max().getAsLong();
    return ResponseEntity.created(new URI("/town/" + temp)).build();
}
```

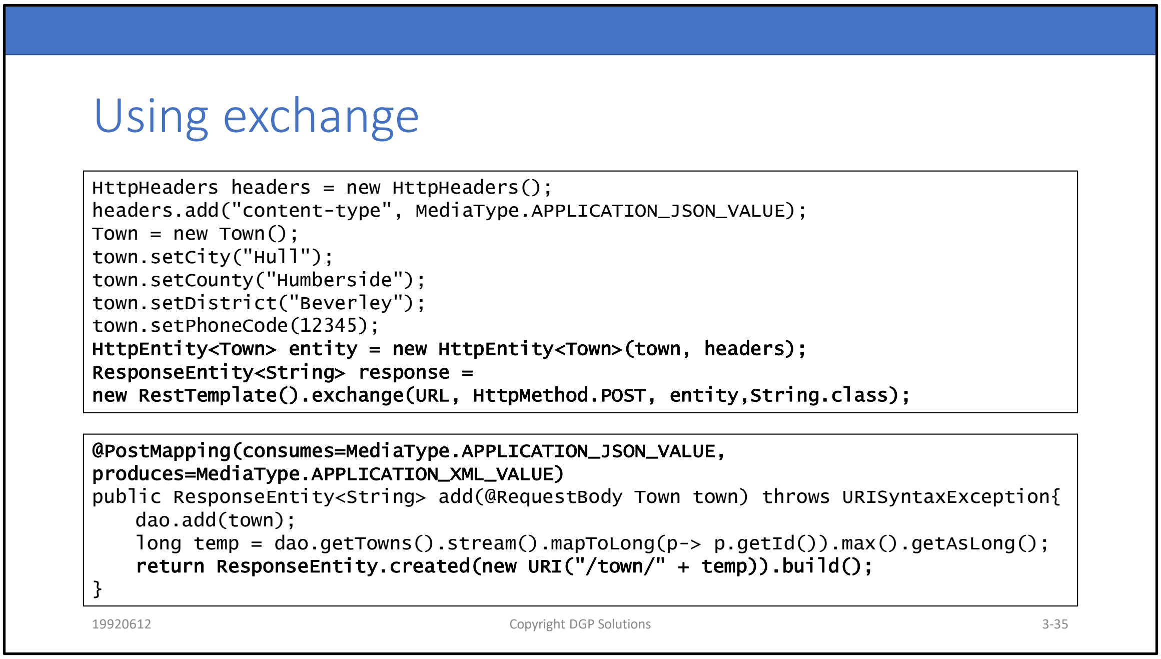
19920612 Copyright DGP Solutions 3-33

- For a POST we set the content-type header, in order to “match” against a service with the corresponding consumes attribute that matches MediaType



exchange

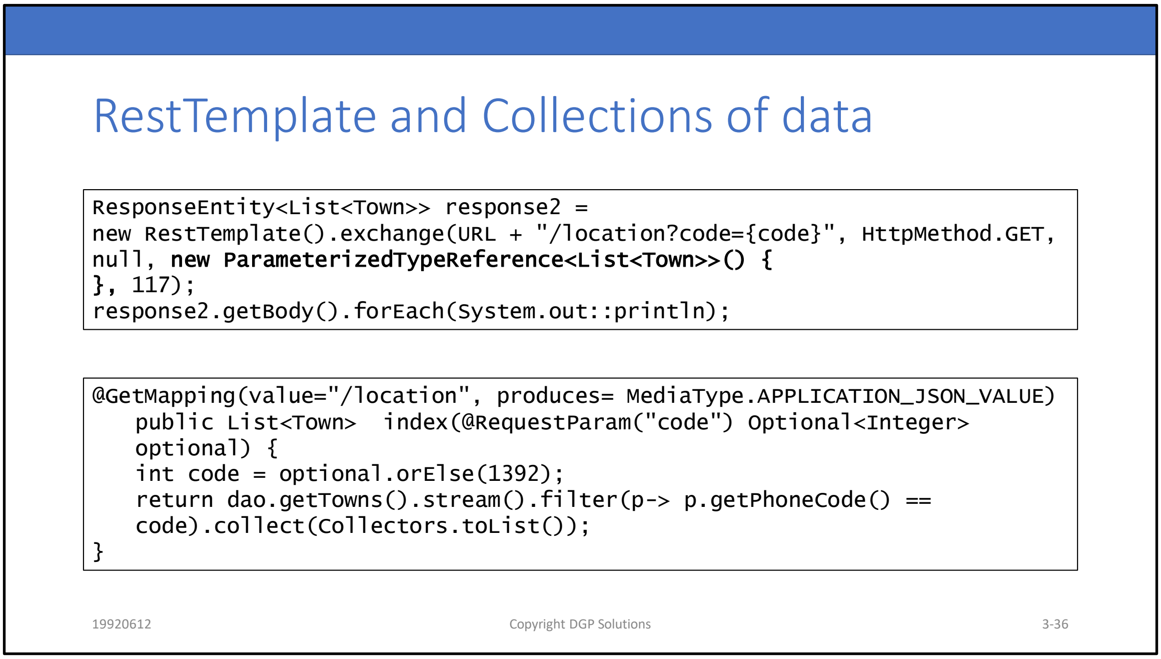
- public <T> ResponseEntity<T> exchange(String url, HttpMethod method, HttpEntity<?> requestEntity, Class<T> responseType, Map<String,?> uriVariables) throws a RestClientException.
- Once again, we can create an HttpEntity with headers and body
- The parameters for this method are:
  - URL
  - HttpMethod
  - HttpEntity
  - The type of the return value
  - The variables to expand the template if necessary



Using exchange

```
HttpHeaders headers = new HttpHeaders();
headers.add("content-type", MediaType.APPLICATION_JSON_VALUE);
Town = new Town();
town.setCity("Hull");
town.setCounty("Humberside");
town.setDistrict("Beverley");
town.setPhoneCode(12345);
HttpEntity<Town> entity = new HttpEntity<Town>(town, headers);
ResponseEntity<String> response =
new RestTemplate().exchange(URL, HttpMethod.POST, entity,String.class);
```

```
@PostMapping(consumes=MediaType.APPLICATION_JSON_VALUE,
produces=MediaType.APPLICATION_XML_VALUE)
public ResponseEntity<String> add(@RequestBody Town town) throws URISyntaxException{
    dao.add(town);
    long temp = dao.getTowns().stream().mapToLong(p-> p.getId()).max().getAsLong();
    return ResponseEntity.created(new URI("/town/" + temp)).build();
}
```

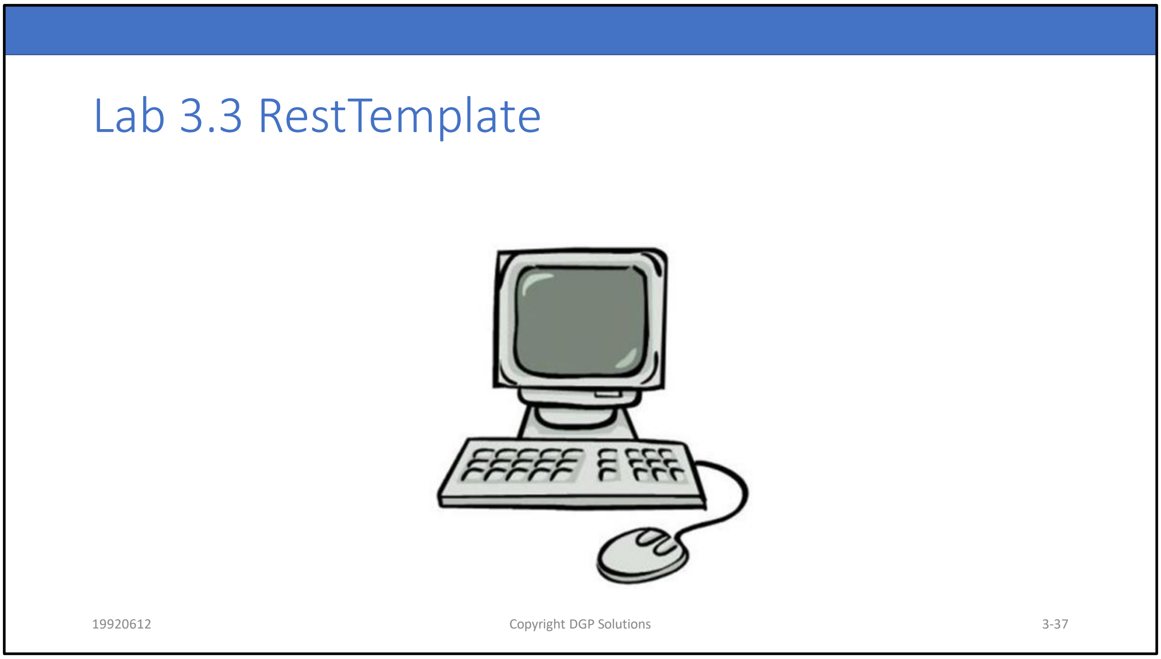


RestTemplate and Collections of data

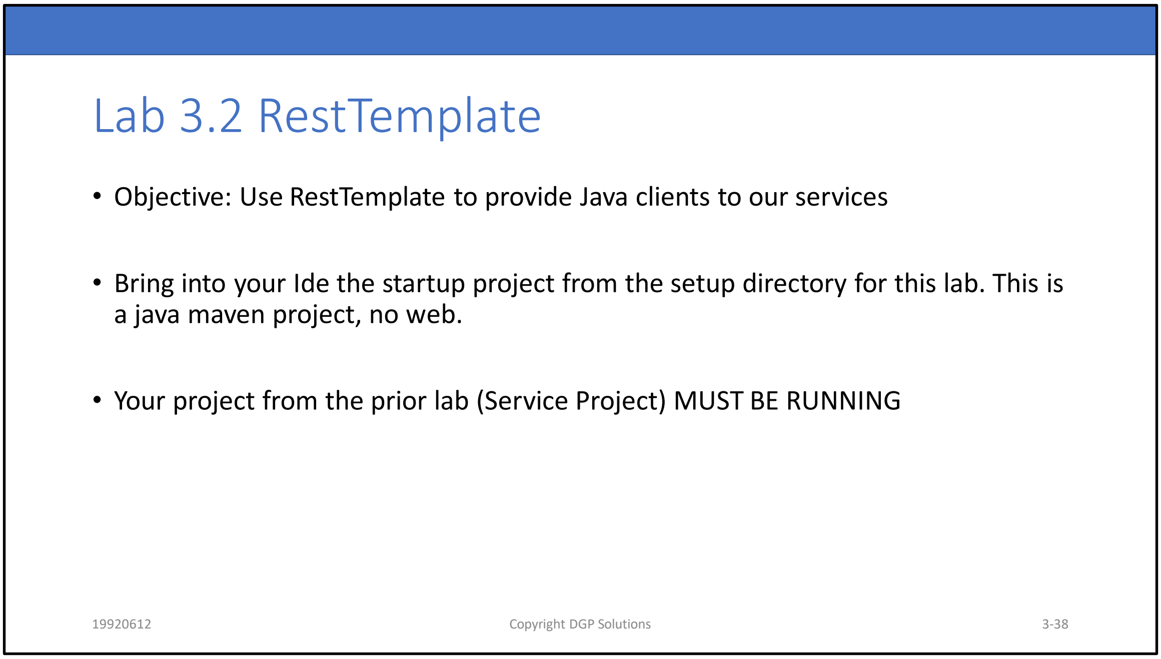
```
ResponseEntity<List<Town>> response2 =
new RestTemplate().exchange(URL + "/location?code={code}", HttpMethod.GET,
null, new ParameterizedTypeReference<List<Town>>() {
}, 117);
response2.getBody().forEach(System.out::println);
```

```
@GetMapping(value="/location", produces= MediaType.APPLICATION_JSON_VALUE)
   public List<Town>  index(@RequestParam("code") Optional<Integer>
   optional) {
   int code = optional.orElse(1392);
   return dao.getTowns().stream().filter(p-> p.getPhoneCode() ==
   code).collect(Collectors.toList());
}
```

- `ParameterizedTypeReference;` Returns the Type representing the direct superclass of the entity (class, interface, primitive type or void) represented by this Class.
- If the superclass is a parameterized type, **the Type object returned must accurately reflect the actual type parameters used in the source code.**



Lab 3.3 RestTemplate

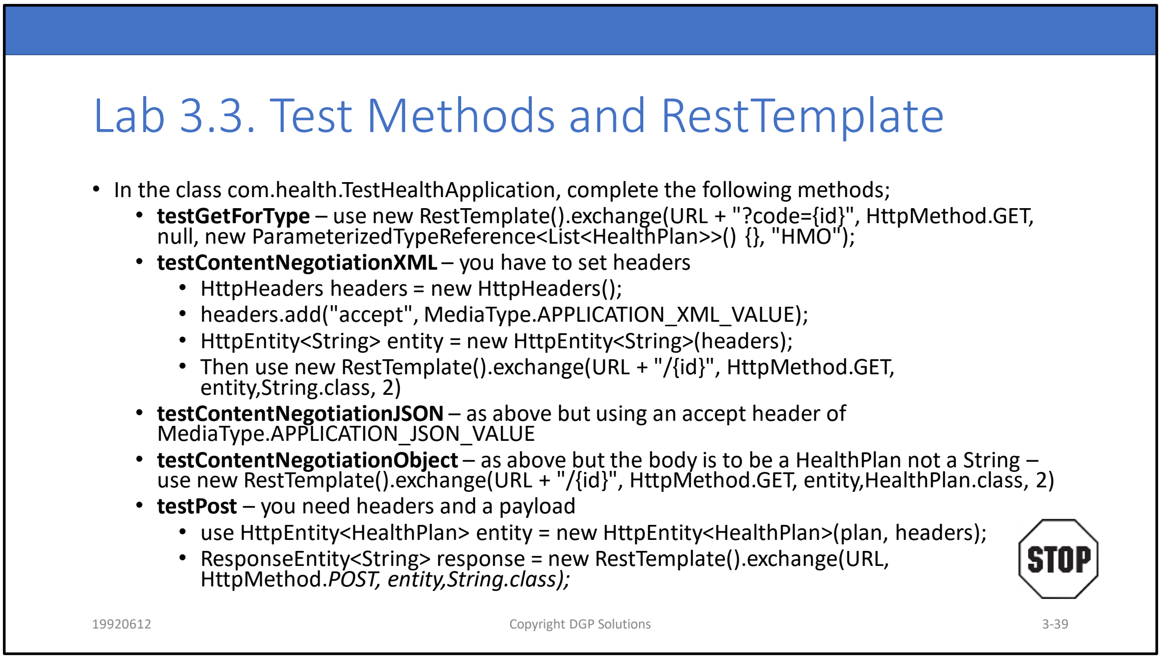


Lab 3.2 RestTemplate

- Objective: Use RestTemplate to provide Java clients to our services

- Bring into your Ide the startup project from the setup directory for this lab. This is a java maven project, no web.

- Your project from the prior lab (Service Project) MUST BE RUNNING



Lab 3.3. Test Methods and RestTemplate

- In the class com.health.TestHealthApplication, complete the following methods;
  - **testGetForType** – use new RestTemplate().exchange(URL + "?code={id}", HttpMethod.GET, null, new ParameterizedTypeReference<List<HealthPlan>>() {}, "HMO");
  - **testContentNegotiationXML** – you have to set headers
    - HttpHeaders headers = new HttpHeaders();
    - headers.add("accept", MediaType.APPLICATION_XML_VALUE);
    - HttpEntity<String> entity = new HttpEntity<String>(headers);
    - Then use new RestTemplate().exchange(URL + "/{id}", HttpMethod.GET, entity,String.class, 2)
  - **testContentNegotiationJSON** – as above but using an accept header of MediaType.APPLICATION_JSON_VALUE
  - **testContentNegotiationObject** – as above but the body is to be a HealthPlan not a String – use new RestTemplate().exchange(URL + "/{id}", HttpMethod.GET, entity,HealthPlan.class, 2)
  - **testPost** – you need headers and a payload
    - use HttpEntity<HealthPlan> entity = new HttpEntity<HealthPlan>(plan, headers);
    - ResponseEntity<String> response = new RestTemplate().exchange(URL, HttpMethod.*POST, entity,String.class);*

STOP