MTAT.03.229
Enterprise System Integration

Lecture 5: REST Adapter (CRUD)

Marlon Dumas
University of Tartu
Overview of last week
Explicit integration layer

An explicit integration layer allows us to:

- Build multiple independent presentation components for the same application, e.g. one for Web, one for mobile, etc.
- Support programmatic application to application interactions (without involving presentation)
- Facilitate testing of the application logic in networked environments

REST is a standards-based architectural style to realize an integration layer

- Allows intermediaries—proxies, gateways, and firewalls—to be introduced without changing the interfaces between components, allowing them to improve performance via large-scale, shared caching
This week

Buildit

SalesRestController

PODTO’

PlantDTO’

SalesService

SalesController

PODTO

PlantDTO

OrderRepository

PurchaseOrder

PlantRepository

Plant

OrdersRepository

Web UI adapter

REST adapter

JPA adapter
REST

• REST stands for **Representational State Transfer**
  ◦ Introduced/defined in 2000 by Roy Fielding in his Doctoral Dissertation
  ◦ BTW, Fielding is one of the principal authors of HTTP specification versions 1.0 and 1.1

• REST is an architecture style for designing networked applications
  ◦ In many ways, the **World Wide Web** itself, can be viewed as a REST-based architecture

• REST is not an standard nor a product
  ➢ In contrast, SOAP, WSDL and a plethora of WS-* standards provide a foundation for SOA on the Web
REST: building blocks

Interaction model
Traversal of hyperlinks

Interaction protocol
Methods: GET, POST, PUT, DELETE, etc.
Status codes: 200 Ok, 404 Not found, etc.
Metadata: Content format, Location, etc.

Resource identifier
e.g., http://en.wikipedia.org/wiki/Representational_State_Transfer
HTTP verbs & behavior model

• Convenience operations
  ◦ HEAD Retrives metadata
  ◦ OPTIONS Asks which methods are supported by a resource

• Basic operations
  ◦ GET Retrieves a representation for a given resource
  ◦ DELETE Deletes an existing resource

• Other operations
  ◦ PUT Can be used for creating or replacing a resource
  ◦ POST Can also be used for creating or updating a resource
  ◦ PATCH Updates a resource
Resource creation with POST

POST /pos

HTTP/1.1 201 Created
Location: /pos/1253

http://rentit.com/rest/

BuildIt

<purchaseOrder>
  <rentalPeriod>
    <start>2017-06-21</start>
    <end>2017-06-24</end>
  </rentalPeriod>
  <plant>
    <name>Excavator</name>
  </plant>
</purchaseOrder>
Resource retrieval

GET /pos/1253

http://rentit.com/rest

HTTP/1.1 200 Ok

<purchaseOrder>
  <id>1253</id>
  <rentalPeriod>
    <start>2017-06-21</start>
    <end>2017-06-24</end>
  </rentalPeriod>
  <cost>1500.00</cost>
  <plant>
    <name>Excavator</name>
  </plant>
  <plant>
    <status>OPEN</status>
  </plant>
</purchaseOrder>
Resource creation: PUT vs. POST

• If PUT is used, then:
  ◦ the resource identifier is set by the client

  PUT /pos/1235

  ◦ the full representation of the resource is always provided

  <purchaseOrder>
  <id>1253</id>
  <start>2016-03-21</start>
  <end>2016-03-24</end>
  <cost>1500.00</cost>
  <plant>
    <name>Excavator</name>
  </plant>
  <status>OPEN</status>
  </purchaseOrder>

  Use PUT only in scenarios where a “full replacement” of the resource representation is implied

• If POST is used, then:
  ◦ the resource identifier is set by the server

  POST /pos

  ◦ only a part of the representation needs to be exchanged

  <purchaseOrder>
  <start>2016-03-21</start>
  <end>2016-03-24</end>
  <plant>
    <name>Excavator</name>
  </plant>
  </purchaseOrder>
Updating the resources (take 1)

• So far, there is not consensus on how to implement the "update" operation on resources
  ◦ Some say that we should use PUT. But should we use full or partial representation?
  ◦ Some others say that we should use POST

• To avoid ambiguity we will do the following:
  ◦ POST  Creation
  ◦ PUT    Replacement
  ◦ PATCH  Partial update

• But this is just a convention and not a norm

PATCH /pos/1235

<purchaseOrder>
  <end>2016-03-31</end>
</purchaseOrder>
HTTP status codes

• The HTTP status codes provide metadata about the state of resources

• They are part of what makes the Web a rich platform for building distributed systems

• They cover five broad categories
  ◦ 1xx – Metadata
  ◦ 2xx – Everything’s fine
  ◦ 3xx – Redirection
  ◦ 4xx – Client did something wrong
  ◦ 5xx – Server did a bad thing

The happy path:

- 200 Ok
- 201 Created
- 202 Accepted
- 204 No content

“Location” should contain the URI for accessing the resource whenever it becomes available:
• 201 – immediately
• 202 – subsequently
HTTP status codes

• The HTTP status codes provide metadata about the state of resources

• They are part of what makes the Web a rich platform for building distributed systems

• They cover five broad categories
  ◦ 1xx – Metadata
  ◦ 2xx – Everything’s fine
  ◦ 3xx – Redirection
  ◦ 4xx – Client did something wrong
  ◦ 5xx – Server did a bad thing

Exception notification:

400  Bad request
404  Not found
405  Method not allowed
406  Not acceptable
409  Conflict
REST API Design Method

1. Capture domain model
   ◦ High-level data model

2. Refine domain model into resource model
   ◦ Identify resources of different types and hypermedia relations

3. Specify the allowed interactions
   ◦ Use sequence diagrams to capture allowed interactions
   ◦ Next week: We will see how to use state diagrams to capture resources lifecycles

4. Derive a Resource Model

5. Map the resource model to an API
   ◦ Use the sequence diagram as a basis for designing the operations
Resource model

• A resource model is a **hierarchical data model** that captures the entities in the domain model (and their attributes) which we wish to expose in a REST API.

• A resource model consists of:
  ◦ An application **<<root>>** denoting the root of the REST API
  ◦ **<<container>>** classes, denoting collections of resources
  ◦ **<<resource instance>>** classes denoting individual resources
  ◦ **<<container>>** and **<<resource instance>>** classes are linked via composition relations, which means that the container is composed of a set of resource instances.
Deriving the Resource Model

1. The root of each aggregate in the domain model becomes a <<container>> class in the resource model
2. Each entity class (including the root of the aggregate) becomes a <<resource instance>> class
3. Each attribute of an entity class becomes an attribute of the respective <<resource instance>>
4. Each <<container>> of an entity class is linked to the <<resource instance>> via a composition association
5. Each 1-to-1 association within an aggregate of the domain model becomes a composition association between two <<resource instance>> classes
6. Each 1-to-N association within an aggregate of the the domain model becomes a composition association between a <<resource instance>> and a <<container>>
7. Each association between aggregates becomes a <<ref>> associations (we will see this next week)
Example

Mapping a 1-to-N relation within an aggregate
Example: Resource model

ResourceModel

- «container» PurchaseOrders
- «resource instance» PurchaseOrder
- «container» Plants
- «resource instance» PlantReservation

URLs:
- http://rentit.com/rest/pos
- http://rentit.com/rest/pos/{po.id}
- http://rentit.com/rest/pos/{po.id}/reservations

RentIt REST endpoint
{URI http://rentit.com/rest}
**Example: REST API**

<table>
<thead>
<tr>
<th>URI</th>
<th>HTTP verb</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/pos</td>
<td>GET</td>
<td>Retrieve all purchase orders</td>
</tr>
<tr>
<td></td>
<td>POST</td>
<td>Create new purchase order</td>
</tr>
<tr>
<td>/pos/{po.id}</td>
<td>GET</td>
<td>Retrieve a specific purchase order</td>
</tr>
<tr>
<td></td>
<td>PATCH</td>
<td>Update a purchase order</td>
</tr>
<tr>
<td></td>
<td>DELETE</td>
<td>Remove a purchase order</td>
</tr>
<tr>
<td>/pos/{po.id}/plants</td>
<td>GET</td>
<td>Retrieve all plants associated to a given purchase order</td>
</tr>
<tr>
<td></td>
<td>POST</td>
<td>Add a plant to a given purchase order</td>
</tr>
<tr>
<td>/pos/{po.id}/plants/{plant.id}</td>
<td>GET</td>
<td>Retrieve a specific plant within a purchase order</td>
</tr>
<tr>
<td></td>
<td>PATCH</td>
<td>Modify a plant?</td>
</tr>
<tr>
<td></td>
<td>DELETE</td>
<td>Remove a plant from a purchase order</td>
</tr>
</tbody>
</table>
Querying and filtering

- In some cases, we will need to restrict our attention to a subset of resource instances:
  - Only open purchase orders
  - Invoices within a fiscal period

- It is also interesting to use certain criteria when querying a “container”
  - Set of plants of certain capacities and available in for a given period of time

- In those cases, use URI query templates of the form:

  GET /plants?name=excavator

  GET /phrs?status=OPEN

- Note: All the above URIs should be requested via an HTTP GET
Implementing the integration layer
Domain object vs. Resource

```java
@Entity
@Getter
public class Plant {
    @Id
    private Long id;
    private String name;
    private String description;
    private BigDecimal price;
}

@Data
public class PlantDTO extends ResourceSupport {
    private Long _id;
    private String name;
    private String description;
    private BigDecimal price;
}

public class PlantAssembler extends ResourceAssemblerSupport<Plant, PlantDTO> {
    public PlantAssembler() {
        super(InventoryRestController.class, PlantDTO.class);
    }
    public PlantDTO toResource(Plant plant) {
        ...
    }
    public List<PlantDTO> toResource(List<Plant> plants) {
        ...
    }
}
```
@RestController
@RequestMapping("/api/orders")
public class PurchaseOrderRestController {

    @RequestMapping(method = RequestMethod.GET, path = "")
    public List<PurchaseOrderResource> getAllPOs() {
        ...
    }

    @PostMapping
    public ResponseEntity<Plant> createPO(@RequestBody PurchaseOrderResource res) {
        ...
    }

    @GetMapping="/pos/{id}"
    public PurchaseOrderResource getPO(@PathVariable Long id) {
        ...
    }

    @PostMapping="/pos/{id}/plants"
    public ResponseEntity<PurchaseOrderResource> addPlantToPO(@PathVariable Long id, @RequestBody PlantResource res) {
        ...
    }
}
@RestController
@RequestMapping("/api/orders")
public class PurchaseOrderRestController {
    @Autowired
    SalesService salesService;

    @PostMapping
    public ResponseEntity<PurchaseOrderDTO> createPO(@RequestBody PurchaseOrderDTO inputPO) {
        PurchaseOrderDTO po = salesService.createPO(inputPO);
        HttpHeaders headers = new HttpHeaders();
        URI location = ServletUriComponentsBuilder.fromCurrentRequestUri().
            pathSegment(po.get_id()).build().toUri();
        headers.setLocation(location);
        return new ResponseEntity<>(po, headers, HttpStatus.CREATED);
    }
}
Example

$ curl -v -X POST -H "Content-type: application/xml" \ 
   -d @po.xml http://localhost:8080/api/orders

> POST /api/orders HTTP/1.1
> Host: localhost:8080
> Accept: */*
> Content-type: application/xml

< HTTP/1.1 201 Created
< Location: http://localhost:8080/api/orders/1
Retrieving a resource

```java
@RestController
@RequestMapping("/api/orders")
public class PurchaseOrderRestController {

    @Autowired
    SalesService salesService;

    @GetMapping
    @ResponseStatus(HttpStatus.OK)
    public List<PurchaseOrderDTO> getAllPOs() {
        return salesService.findAllPurchaseOrders();
    }
}
```
Content negotiation

```
< HTTP/1.1 200 OK
< Content-Type: application/json; charset=UTF-8

{
    "purchaseOrder": [ {
        "startDate":1381093200000,
        "endDate":1381438800000,
        "total":1250.0,
        "plant":null
    } ]
}
```
Content negotiation


< HTTP/1.1 200 OK
< Content-Type: application/xml;charset=UTF-8

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<purchaseOrders>
  <purchaseOrder>
    <endDate>2013-10-11T00:00:00+03:00</endDate>
    <startDate>2013-10-07T00:00:00+03:00</startDate>
    <total>1250.0</total>
  </purchaseOrder>
</purchaseOrders>
Exception handling within our REST controller

```java
@RestController
@RequestMapping("/api/orders")
public class PurchaseOrderRestController {

    @ExceptionHandler({PlantUnavailableException.class,
                        InvalidRentalPeriodException.class})
    public ResponseEntity<String> handleBadRequest(Exception ex) {
        return new ResponseEntity<>(ex.getMessage(), HttpStatus.CONFLICT);
    }

    @ExceptionHandler(EntityNotFoundException.class)
    public ResponseEntity<String> handleNotFound(Exception ex) {
        return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT_FOUND);
    }
}
```