Overview of last week
What if the execution of an enterprise software spans across the boundaries of an organization?

- Goal: Interoperability
- Challenges: Distribution, Heterogeneity, Loose-coupling, Security, etc.

Our focus will be on two architectural styles:
- REST: Resource oriented architecture
- SOA: Service oriented architecture
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
(Conceptual) 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
The problem in a nutshell

• An application programming interface (API) specifies how some software components should interact with each other
• Our problem consists in designing APIs for some of the components in the enterprise software
Approach number 1

- For each resource identified in your system, list all possible operation and associate a unique URI to each of them.

```
/getPlant  /createPlantHireRequest
/getAllPlants /acceptPlantHireRequest
/isPlantAvailable /rejectPlantHireRequest

/createPurchaseOrder /createRemittanceAdvice
/acceptPurchaseOrder /createInvoiceResponse
/rejectPurchaseOrder /createCreditNote
/getPurchaseOrder ...
/getAllPurchaseOrders /createInvoice
/updatePurchaseOrder /sendInvoiceToCustomer
/isPurchaseOrderOpen /getInvoiceStatus
/isPurchaseOrderPaid /acceptInvoice
/rejectInvoice /rejectInvoice
```
Approach 1, variant 1

GET
/isPlantAvailable?sku=exc1253ab98&start=2013-10-07&end=2013-10-11

Full URI:
Approach 1, variant 2

Client

POST

/isPlantAvailable

Server

<request>
  <plant>
    <name>Excavator</name>
  </plant>
  <period>
    <start>7-10-2013</start>
    <end>11-10-2013</end>
  </period>
</request>

<response>
  true
</response>

Full URI:
http://rentit.com/api/isPlantAvailable
Approach 1, analysis

**PROs**

- Very easy to understand
- Great for simple procedure-calls
- Simple to code
  - Do it with the servlet API, HttpListener, IHttpHandler, RAILS, whatever!
- Interoperable
  - It’s just URIs!

**CONs**

- It’s brittle RPC!
- Tight coupling, no metadata
  - No typing or “return values” specified in the URI
- Not robust
  - You have to handle failure cases manually
- No metadata support
  - You must construct the URIs yourself, map them to the function manually
Approach 2

POST

http://rentit.com/api

Client

Server

<isPlantAvailable>
  <plant>
    <name>Excavator</name>
  </plant>
  <period>
    <start>7-10-2013</start>
    <end>11-10-2013</end>
  </period>
</isPlantAvailable>

<isPlantAvailableResponse>
  true
</isPlantAvailableResponse>
Approach 2, analysis

**PROs**
- Simplicity – just use HTTP POST and XML
- Re-use existing infrastructure and libraries
- Interoperable
  - It’s just XML and HTTP
- Can use complex data structures
  - By encoding them in XML

**CONs**
- Client and server must collude on XML payload
  - Tightly coupled approach
- No metadata support
- Does not use Web for robustness

This approach is a kind of (homemade) variant of a SOAP-like middleware

→ Consider better using SOAP + WS-* for robustness
Richardson Maturity Model

Approach 1 (a.k.a. URI tunnelling)

• Lots of URIs
  ◦ But really has a more level 0 mindset

• Doesn’t understand HTTP
  ◦ Only one verb

• No hypermedia

Approach 2 (a.k.a. POX-based REST)

• Single well-known endpoint
  ◦ Not really URI friendly

• Doesn’t understand HTTP
  ◦ Only one verb

• No hypermedia
What is a resource?

• Let’s get inspired from the WWW
  ◦ Everything is a resource: an html document, a video, an audio, etc.

• URIs as handles to resources:
  ◦ Each resource has at least one URI
  ◦ Resources can have more than one URI
  ◦ http://foo.com/software/latest
  ◦ http://foo.com/software/v1.4
  ◦ Canonical URIs are long-lived

• Hint: Resources are usually described by nouns and not verbs in a software specification
  ◦ /orders vs. /createPurchaseOrder
Example: Resource model

```
ResourceModel

«container» PurchaseOrders

«root»
RentIt REST endpoint
{URI http://rentit.com/rest}

«container» Plants

«resource instance» PurchaseOrder

start: Date
end: Date
cost: Float

http://rentit.com/rest/pos

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

http://rentit.com/rest/pos/{po.id}/plants

«resource instance» POPlants

name: String
description: String
cost: Float

«container» Plant
```
HTTP verbs & behavior model

• Convenience operations
  ◦ HEAD          Retrieves 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

HTTP/1.1 201 Created
Location: /pos/1253

POST /pos

http://rentit.com/rest

<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/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>
  <status>OPEN</status>
</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>
## URI patterns and HTTP verbs

<table>
<thead>
<tr>
<th>URI</th>
<th>HTTP verb</th>
<th>Comments</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
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
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) {
        ...
    }
}
Resource creation

```java
@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);
    }
}
```