Overview of last week
A picture of todays lecture

- 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
Service Analysis & Design

• **Service Analysis.** identification and definition of the business services that an organization provides or consumes, internally or externally.
  → Collection of business-level services, their context and their interactions

• **Service API Design.** identification and definition of technical services (e.g. REST) to support the delivery of business services.
  → Collection of service APIs (operations, inputs & outputs)
Decomposing Information Systems

1. **Functional or capability-driven decomposition**
   - Which business functions, capabilities or specialization perform work in the organization (organizational chart)

2. **Process-driven decomposition**
   - Which chains of activities create value to the organization?

3. **Data-driven decomposition**
   - What information is stored & circulates in the organization?
Service Analysis and Design Methods

**Capability-driven methods**


**Process-driven methods**


**Data-driven methods**

Outlook

Buildit

SalesRestController

SalesService

PODTO'

PlantDTO'

REST adapter

PODTO

PlantDTO

SalesController

PurchaseOrder

OrdersRepository

PlantRepository

Plant

Web UI adapter

JPA adapter

DDD & DATA ACCESS  LUCIANO GARCÍA-BAÑUELOS  6
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
Capability-driven Service Analysis

• Service
  ◦ Sub-system where a collection of tasks are performed to achieve a goal.

• Actor
  ◦ User or “external” system (application) that interacts with a given service

• Interaction
  ◦ An encounter between two services or between an actor and a service to achieve a goal.
  ◦ Includes: physical or electronic, manual or automatic
Capability-driven Service Analysis

- Definition of the services scope: what the services are
- External actors driving or interacting with the services
- Reasons of the service-to-service and service-to-actor interactions
- Details of services to be delivered by the IT team

<table>
<thead>
<tr>
<th>WHAT</th>
<th>Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHO</td>
<td>Actors</td>
</tr>
<tr>
<td>WHY</td>
<td>Interactions</td>
</tr>
<tr>
<td>HOW</td>
<td>API Design (resources &amp; operations)</td>
</tr>
</tbody>
</table>
The framework is applied at increasingly finer levels of granularity

- **Level 0.** Definition of the core services to the business domain.
- **Level 1.** Decomposition of the Level 0 model into finer-grained services for each core service.
- **Level 2+** Refinements of previous levels to identify support & shared services integrating & complementing the services already defined.
"Level 0" Context Diagram

**What**
- Manufacture
- Logistics & Warehouse
- Sales
- Finance

**Who**
- Customer
- Suppliers
- Logistics Company

**Why**
- Order
- Invoice
- Contact
- Payment
- Make Item(s)
- Ship Order
- Check Stock
- Deliver
- Add to Stock
- Bill Customer
- Supply

**Rule of thumb:**
2-8 services at this level
“Level 1” Context Diagram (Manufacture)

What

Research & Development → Supplies In

Who

Regulatory Approval → Product Manufacture

Why

Regulator → Order Supplies

Diagrams:

- Supply Stock
- Product Manufacture
- Quality Control
- Packaging
- Stock Management
- Order Supplies
- External

Legend:

- Manufacture
- Sales
- Logistics & Warehouse
- Finance
Process-Driven Service Design

1. Identify which process(es) need to be supported and their actors

2. Capture each process (e.g. as a process model)

3. Analyze tasks/events in process model to identify:
   ◦ Atomic interactions (operations) that the process should provide
   ◦ Atomic interactions (operations) that the process requires from other systems
Process-Driven Service Design
Steps 1. Identifying Process & Actors

**External Actor**
- **Customer**
  - Procurement Service

**Supplier**
- (Order Management Service)

**Process we need to support**

**Service Required by Process**
- **Warehouse**
  - Inventory Control Service
- **Finance**
  - Invoicing Service
2. Capturing the Order Management Process
Step 3. Identify operations (Order Management Service)

Provided operations (to customer)

- Receive RFQ
- Receive order

Required operations

- From inventory control
  - Check stock availability
  - Request order shipment
- Invoicing
  - Request invoicing
Step 3. Identify Operations

- Less obvious provided operations of order management service
  - Cancel order/request order cancellation
  - Modify order/request order modification
  - Check order status
  - ...
Step 3. Identify operations

Less obvious required operations

• To prepare quote
  ◦ Retrieve price list
  ◦ Retrieve customer data (contractual terms)

• To prepare shipment order
  ◦ Retrieve customer delivery data

• To prepare invoice request
  ◦ Retrieve customer invoicing data
Rentit’s domain

Sub-domain

Domain

Reservations

Plant reservation

Maintenance

Plan catalog and purchase orders

Transportation

Invoicing

Bounded Context

Sales (core)

Invoicing

Plant transportation

Maintenance plan

Plant catalog

Rentit’s domain

Sub-domain

Domain

Reservations

Plant reservation

Maintenance

Plan catalog and purchase orders

Transportation

Invoicing

Bounded Context

Sales (core)

Invoicing

Plant transportation

Maintenance plan

Plant catalog

Sub-domain

Domain

Reservations

Plant reservation

Maintenance

Plan catalog and purchase orders

Transportation

Invoicing

Bounded Context

Sales (core)

Invoicing

Plant transportation

Maintenance plan

Plant catalog

Sub-domain

Domain

Reservations

Plant reservation

Maintenance

Plan catalog and purchase orders

Transportation

Invoicing

Bounded Context

Sales (core)

Invoicing

Plant transportation

Maintenance plan

Plant catalog
Rentit’s domain model
Example repository

```java
public interface PlantCatalog
{
    <interface>
    PlantCatalog

    void fetchAllPlants()
    PlantCatalog

    PlantInventoryItem findPlantsByName(String name)
    PlantCatalog

    PlantInventoryEntry findAvailablePlants(PlantInventoryEntry entry, BusinessPeriod period)
    PlantCatalog

    void bookPlantForRental(PlantInventoryItem item, BusinessPeriod period)
    PlantReservation

    BusinessPeriod
    startDate: Date
    endDate: Date
}
```
Data-Driven Service Design

• Suitable particularly for RESTful service design
• Starts from a data model
• Key entities in model become “resources”
• CRUD framework used to identify operations
• One possible method (you must read it!):
Data-Driven Service Design

1. Capture domain model
   ◦ High-level data model

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

3. Derive REST API
   ◦ Use CRUD (or SCRUD) framework
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

**Hypermedia**

**HTTP**

**URI**
Approach number 1

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

/getPlant
/getAllPlants
/isPlantAvailable

/createPurchaseOrder
/acceptPurchaseOrder
/rejectPurchaseOrder
/getPurchaseOrder
/getAllPurchaseOrders
/updatePurchaseOrder
/isPurchaseOrderOpen
/isPurchaseOrderPaid

/createPlantHireRequest
/acceptPlantHireRequest
/rejectPlantHireRequest
/updatePlantHireRequest

/createInvoice
/sendInvoiceToCustomer
/getInvoiceStatus
/acceptInvoice
/rejectInvoice

/createRemittanceAdvice
/createInvoiceResponse
/createCreditNote

...
Example

ResourceModel

Plant

name: String
description: String
price: Float

PurchaseOrder

start: Date
date: Date
cost: Float

Customer

name: String
vatNumber: String

DomainModel

Plant

name: String
description: String
price: Float

PurchaseOrder

start: Date
date: Date
cost: Float
Example: Resource model

ResourceModel

- «container» PurchaseOrders
- «container» Plants
- «root» RentIt REST endpoint
  {URI http://rentit.com/rest}
- «resource instance» PurchaseOrder
  start: Date
date: Date
cost: Float
- «container» POPlants
- «resource instance» Plant
  name: String
description: String
price: Float

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

POST /pos

HTTP/1.1 201 Created
Location: /pos/1253

http://rentit.com/rest

BuildIt

RentIt

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

  ```xml
  <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

  ```xml
  <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
## 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>POST</td>
<td></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>PATCH</td>
<td></td>
<td>Update a purchase order</td>
</tr>
<tr>
<td>DELETE</td>
<td></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>POST</td>
<td></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>PATCH</td>
<td></td>
<td>Modify a plant?</td>
</tr>
<tr>
<td>DELETE</td>
<td></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) {
        ...
    }
}
```
REST service façade

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