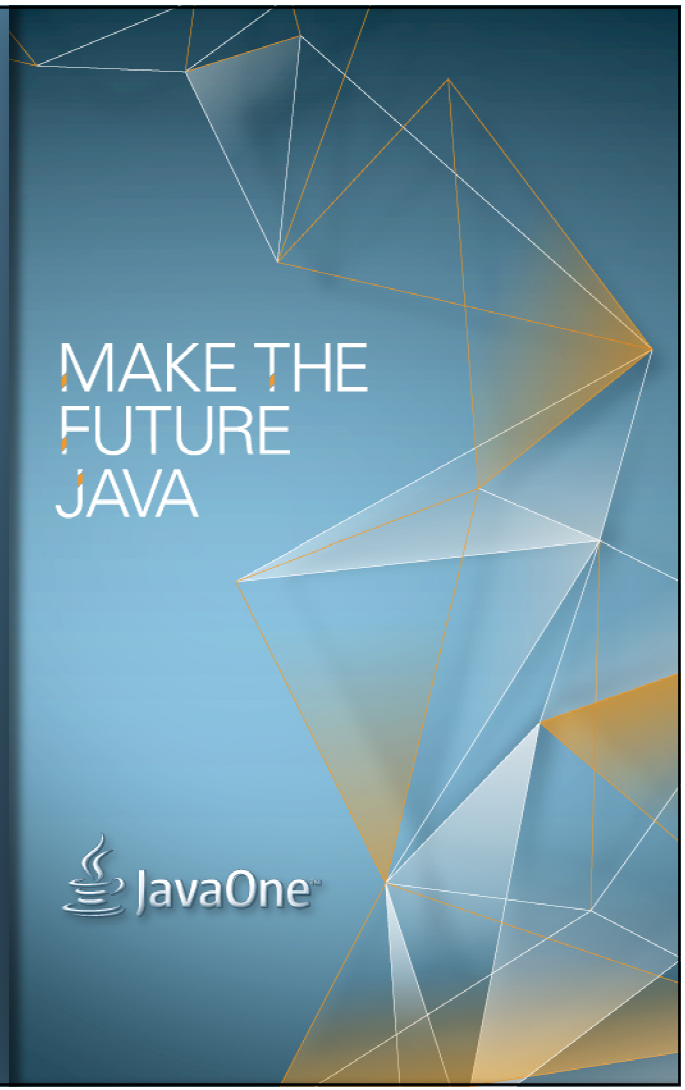
JavaOne™

ORACLE

Using JSR 303, Bean Validation, with the Common Data Model in SOA

Donatas Valys
Client Architect SOA



Donatas Valys

Client Architect SOA , Oracle

Speaker Bio

Donatas Valys is a senior principal consultant at the Oracle Consulting in Germany. He has been working for more than 15 years in the area of enterprise software development and integration with Java and Oracle technologies. At Oracle Consulting he works as a client architect, developer and consultant in SOA, integration, Oracle ADF and Java EE projects helping enterprise customers to achieve predictable success.

Speaker Sessions

- Using JSR 303, Bean Validation, with the Common Data Model in SOA

mailto:donatas.valys@oracle.com
http://donatas.nicequestion.com
@donatasnq Follow @donatasnq

JavaOne ORACLE

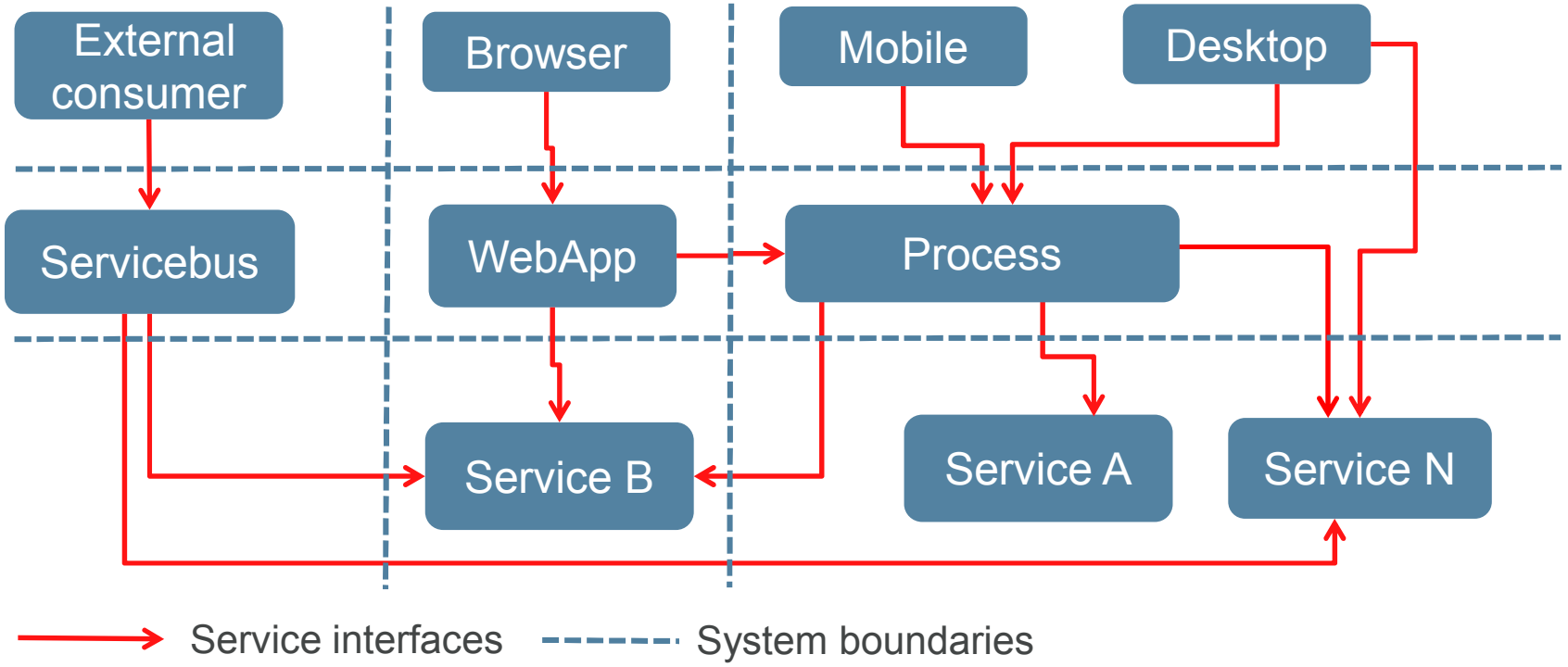
Program Agenda

- Challenges in layered SOA
- Using Bean Validation in SOA
  - Annotation design for the common data model
  - Implement simple and complex semantic business constraints
  - Reuse and enforce them on service provider and consumer side

Challenges in layered SOA

Layered SOA

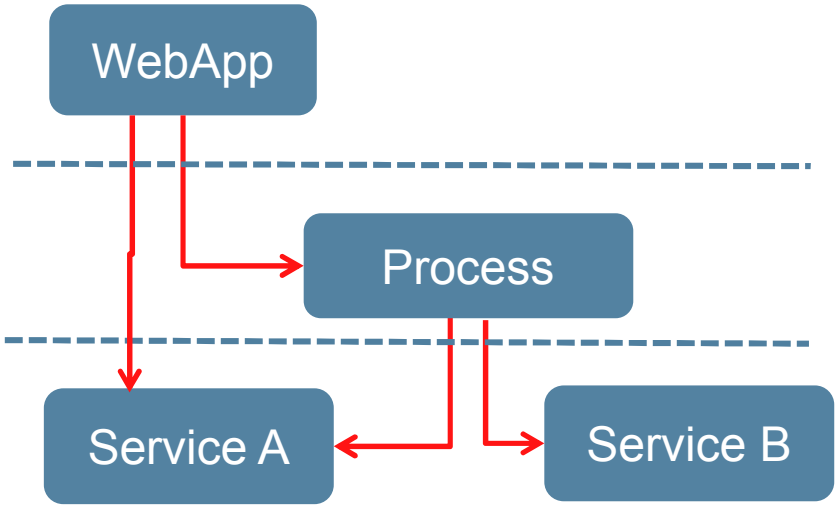
Service providers and consumers



Service interface

Describing service interface using java

```
public interface ServiceA {
   public returnMessage
      doSomething(InputMessage input)
}
```



System boundaries

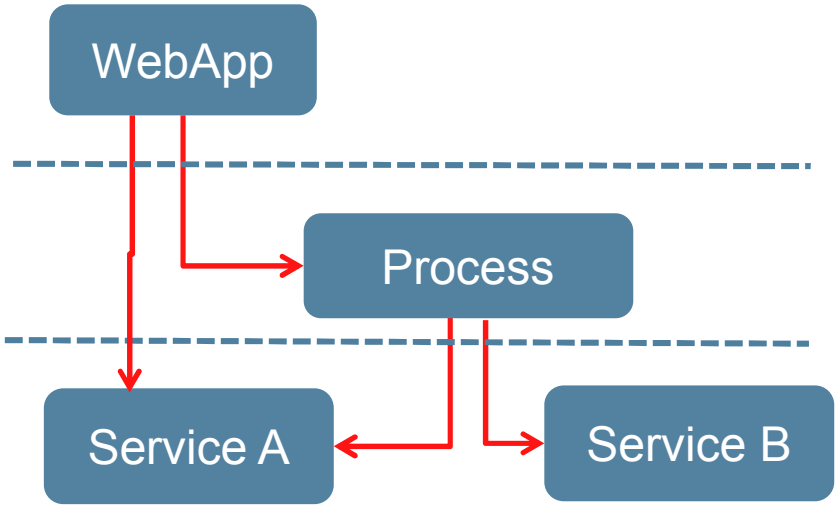
What about WSDL, XSD, SOAP, HTTP etc.?

Rethinking “best practices”

- Service != allways WebService
- Service != allways SOAP

Focus on Enterprise API:

- Business interfaces
- Business data
- Business constraints

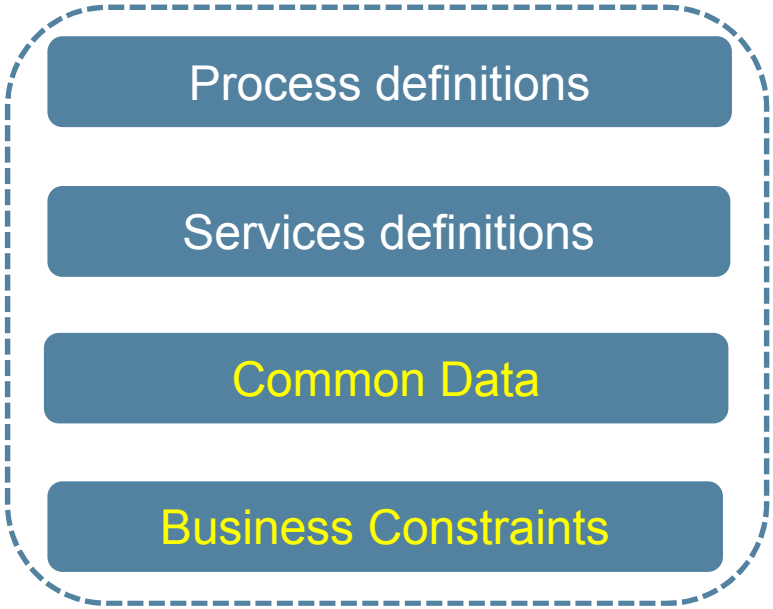


System boundaries

 | |

Service model - *Enterprise API*

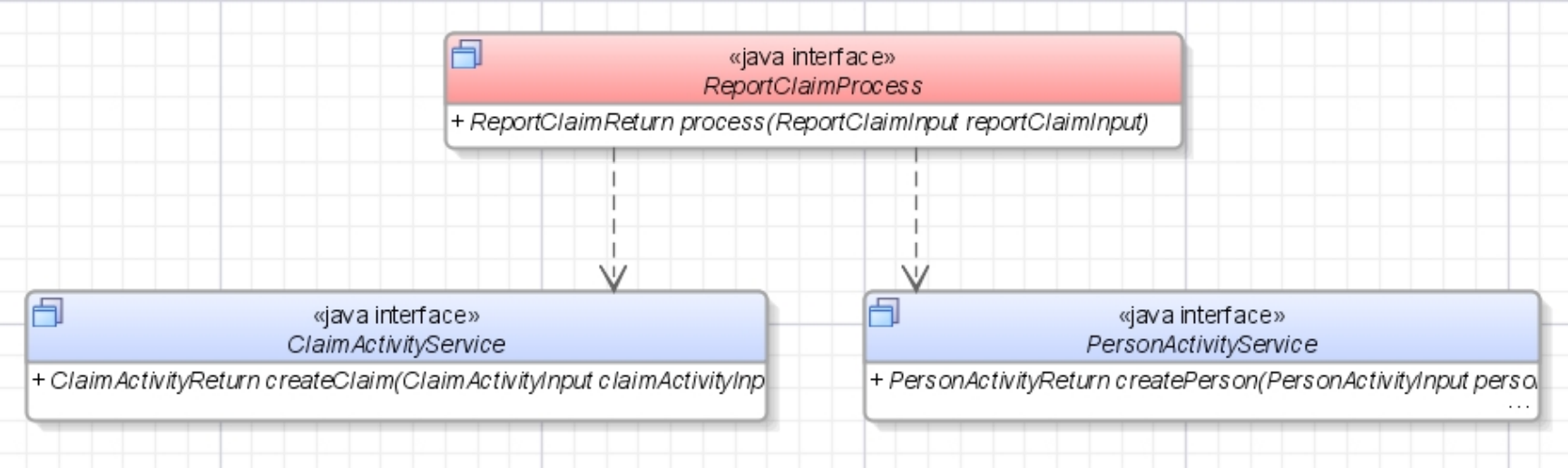
Defining sample service model for insurance domain



- ReportClaimProcess
  - *process*
- ClaimActivityService
  - *createClaim*
- ReportClaimInput,
  Claim, Person
- *NotNull, Size*
  ClaimConstraint

Process and service definition

Sample process and service interfaces

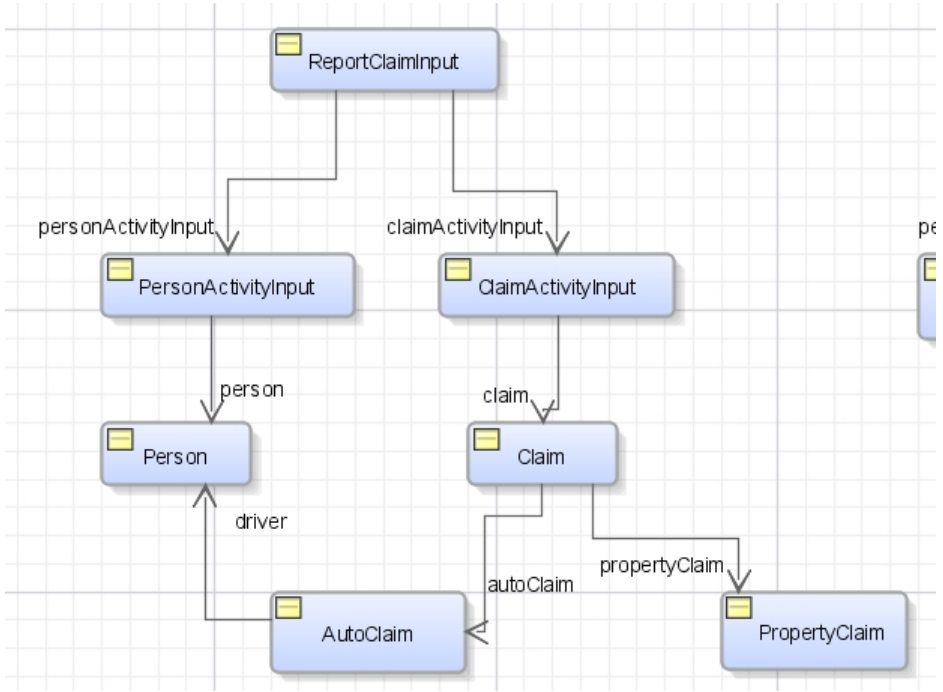


 | |

Common Data – Vocabulary for *Enterprise API*

Defining common data model for the sample business domain

- Data transfer objects
- Composition
- No behavior
- *Remote transfer*



 | |

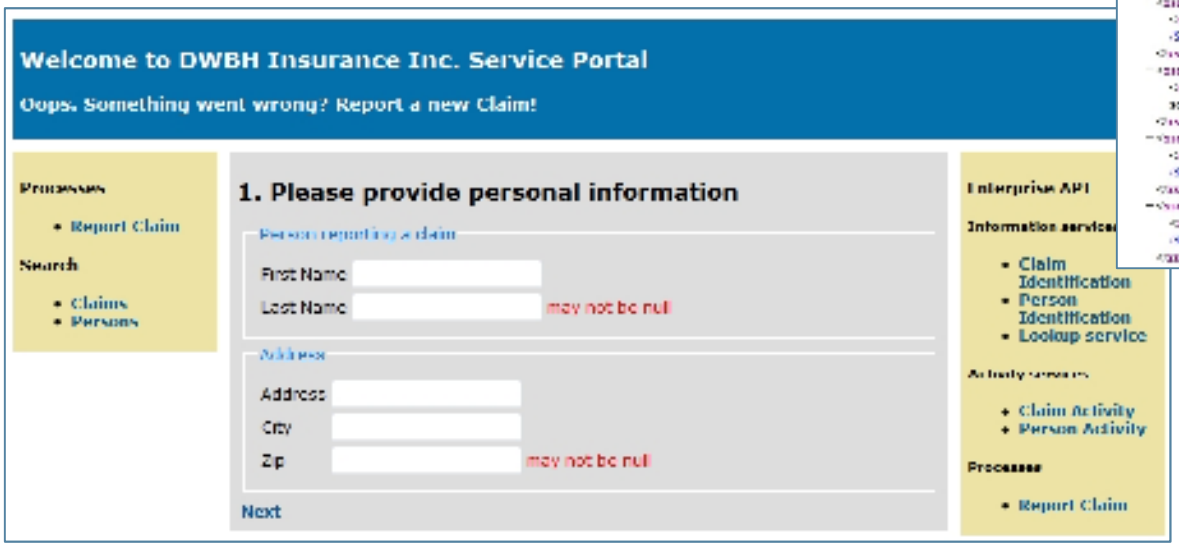
Using Bean Validation in SOA

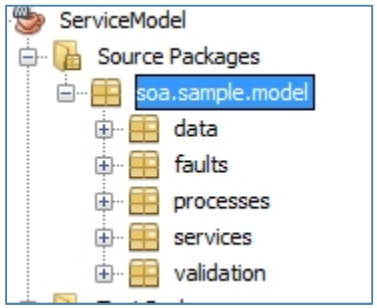
 | |

JavaOne ORACLE

DWBH Insurance Service Portal

Demo application

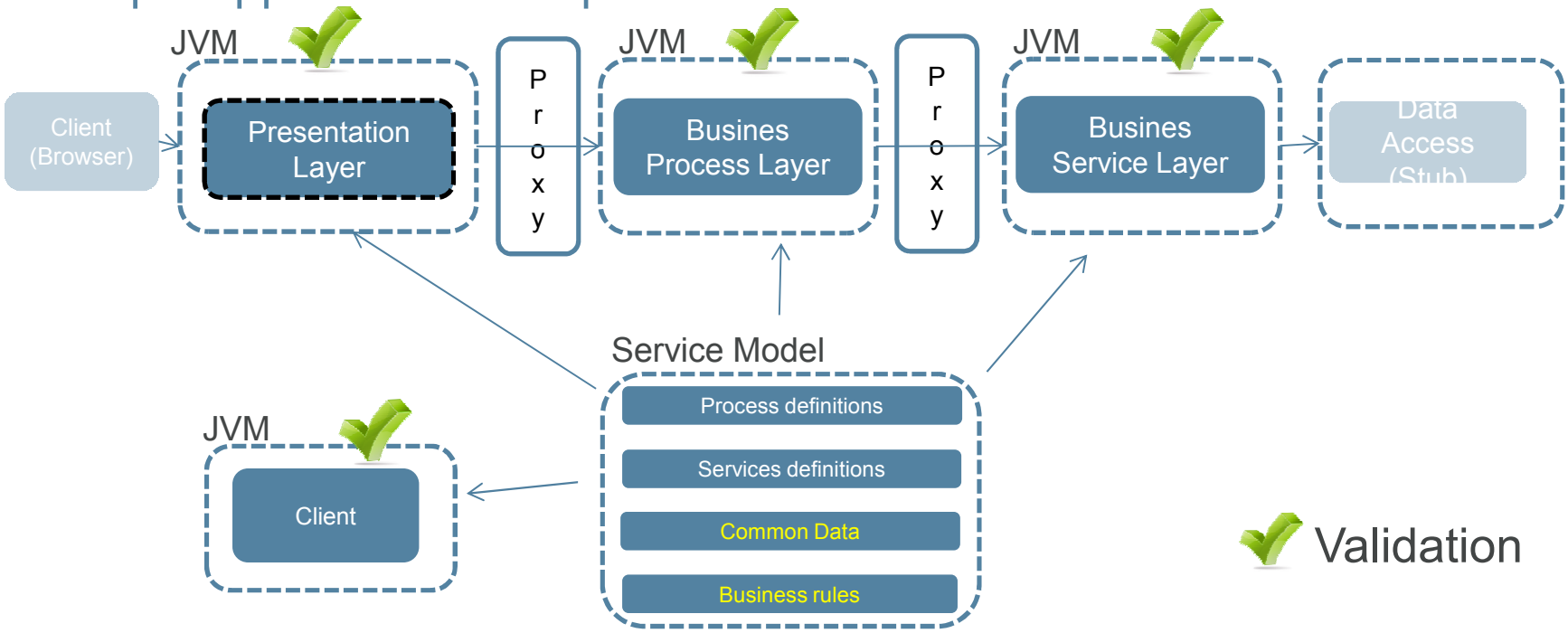




 | |

Validation and SOA

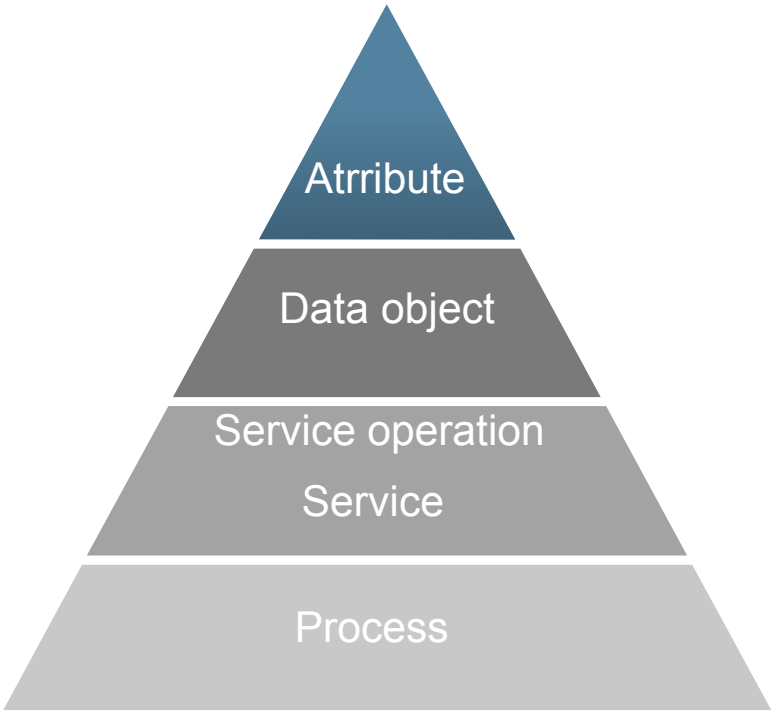
Sample application conceptual architecture

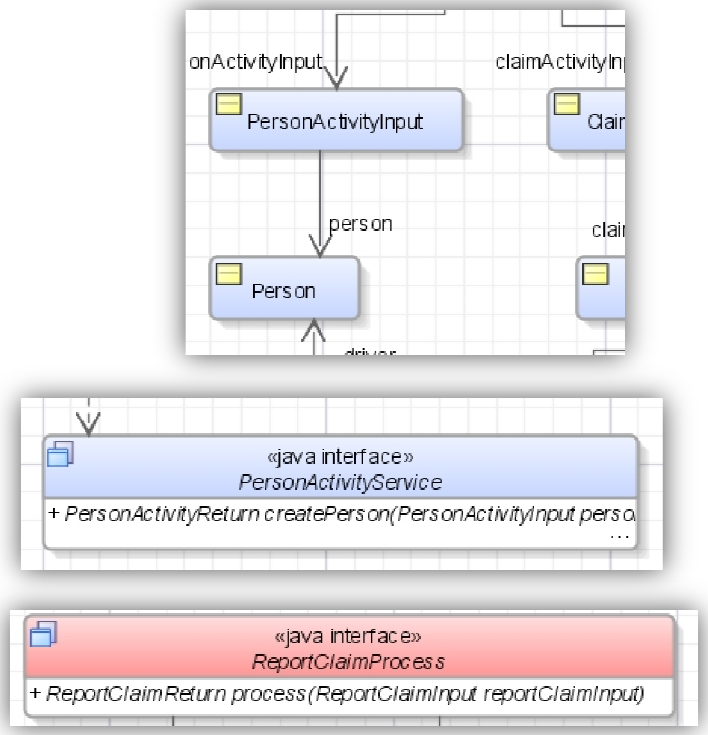


 | | CON5020 Donatas Valys

Granularity of Business Constraints

Simple to complex





 | |

JSR 303 Bean Validation

Preface

JSR 303 - Bean Validation - defines a metadata model and API for entity validation. The default metadata source is annotations, with the ability to override and extend the meta-data through the use of XML. The API is not tied to a specific application tier or programming model. It is specifically not tied to either the web tier or the persistence tier, and is available for both server-side application programming, as well as rich client Swing application developers.

Hibernate Validator JSR 303 Reference Implementation Reference Guide 4.3.0.Final
http://docs.jboss.org/hibernate/validator/4.3/reference/en-US/pdf/hibernate_validator_reference.pdf

Defining Bean Validation Constraints

Example: class Person

- Identifier: Id
  - **@Null** – when creating
  - **@NotNull** – when updating
- @Size – implicit group **Default**

- Custom validation groups:
  - **New** – when creating
  - **Update** – when updating

```
public class Person implements Serializable {

    @Null(groups = New.class)
    @NotNull(groups = Update.class)
    private Long id;

    @Null(groups = New.class)
    private Integer personalNumber;

    @Size(min=2, max = 10)
    private String firstName;
```

```
public interface New { }

public interface Update { }
```


Object Graph Validation

Example: class PersonActivityInput

- **@Valid** – apply validations to *dependant* objects **person**, **address**
- **PersonActivityInput** – *input message , a* container for **PersonActivityService** *operations*: **createPerson**, **updatePerson**

```
public class PersonActivityInput implements Serializable {

    @Valid
    @NotNull
    private Person person;

    @Valid
    @NotNull
    private Address address;


public interface PersonActivityService {

    public PersonActivityReturn createPerson
            (PersonActivityInput personActivityInput)
                throws BusinessException;
```


Using Groups for Service-specific Constrains

Example: class Address

- **address, city** – @NotNull for **ClaimActivityService**
- **zip** – @NotNull also for **PersonActivityService**

Service interfaces as bean validation groups for service-specific constraints

```
public class Address implements Serializable {

    @Null(groups = New.class)
    @NotNull(groups = Update.class)
    private Long id;

    @Size(max = 200)
    @NotNull(groups  = {ClaimActivityService.class})
    private String address;

    @Size(max = 50)
    @NotNull(groups  = {ClaimActivityService.class})
    private String city;

    @Size(min = 5, max = 10)
    @NotNull(groups  = {
        PersonActivityService.class,
        ClaimActivityService.class}
            )
    private String zip;
```


Using Class-level Custom Constraints

Example: class Claim

- ClaimType: Auto, Property
- if ***type = Auto*** then
  autoClaim not null
- if ***type = Property*** then
  propertyClaim not null
- **Custom @ClaimConstraint**

```
public enum ClaimType  {
    Auto("AUTO","Auto"),Property("PROP","Property");

    private String code;
    private String title;


@ClaimConstraint
public class Claim implements Serializable {

    @NotNull
    private ClaimType type;

    @Valid
    private AutoClaim autoClaim;

    @Valid
    private PropertyClaim propertyClaim;
```

 | |

Defining Custom Constraint Annotation

Example: contraint @ClaimConstraint

```
@Target( { METHOD, FIELD, ANNOTATION_TYPE, TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = ClaimValidator.class)
@Documented
public @interface ClaimConstraint {

    String message() default "{soa.sample.model.validation.constraints.ClaimConstraint}";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

- *@interface* keyword for annotation type
- Mandatory attributes: *message, groups and payload*
- *@Contraint* (*validatedBy* = **ClaimValidator.class**) specifies validator

Implementing Custom Validator

Example: class ClaimValidator

```
@Override
public boolean isValid(Claim claim, ConstraintValidatorContext constraintValidatorContext) {
    // Do not validate null values
    if (claim == null || claim.getType() == null)
        return true;
    if (claim.getType().equals(ClaimType.Auto) && claim.getAutoClaim() == null)
        return false;
    if (claim.getType().equals(ClaimType.Property) && claim.getPropertyClaim() == null)
        return false;
```

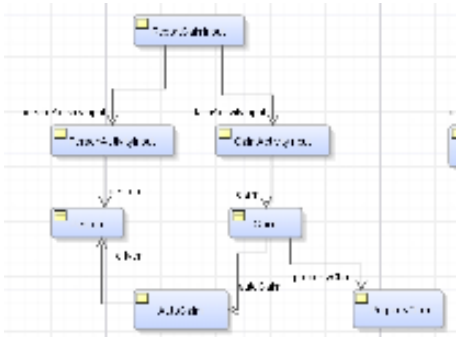
- **Claim** object **instance** available
- isValid – true or false
- no validation for null here

Process-level Validation Constraint

Example: class ReportClaimInputValidator

```
// Business rule:
// "Do not accept auto claims from a persons whose lastname starts wiht 'X' (bad experience)
//
// The business rule is just intended to show, that on a business process level a validation logic can
// access all process data. Therefore more complex and process-specific business rules can be implemented here
if (reportClaimInput.getClaimActivityInput().getClaim().getType() == ClaimType.Auto &&
    reportClaimInput.getPersonActivityInput().getPerson().getLastName().toUpperCase().startsWith("X")) {
    return false;
}
```

- **ReportClaimInput composes all process data**
- Complex semantic process-specific validations



Unique Validation Constraint

Example: class Person

```
@PersonUniqueConstraint(message="Person already exists",groups= {Unique.class})
public class Person implements Serializable {
```

- *Resource intensive – probably* ***significant amounts of data*** *in memory required*

- **Dedicated group *Unique.class***
- **Delegate work to backend (database, etc.)**

 | |

Designing Unique Validator

Example: class PersonUniqueValidator

```
@Alternative
public class PersonUniqueValidator implements ConstraintValidator<PersonUniqueConstraint, Person> {

    @Inject ConstraintValidator<PersonUniqueConstraint, Person> delegateValidator;
```

- Using CDI to inject delegate
- CDI @Alternative to avoid cyclic references
- Provide environment specific delegate implementation

Implementing Unique Validator Delegate

Example: class PersonUniqueValidatorStub

```
public class PersonUniqueValidationStub implements ConstraintValidator<PersonUniqueConstraint, Person> {

    private DataStoreStub dataStore = DataStoreStub.DATABASE;

    public boolean isValid(Person person, ConstraintValidatorContext context) {
        return !dataStore.personExists(person);
    }
```

- **Service provider layer**
- **Delegate to backend data store to check if person exists**

Implementing Unique Validator Delegate

Example: class ValidatorProxyProducer

```
public class ValidatorProxyProducer {
    @Inject PersonIdentificationService service;
    @Produces ConstraintValidator<PersonUniqueConstraint, Person> getClaimIdentificationService() {
        return new ConstraintValidator<PersonUniqueConstraint, Person>() {
            public void initialize(PersonUniqueConstraint constraintAnnotation) {
            }
            public boolean isValid(Person value, ConstraintValidatorContext context) {
                return !service.personExists(value);
            }
```

- Service consumer layer
- Using CDI to inject service proxy and produce validator
- Delegate to exposed service to check if person exists

Enforcing Validation

Enforcing Validation - Service Provider

Example: class PersonActivityServiceStub

```
public PersonActivityReturn createPerson(PersonActivityInput personCreateInput) throws BusinessException {

    Set<ConstraintViolation<PersonActivityInput>> constraintViolations =
            ValidationUtils.getValidator().validate(personCreateInput, PersonActivityCreatePerson.class);

    ValidationUtils.throwBusinessException(constraintViolations);
```

- **Using BeanValidation API: *validate(Object, { Groups })***
- **Get ContraintViolation set as a result**
- **Convert to domain specific BusinessException**
- **Using a specific validation group for service operation**

Service operation specific validation

Example: interface PersonActivityCreatePerson

- public interface PersonActivityUpdatePerson extends Default, Update , PersonActivityService { }
- public interface PersonActivityCreatePerson extends Default, New, Unique, PersonActivityService { }

- **Combine validation groups**
- **Validate constraints specific to service operation**

Enforcing Validation - Service Consumer

Example JSF2: identifyPerson.xhtml

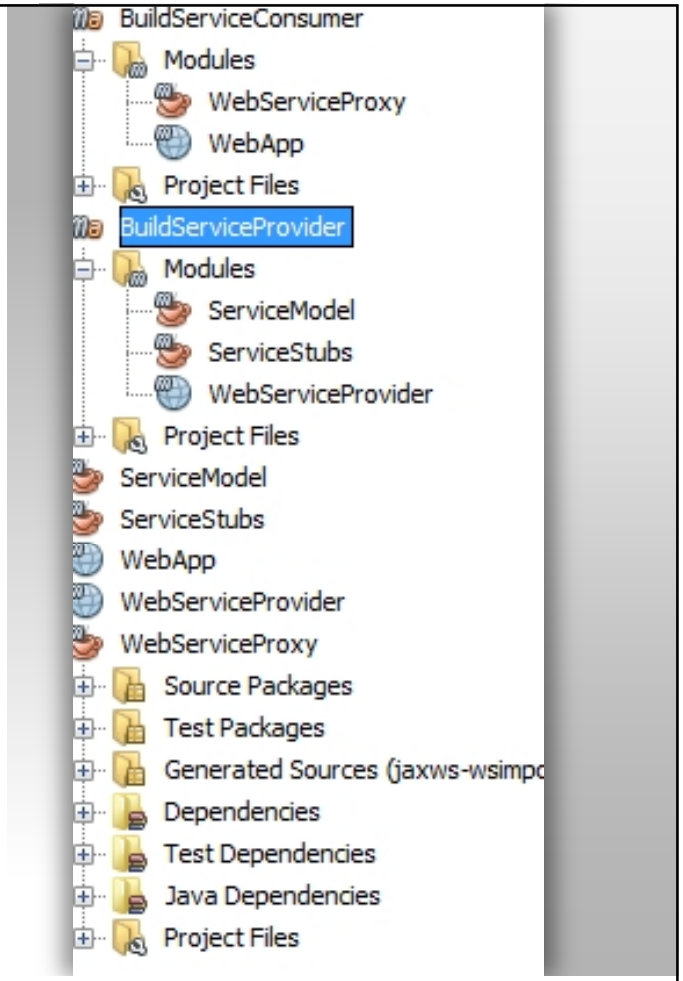
```
<f:validateBean validationGroups="soa.sample.model.validation.groups.PersonActivityCreatePerson">

    <h:messages globalOnly="true" class="error"/>
    <fieldset id= "person">
        <legend>Person reporting a claim</legend>

        <h:panelGrid columns="3">

            <h:outputLabel for="firstname" value="First Name" />
            <h:inputText id="firstname" value="#{ReportClaimProcessController.reportClaimInput.pers
            <h:message for="firstname"/>
```

- **<f:validateBean> - validates attribute values**
- **Using the same validation group**
- **Using common data model – *reportClaimInput.person***

Demo Application

Download
https://github.com/donatasnicequestion

1. Build Service Provider
2. Run ***WebServiceProvider***
3. Build Service Consumer
4. Run ***WebApp***

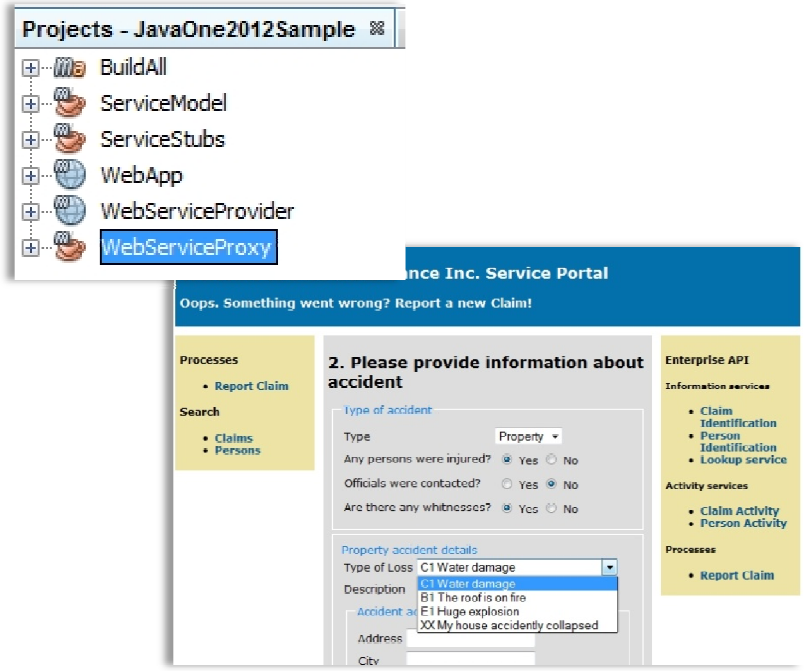


 | |

Demo Application – SOA Playground

NetBeans, Glassfish, JEE6, BeanValidation, CDI, JAX-WS, JSF2

- Service Provider
  - ServiceModel
  - ServiceStubs
  - WebServiceProvider
- Service consumer
  - WebServiceProxy
  - WebApp



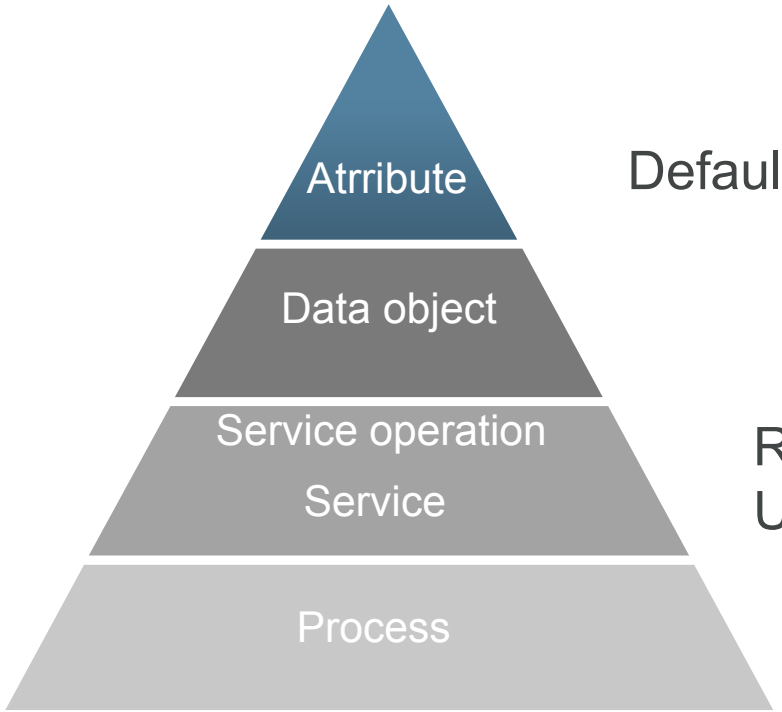
 | |

Summary

JavaOne ORACLE

Summary

Bean Validation with the Common Data Model in SOA



Simple validation: @NotNull, @Size

Default and Custom validation groups

Semantic validation:
Custom Constraint and Validator

Resource-intensive validation:
Unique group, @PersonUniqueConstraint

Process specific constrains:
@validate Object Graph validation

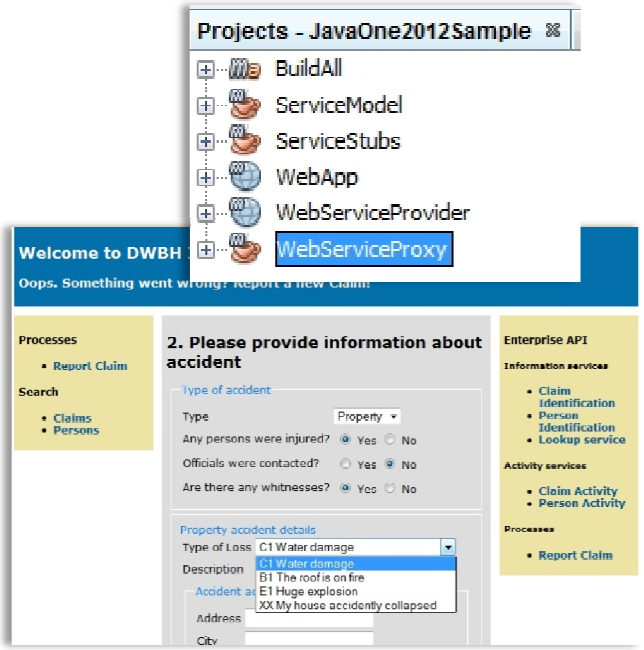
 | |

JavaOne ORACLE

Conclusion

- **Start with business service design**
- **Continue with a common data model**
- **Use bean validation to define business constrains**

- **Java helps to keep your SOA simple!**



 | |

References

Bean Validation 1.0 (stable)

- Java Comunity Process - JSR 303: Bean Validation
  http://jcp.org/en/jsr/detail?id=303
- JSR 303 Bean Validation 1.0 Final Release
  http://download.oracle.com/otndocs/jcp/bean_validation-1.0-fr-oth-JSpec/
- Hibernate Validator 4.x is the reference implementation
  http://www.hibernate.org/subprojects/validator.html

 | |

References

Bean Validation 1.1 (work in progress)

- Bean Validation 1.1 specification website
  http://beanvalidation.org/
- Bean Validation specification 1.1.0.Alpha1 (early draft 1)
  http://beanvalidation.org/1.1/spec/

- What's new in 1.1?
  Openness, Dependency injection, Method validation ...
  http://beanvalidation.org/1.1/spec/#whatsnew

Questions?

mailto:donatas.valys@oracle.com
http://donatas.nicequestion.com
@donatasnq

 | |

JavaOne ORACLE

Thank you for attending my session!

mailto:donatas.valys@oracle.com
http://donatas.nicequestion.com
@donatasnq


MAKE THE
FUTURE
JAVA
JavaOne
ORACLE


JavaOne™