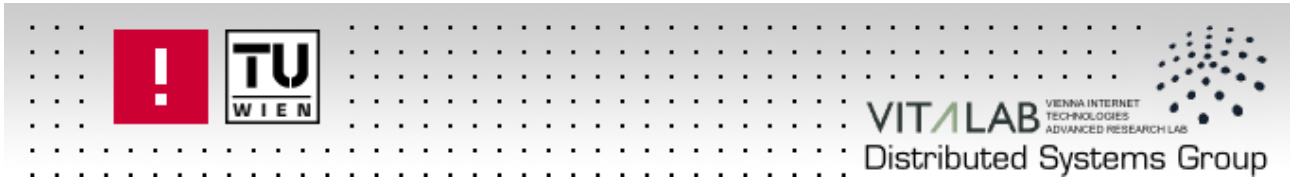


Assignment 1

VU Entwurfsmethoden für Verteilte Systeme

SS2010

ShinyRestFramework

0927232, Josef Dabernig, e0927232@student.tuwien.ac.at

Group evs060

Team members:

- 0325394, Clemens Helm, clemens.helm@gmail.com
- 0425028, Martin Kirchner, martin.kirchner@gmail.com
- 0460615, Andreas Juch, e0460615@student.tuwien.ac.at
- 0927232, Josef Dabernig, e0927232@student.tuwien.ac.at
- 9726426, Bernhard Löwenstein, b.loewenstein@gmx.at

Table of Contents

Introduction

ShinyRestFramework is a java-based framework that allows you to generate RESTful web services.

Web services are created by sub-classing the abstract **RestService** class. Every restful webservice provides CRUD & search functionality for an entity class. The entity class and and advanced settings are configured through **annotations**.

The **RestServer** class allows to publish restful web services.

Quickstart

The Ant build file containts the following targets:

- **build**: Compile all classes and prepare to run the application.
- **clean**: Undo all changes during build.
- **run**: Starts the demo application with Rack, Item & Placement services on http://localhost:8777/
- **test**: Starts the demo application & runs junit tests on the services.

Configuration

There are 3 key-elements to ShinyRestFramework configuration:

- evs.rest.core.**RestService**
- evs.rest.core.**RestServiceConfig**
- evs.rest.core.**annotations**.*

RestService

In order to using ShinyRestFramework, the user first defines restful services by extending the abstract RestService class.

Speaking of *Remoting Patterns*, the RestService represents the **Invoker** which receives dispatched requests from the Server Request Handler, reads the request, demarshalles entities and interprets and executes the operation.

RestServiceConfig

RestService configuration tasks are encapsulated by its abstract superclass RestServiceConfig. On service creation (within the constructor), provided annotations are processed through reflection.

Annotations

Custom RestServices are configured through annotations provided by the ShinyRestFramework. The following configuration are available:

- **RestEntity**
  - Entity class a given RestService represents

- required
- example: `@RestEntity(Rack.class)`

- **RestId**
  - the class of the id column, of the RestEntity of a RestService
  - optional, defaults to Long.class
  - example: `@RestId(Long.class)`
- **RestPath**
  - a custom path specification for a RestService
  - optional, a default path-name will be generated from the service's class name. eg.: Class Rack would be "racks"
  - example: `@RestPath("rack")`
- **RestAcceptedFormats**
  - supported RestFormat formats of a given RestService
  - optional, defaults to JSON and XML
  - example: `@RestAcceptedFormats({RestFormat.JSON, RestFormat.XML})`
- **RestSearchIndexedFields**
  - a list of full-text-searchable columns of the RestEntity of a RestService
  - optional, otherwise no search will be available
  - `example: @RestSearchIndexedFields({"name", "description"})`
- **RestSearchPath**
  - custom path specification for a RestService full-text-search
  - only evaluated, if RestSearchIndexedFields are configured
  - optional, defaults to “search”
  - example: `@RestSearchPath("search")`

Convention over Configuration

ShinyRestFramework implements the Convention over Configuration paradigm. Most configuration settings are optional because RestServiceConfig is able to fall back onto default values. This enables a minimal RestService confuration, as shown in *evs.rest.demo.PlacementService*:

```
@RestEntity(Placement.class)
public class PlacementService extends RestService {
    public PlacementService() throws Exception {
          super();
    }
    private static final long serialVersionUID = 1L;
}
```

On the other hand, advanced users may apply optional configuration settings in order to tailoring the RestService according to their specific needs, as shown in *evs.rest.demo.RackService*:

```
@RestEntity(Rack.class)
@RestId(Long.class)
@RestPath("rack")
@RestAcceptedFormats({RestFormat.JSON, RestFormat.XML})
@RestSearchIndexedFields({"name", "description"})
@RestSearchPath("search")
public class RackService extends RestService {
    public RackService() throws Exception {
          super();
    }
    private static final long serialVersionUID = 1L;
}
```

Publishing

Restful webservices through ShinyRestFramework are published using the RestServer.

Speaking of *Remoting Patterns*, the RestServer represents the **Server Request Handler** which handles all the client communication and dispatches to the RestService Invokers. The RestServer facades a jetty-server which handles the http communication. The RestServices, one adds to the RestServer will be registered as HttpServlets within the jetty-server using their path configuration then and appropriate requests therefore automatically get dispatched to the according RestService.

RestServer

The RestServer class allows to use pre-configured RestServices as shown in *evs.rest.test.TestRestService.prepareClass()*:

```
RestServer provider = new RestServer();

provider.addInterceptor(new BasicInterceptor());

rackService = new RackService();
provider.addService(rackService);

itemService = new ItemService();
provider.addService(itemService);

placementService = new PlacementService();
provider.addService(placementService);

provider.setServerPath("/");

provider.start();
```

The following operations are available for server setup:

- **addService(RestService service)**
  - registeres a RestService for
- **setServerPath(String path)**
  - sets the server path. by default, RestServer will use *RestConst.DEFAULT_SERVICE_PATH*
- **setServerPort(int port)**
  - sets the server port. by default, RestServer will use *RestConst.DEFAULT_SERVER_PORT*
- **setPersistance(RestPersistence persistance)**

    - sets the persistence unit, which the services of this RestServer will use
- **addInterceptor(RestInterceptor interceptor)**
    - registers interceptors for the services

When server setup is finished, the start() method is used to start the server, register all services and publish them.

Approaches

There are many different approaches to realizing such a framework. The following options where considered while planning :

- **embedded-jetty**
    - ShinyRestFramework uses jetty's embedded mode in order to dynamically start a webserver and publish rest services. This allows to dynamically calculate service urls from the services entity classes name and therefore supports the implementation of Convention over Configuration.
- **web.xml**
    - a web.xml file would define the services and their url-mappings. This static approach would enable combining ShinyRestFramework with different web servers like Tomcat. On the other hand side, the static mapping somewhat limits the frameworks paradigm of Convention over Configuration, as it would force the user to specify urls.
- **root-servlet**
    - another option would be a “hybrid” approach of both allowing the user to use web.xml in order to specifying the server url and combining ShinyRestFramework with different web servers like Tomcat. The root-servlet would then forward all requests to the framework and by this way enable dynamic linking of service paths.

For simplicity, ShinyRestFramework sticks to the embedded-jetty variant while the most flexibility seems to be possible using the web.xml configured root-servlet approach.

Operation

ShinyRestFramework Operations are handled by the RestService class.

Operation Types

These are RESTful CRUD + search:

- **Post**
    - handles the REST POST operation
    - *usage: POST http://SERVICE_URI*
    - creates a new object in the database
    - responses
        - *HttpStatus.OK_200* on success. the saved object will be returned as encoded entity in the same format as the request was by it's mimeType definition

        - *HttpStatus.UNSUPPORTED_MEDIA_TYPE_415* if request mimeType can't be mapped to a RestMarshaller instance or this particular service doesn't support it
        - *HttpStatus.BAD_REQUEST_400* for validation errors. the response will textually describe all validation errors found
        - *HttpStatus.INTERNAL_SERVER_ERROR_500* for any other error on service or database level. there won't be any details on the error in the response for security reasons
- **Get**
    - handles the REST GET operation
    - *usage: GET http://SERVICE_URI/{id}*
    - retrieves an existing object by its id from the database
    - responses
        - *HttpStatus.OK_200* on success. the retrieved object will be returned as encoded entity in the same format as the request was by it's mimeType definition
        - *HttpStatus.NOT_FOUND_404* if the object could not be found
        - *HttpStatus.UNSUPPORTED_MEDIA_TYPE_415* if request mimeType can't be mapped to a @RestMarshaller instance or this particular service doesn't support it
        - *HttpStatus.INTERNAL_SERVER_ERROR_500* for any other error on service or database level. there won't be any details on the error in the response for security reasons
- **Put**
    - handles the REST DELETE operation
    - *usage: PUT http://SERVICE_URI/{id}*
    - updates an existing object within the database.
    - responses
        - *HttpStatus.OK_200* on success. the updated object will be returned as encoded entity in the same format as the request was by it's mimeType definition
        - *HttpStatus.UNSUPPORTED_MEDIA_TYPE_415* if request mimeType can't be mapped to a @RestMarshaller instance or this particular service doesn't support it
        - *HttpStatus.INTERNAL_SERVER_ERROR_500* for any other error on service or database level. there won't be any details on the error in the response for security reasons
- **Delete**
    - handles the REST DELETE operation
    - *usage: PUT http://SERVICE_URI/{id}*
    - deletes an existing object from the database
    - responses
        - *HttpStatus.OK_200* on success.

        - *HttpStatus.NOT_FOUND_404* if the object could not be found
        - *HttpStatus.INTERNAL_SERVER_ERROR_500* for any other error on service or database level. there won't be any details on the error in the response for security reasons
- **Search**
    - handles the search operation
    - *usage: GET http://SERVICE_URI/SEARCH_PATH?text=SEARCH_TEXT&FIELDS=FIELDA&FIELDS=FIELDB*
        - *SEARCH_PATH* according to RestSearchPath
        - *FIELDS* are optional, they default to the RestSearchIndexedFields setting
    - searches for the given search text within the specified fields
    - responses
        - *HttpStatus.OK_200* on success. a list of found objects will be returned as encoded entity in the same format as the request was by it's mimeType definition
        - *HttpStatus.UNSUPPORTED_MEDIA_TYPE_415* if request mimeType can't be mapped to a @RestMarshaller instance or this particular service doesn't support it
        - *HttpStatus.INTERNAL_SERVER_ERROR_500* for any other error on service or database level. there won't be any details on the error in the response for security reasons

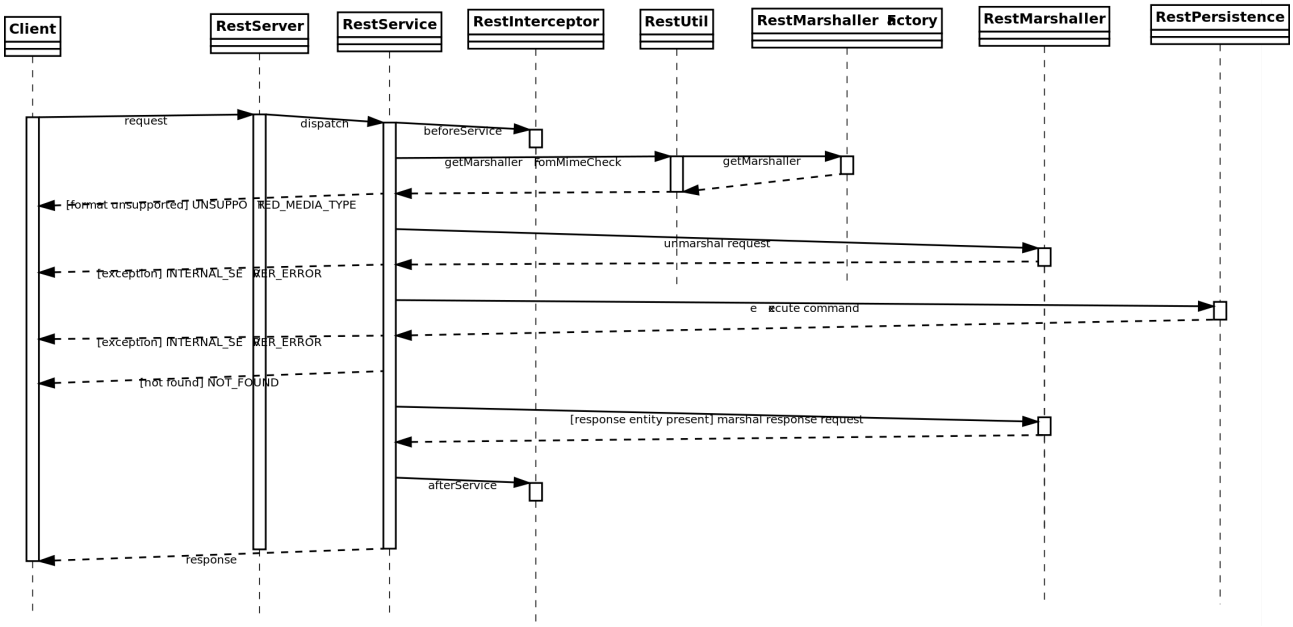
Operation Workflow

Generally said, for any of those operations, the RestService...

1. ... accesses it's RestServiceConfig configuration settings for example in order to determine the service's entity class
2. ... processes the requests **mime type**, compares it with the supported RestFormats of the Service and on success retrieves the according RestMarshaller instance
3. ... processes **parameters** and/or **deserializes** the request using the marshaller
4. ... executes the **command** on database level using the RestPersistance, the service is configured for
5. ... reacts on **exceptions** by mapping them to HttpStatus response codes
6. ... finally **marshalles** valid **response** entities, if applicable

The following sequence diagram illustrates the operation workflow:

ShinyRestFramwork Sequence Diagram



Components

Persistance

ShinyRestFramework encapsulated the database layer by the **RestPersistance** interface. It provides the following the CRUD + search operations, similar to the RestFul CRUD operations, but with different naming:

- **create** - create a new object within the database
- **read** - retrieves an existing object from the database
- **update** - updates an existing object within the database
- **delete** - deletes an existing object from the database
- **search** – full-text-search

HibernatePersistance is the default RestPersistance impmentation of ShinyRestFramework. For the basic CRUD functionality it sticks to the JPA standard while the search functionality is based on Hibernate Criteria.

Marshalling

A RestService may provides a one or many **RestFormats** which are configured using the **RestAcceptedFormats** annotation. Every RestFormat is mapped to the according mime-type as implemented in evs.rest.core.marshal.RestFormat.fromMimeType().

The interface **RestMarshaller** abstracts different marshallers which convert from text streams to objects. ShinyRestFramework currently implements the two impmentations **XStreamJSONMarshaller** and **XStreamXMLMarshaller**.

The **RestMarshallerFactory** provides RestMarshaller instances for given RestFormats.

Additionally the **RestUtil** wrappes this function by determining the mime type for a given request first.

Notes / TODOs

- The Put operation doesn't only update parameters, included within the request but the whole object. This is due to the approach not passing separate parameters but an entire object, this means the framework wouldn't be able to decide if an unset/null value means no-change or a set-to-null operation.
- The marshallers don't implement the Absolute Object Reference pattern. This would require additional work. At the moment, full object graphs are transferred over the net.
- The combination of HibernatePersistence and the xstream marshallers causes distortion of the serialized xml/json-output. HibernatePersistanceBag instances wrap for example the rack.placements listing.

Searching

The RestService implements a basic search functionality which may be configured using the **RestSearchIndexedFields** and **RestSearchPath** annotations.

Any search invokation will be translated into a search request on database level querying for a specific full-text search term for a given class while specifying a list of fields that should be included for the given query.

The HibernatePersistance implementation uses *org.hibernate.Criteria* in order to build a *Restriction* which includes all specified fields as follows:

```
Criteria crit = session.createCriteria(clazz);
Junction junction = Restrictions.disjunction();
for (String field : fields) {
    junction.add(Restrictions.like(field, text));
}
crit.add(junction);
List<T> results = crit.list();
```

Notes / TODOs

- Search currently doesn't implement pagination or limit.

Transactions

Transactions are handled within the RestPersistance database layer.

The HibernatePersistance implementation uses the EntityManager transactions:

```
em.getTransaction().begin();
em.persist(object);
em.getTransaction().commit();
```

Validation & Exceptions

The ShinyRestFramework currently imlements validation only on database level in the default HibernatePersistance implementation.

The following exceptions as speaking of **Remoting Errors** are defined:

- persistance
  - RestPersistanceException - wraps any database exception
  - RestPersistanceValidationException – validation errors
- marshaller
  - RestMarshallerException – wraps any marshalling exception
  - UnknownRestFormatException – a mimeType to RestFormat conversion failed

HibernatePersistance uses Hibernate Validator in order to support javax.validation annotations.

*The Rack demo implements a custom PlacementConstraint with a PlacementValidator which ensures that the total of placement.amount * item.size must not exceed rack.place.*

Validation is performed before save & update operations on the database level. Validation errors will be thrown as a RestPersistanceValidationException. The RestService implements a private method *evs.rest.core.RestService.validationErrors()* which reacts on the exception and renders an according HTTP response.

Notes/TODOs

- Advanced exception handling / better separation of concerns could be achieved by throwing custom RestServiceExceptions which are converted to HttpStatus responses using interceptors / filters(?).

Interceptors / Logging

The RestInterceptor interface specifies possible interceptors which can be applied to a RestService using RestServer.addInterceptor(). There are interceptor methods before and after every service and database action. For example the post interception points are:

```
public void beforeServicePost(HttpServletRequest req, HttpServletResponse resp);
public void afterServicePost(HttpServletRequest req, HttpServletResponse resp);
public void beforeDBCreate(Object object);
public void afterDBCreate(Object object);
```

ShinyRestFramework comes with a reference implementation of the BasicInterceptor which writes each action with it's timestamp to the log, using log4j.

Notes/TODOs

- The interceptor is stateless and therefore can't align before* with after* methods in order to reliablly calculate database response times for example.
- The execution of the interceptor-chain should be inverse for before and after operations.

Tests

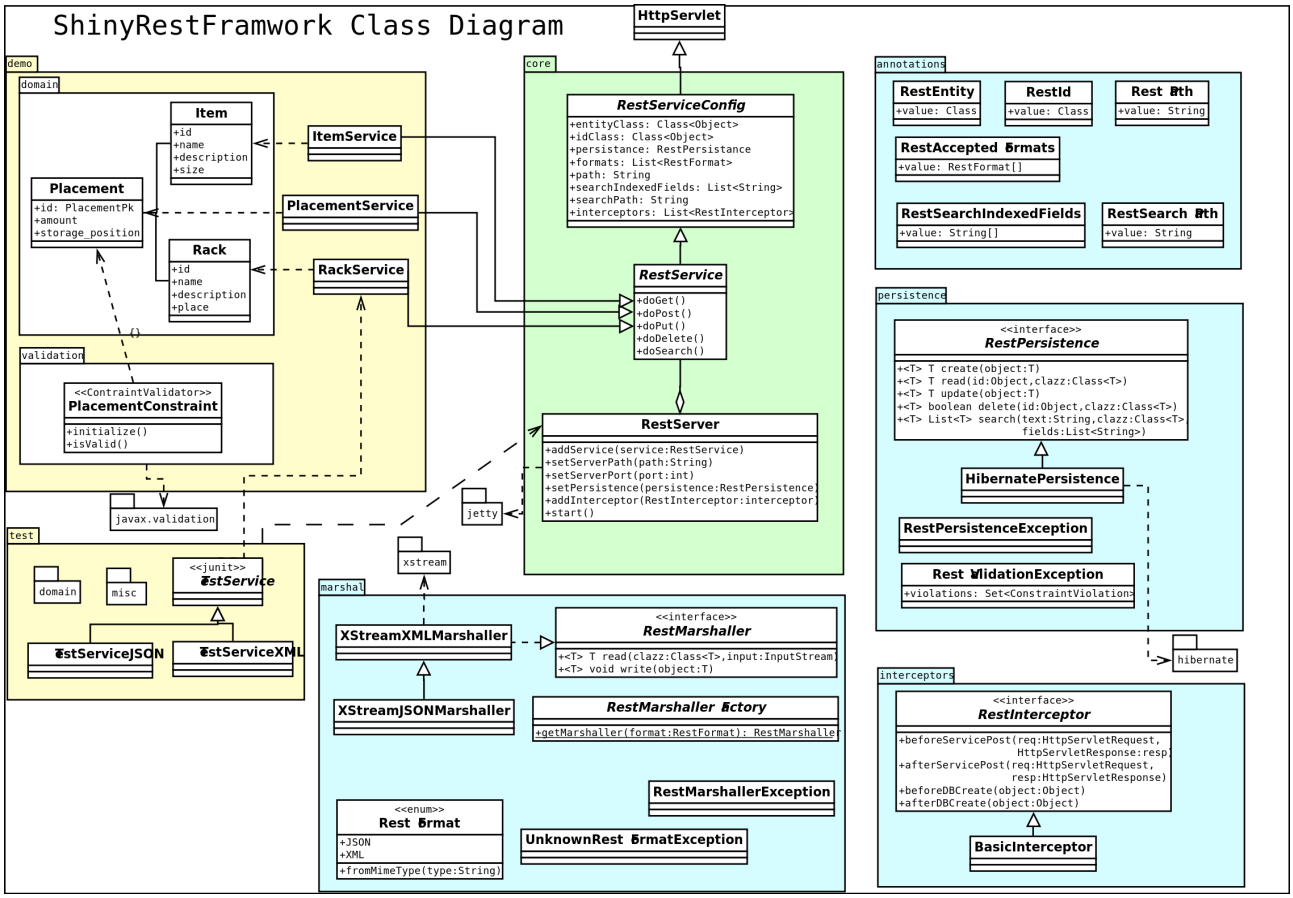
The abstract junit 4 test class **TestRestService** first starts the **RackService**, **ItemService** and **PlacementService** demo service instances using RestServer.

There are 2 implementations of TestRestService for the 2 currently supported RestFormats JSON and XML. In order to run the tests, execute "ant run" from the command line.

Test-cases for post, get, put, delete and search cover common usage scenarias of RestService by Apache **HttpComponents** in order to access the Restful services via Http and the

ShinyRestFramework marshallers in order to serialize objects.

Class Diagram



(sorry, the font messed up on export from dia)

Remarks

- Only CRUD + search functionality implemented, there is no index service (“Parts list” as in lecture slides)
- No **Interface Description** pattern implementation, WSDL could be generated using reflection.
- No **Lifecycle Management Pattern** implemented.
- The RestId annotation allows specifying the class of the id field. Still the implementation of id type conversion is missing (see getIdFromPath @ RestUtil)
- some browsers (safari, some opera versions) don't support PUT & DELETE requests. We could enable a generic GET/POST requests which specifiy an optional method=PUT parameter
- some TODO remarks in code ;)