**SOAP VS REST:**

**There are many differences between SOAP and REST web services. The important 10 differences between SOAP and REST are given below:**

|  |  |  |
| --- | --- | --- |
| **No.** | **SOAP** | **REST** |
| **1)** | **SOAP is a protocol.** | **REST is an architectural style.** |
| **2)** | **SOAP stands for Simple Object Access Protocol.** | **REST stands for Representational State Transfer.** |
| **3)** | **SOAP can't use REST because it is a protocol.** | **REST can use SOAP web services because it is a concept and can use any protocol like HTTP, SOAP.** |
| **4)** | **SOAP uses services interfaces to expose the business logic.** | **REST uses URI to expose business logic.** |
| **5)** | **JAX-WS is the java API for SOAP web services.** | **JAX-RS is the java API for RESTful web services.** |
| **6)** | **SOAP defines standards to be strictly followed.** | **REST does not define too much standards like SOAP.** |
| **7)** | **SOAP requires more bandwidth and resource than REST.** | **REST requires less bandwidth and resource than SOAP.** |
| **8)** | **SOAP defines its own security.** | **RESTful web services inherits security measures from the underlying transport.** |
| **9)** | **SOAP permits XML data format only.** | **REST permits different data format such as Plain text, HTML, XML, JSON etc.** |
| **10)** | **SOAP is less preferred than REST.** | **REST more preferred than SOAP.** |

REST is almost always going to be faster. The main advantage of SOAP is that it provides a mechanism for services to describe themselves to clients, and to advertise their existence.

REST is much more lightweight and can be implemented using almost any tool, leading to lower bandwidth and shorter learning curve. However, the clients have to know what to send and what to expect.

In general, When you're publishing an API to the outside world that is either complex or likely to change, SOAP will be more useful. Other than that, REST is usually the better option.

**Why SOAP?**

Here are a few reasons you may want to use SOAP.

**WS-Security**

While SOAP supports SSL (just like REST) it also supports WS-Security which adds some enterprise security features. Supports identity through intermediaries, not just point to point (SSL). It also provides a standard implementation of data integrity and data privacy. Calling it “Enterprise” isn’t to say it’s more secure, it simply supports some security tools that typical internet services have no need for, in fact they are really only needed in a few “enterprise” scenarios.

**WS-AtomicTransaction**

Need ACID Transactions over a service, you’re going to need SOAP. While REST supports transactions, it isn’t as comprehensive and isn’t ACID compliant. Fortunately ACID transactions almost never make sense over the internet. REST is limited by HTTP itself which can’t provide two-phase commit across distributed transactional resources, but SOAP can. Internet apps generally don’t need this level of transactional reliability, enterprise apps sometimes do.

**WS-ReliableMessaging**

Rest doesn’t have a standard messaging system and expects clients to deal with communication failures by retrying. SOAP has successful/retry logic built in and provides end-to-end reliability even through SOAP intermediaries.

In Summary, SOAP is clearly useful, and important. For instance, if I was writing an iPhone application to interface with my bank I would definitely need to use SOAP. All three features above are required for banking transactions. For example, if I was transferring money from one account to the other, I would need to be certain that it completed. Retrying it could be catastrophic if it succeeds the first time, but the response failed.

**RESTful Web Service - JAX-RS Annotations - Contents:**

|  |  |
| --- | --- |
| **Annotation** | **Package Detail/Import statement** |
| @GET | import javax.ws.rs.GET; |
| @Produces | import javax.ws.rs.Produces; |
| @HeaderParam | Import javax.ws.rs.HeaderParam |
| @Path | import javax.ws.rs.Path; |
| @PathParam | import javax.ws.rs.PathParam; |
| @QueryParam | import javax.ws.rs.QueryParam; |
| @POST | import javax.ws.rs.POST; |
| @Consumes | import javax.ws.rs.Consumes; |
| @FormParam | import javax.ws.rs.FormParam; |
| @PUT | import javax.ws.rs.PUT; |
| @DELETE | import javax.ws.rs.DELETE; |

Examples:

**@GET**

Annotate your Get request methods with @GET.

|  |  |
| --- | --- |
|  | @GET  public String getHTML() {    ...  } |

### @Produces

@Produces annotation specifies the type of output this method (or web service) will produce.

|  |  |
| --- | --- |
|  | @GET  @Produces("application/xml")  public Contact getXML() {    ...  } |
|  | @GET  @Produces("application/json")  public Contact getJSON() {    ...  } |

### @HeaderParam

### Maps the header parameters of http request .

@GET

@Produces(MediaType.***TEXT\_HTML***)

**public** String getIt(

@HeaderParam("firstName") String firstName,

@HeaderParam("lastName") String lastName,

@HeaderParam("dataSource") String dataSource) **throws** Exception {

String msg = "hi there";

**return** msg + "firstName:" + firstName + "lastName:" + lastName + "datasource:" + dataSource;

### }

### @Path

@Path annotation specify the URL path on which this method will be invoked.

|  |  |
| --- | --- |
|  | @GET  @Produces("application/xml")  @Path("xml/{firstName}")  public Contact getXML() {    ...  } |

### @PathParam

We can bind REST-style URL parameters to method arguments using @PathParam annotation as shown below.

|  |  |
| --- | --- |
|  | @GET  @Produces("application/xml")  @Path("xml/{firstName}")  Public Contact getXML(@PathParam("firstName") String firstName) {    Contact contact = contactService.findByFirstName(firstName);    return contact;  } |
|  | @GET  @Produces("application/json")  @Path("json/{firstName}")  public Contact getJSON(@PathParam("firstName") String firstName) {    Contact contact = contactService.findByFirstName(firstName);    return contact;  } |

### @QueryParam

Request parameters in query string can be accessed using @QueryParam annotation as shown below.

@GET

@Path("data/{dataSource}")

@Produces(MediaType.***APPLICATION\_JSON***)

**public** Response getCustomerDetails(

@HeaderParam("firstName") String firstName,

@HeaderParam("lastName") String lastName,

@PathParam("dataSource") String dataSource,

@QueryParam("role") String role,

@Context **final** HttpServletResponse servletResponse) {

**if** (role.equals("admin")) {

Customer customer = **new** Customer();

customer.setFirstName(firstName);

customer.setLastName(lastName);

servletResponse.setHeader("dataSource", dataSource);

**return** Response.*status*(Status.***OK***).entity(customer).build();

} **else** {

**return** Response.*status*(Status.***FORBIDDEN***).entity("").build();

}

}

### @POST

Annotate POST request methods with @POST.

|  |  |
| --- | --- |
|  | @POST  @Consumes("application/json")  @Produces("application/json")  Public Response create(Contact contact) {  ...  } |

### @Consumes

The @Consumes annotation is used to specify the MIME media types a REST resource can consume.

@GET

@Path("data/{dataSource}")

@Consumes(MediaType.***APPLICATION\_JSON***)

@Produces(MediaType.***TEXT\_HTML***)

**public** Response getCustomerDetails(

Customer customer,

@PathParam("dataSource") String dataSource,

@QueryParam("role") String role,

@Context **final** HttpServletResponse servletResponse)

**throws** Exception {

String msg="";

**if** (role.equals("admin")) {

msg = testBean.getDetails(customer.getFirstName(), customer.getLastName(), "WEB");

**return** Response.*status*(Status.***OK***).entity(msg).build();

} **else** {

**return** Response.*status*(Status.***FORBIDDEN***).entity(msg).build();

}

}

### @FormParam

The REST resources will usually consume XML/JSON for the complete Entity Bean. Sometimes, you may want to read parameters sent in POST requests directly and you can do that using @FormParam annotation. GET Request query parameters can be accessed using @QueryParam annotation.

@POST

@Path("mdm/form")

@Produces("text/plain")

**public** String getFormDetails(

@FormParam("firstName")String firstName,

@FormParam("lastName")String lastName,

@FormParam("dataSource")String dataSource,

@FormParam("selection")String selectionType) **throws** Exception {

String msg="hi there";

**if**(selectionType.equals(ConstantsUtility.***SELECTION\_TYPE\_FETCH***)){

msg=testBean.getDetails(firstName, lastName, dataSource);

}

**else** **if**(selectionType.equals(ConstantsUtility.***SELECTION\_TYPE\_PUT***)){

msg=testBean.putCustomerDetails(firstName, lastName,dataSource);

}

**return** msg;

}

|  |  |
| --- | --- |
|  |  |

### @PUT

Annotate PUT request methods with @PUT.

@PUT

@Path("/{dataSource}")

@Produces(MediaType.***APPLICATION\_JSON***)

**public** Response putCustomerDetails(

@HeaderParam("firstName") String firstName,

@HeaderParam("lastName") String lastName,

@PathParam("dataSource") String dataSource,

@Context **final** HttpServletResponse servletResponse) {

Customer customer = **new** Customer();

customer.setFirstName(firstName);

customer.setLastName(lastName);

servletResponse.setHeader("dataSource", dataSource);

**return** Response.*status*(Status.***OK***).entity(customer).build();

}

|  |  |
| --- | --- |
|  |  |

### @DELETE

Annotate DELETE request methods with @DELETE.

|  |  |
| --- | --- |
|  | @DELETE  @Produces("application/json")  @Path("{contactId}")  public Response delete(@PathParam("contactId") int contactId) {  ...  } |