

JSPIDER  
BASAVANGUDI  
BANGALORE

WEB SERVICES  
JERSEY CLIENT API  
( CONSUMER)

| Praveen D

## Table of Contents

JAX-RS Client API.....	2
1. Obtaining the Client Instance.....	2
2. Setting the Client Target.....	3
3. Create a request based on the target .....	4
4. Invoke the request.....	5
5. Process the Response .....	5

*JSpiders*

Training & Development Center

## JAX-RS Client API

- The JAX-RS Client API provides a high-level API for accessing any REST resources, not just JAX-RS services.
- The Client API is defined in the "javax.ws.rs.client.\*" package
- The following steps are needed to access a REST resource using the Client API.
  1. Obtain an instance of the javax.ws.rs.client.Client interface
  2. Configure the Client instance with a target
  3. Create a request based on the target
  4. Invoke the request
  5. Process the Response

### 1. Obtaining the Client Instance

- To utilize the client API it is first necessary to build an instance of a Client using one of the static ClientBuilder factory methods.

```
//First Approach  
Client client = ClientBuilder.newClient();
```

OR

```
//Second Approach  
ClientBuilder builder = ClientBuilder.newBuilder();  
Client client = builder.build();
```

- ClientBuilder is an Abstract Class part of JAX-RS API which has "N" number of Abstract Methods is the Main entry point to the client API.
- "org.glassfish.jersey.client.JerseyClientBuilder" part of Jersey framework is a concrete class which extends ClientBuilder

```
public class JerseyClientBuilder extends ClientBuilder
```

- newBuilder() is a Static method of ClientBuilder create a new instance of ClientBuilder
- build() is a non-static method but abstract method, return type is Object of type "Client"

```
public abstract Client build();
```

- Hence the implementation of this method is provided by the JAX-RS implementation provider such as Jersey, CXF, etc.,
- newClient() is a Static method of ClientBuilder, it returns new Client instance

- `newClient()` method which internally invokes `"newBuilder().build()"`

```
public static Client newClient() {
    return newBuilder().build();
}
```

- `"javax.ws.rs.client.Client"` is an Interface part of JAX-RS API. Implementation class provided by the JAX-RS implementation provider such as Jersey, CXF, RestEasy, etc.,
- `"org.glassfish.jersey.client.JerseyClient"` part of Jersey framework is a concrete class which implements `"javax.ws.rs.client.Client"`

```
public class JerseyClient implements javax.ws.rs.client.Client
```

- We can go for 2nd approach, in a slightly more advanced scenarios (i.e. if you want to interact with web services using https protocol), where `ClientBuilder` can be used to configure additional client instance properties, such as a SSL transport settings, etc.,
- For http we can go for First Approach
- Use the `close()` method to close Client instances after its job is done
- Client instances are heavyweight objects. For performance reasons, limit the number of Client instances, as the initialization and destruction of these instances may be expensive

## 2. Setting the Client Target

- The target of a client, the REST resource at a particular URI, is represented by an instance of the `javax.ws.rs.client.WebTarget` interface.
- we can obtain a `WebTarget` instance by invoking `target()` method and passing in the URI of the target REST resource.

```
Client client = ClientBuilder.newClient();
WebTarget myResource=client.target("http://www.example.com/webapi");
```

- `"javax.ws.rs.client.WebTarget"` is an Interface part of JAX-RS API
- `"org.glassfish.jersey.client.JerseyWebTarget"` part of Jersey framework is a concrete class which implements `"javax.ws.rs.client.WebTarget"`
- For complex REST resources, it may be beneficial to create several instances of `WebTarget`. For example,

```
Client client = ClientBuilder.newClient();
WebTarget base = client.target("http://www.example.com/webapi");
```

```
//http://www.example.com/webapi/read
WebTarget read = base.path("read");
```

```
//http://www.example.com/webapi/write
WebTarget write = base.path("write");
```

- In the above example, a base target is used to construct several other targets that represent different services provided by a REST resource.
- The `WebTarget.path()` method creates a new `WebTarget` instance by appending the current target URI with the path that was passed in

### 3. Create a request based on the target

- After setting and applying any configuration options to the target, we use one of the `WebTarget.request()` methods to begin creating the request.
- This is usually accomplished by passing to `WebTarget.request()` the accepted media response type for the request
  - either as a string of the MIME type or
  - using one of the constants in `javax.ws.rs.core.MediaType`
- The `WebTarget.request()` method returns an instance of `javax.ws.rs.client.Invocation.Builder`, a helper object that provides methods for preparing the client request.
- "Invocation" is an Interface and "Builder" is a inner-interface of "Invocation"
- `Invocation.Builder` has a lot of methods that allow us to
  - set different types of request headers
  - various `acceptXXX()` methods for content negotiation
  - `cookie()` methods allow us to set HTTP cookies you want to return to the server
- **NOTE:-**
  - `WebTarget` has additional `request()` methods whose parameters take one or more String or `MediaType` parameters
  - These parameters are media types you want to include in an Accept header
  - It makes the code more readable if we use `Invocation.Builder.accept()` method. However it's a matter of personal preference
  - Example:-
 

```
Client client = ClientBuilder.newClient();
String url = "http://www.example.com/webapi/read";

client.target(url).request(MediaType.TEXT_HTML);
```

**OR**

```
client.target(url).request().accept(MediaType.TEXT_HTML);
```

## 4. Invoke the request

- After setting the media type, invoke the request by calling one of the methods of the `Invocation.Builder` instance that corresponds to the type of HTTP request the target REST resource expects.
- These methods are:
  1. `head()`
  2. `trace()`
  3. `put()`
  4. `post()`
  5. `delete()`
  6. `options()`
  7. `get()`
- The return type should correspond to the entity returned by the target REST resource.
- If the target REST resource is expecting an HTTP POST request, call the `Invocation.Builder.post` method

```
StoreOrder order = new StoreOrder(...);

Client client = ClientBuilder.newClient();
WebTarget myResource
    = client.target("http://www.example.com/webapi/write");
TrackingNumber trackingNumber
    = myResource
        .request(MediaType.APPLICATION_XML)
        .post(Entity.xml(order), TrackingNumber.class);
```

- In the above example, the return type is a custom class and is retrieved by setting the type in the `Invocation.Builder.post(Entity<?> entity, Class<T> responseType)` method as a parameter
- If the return type is a collection, use `javax.ws.rs.core.GenericType<T>` as the response type parameter, where T is the collection type

```
List<StoreOrder> orders
    = client
        .target("http://www.example.com/webapi/")
        .path("allOrders")
        .request(MediaType.APPLICATION_XML)
        .get(new GenericType<List<StoreOrder>>() {});
```

- In the above example, methods are chained together in the Client API to simplify how requests are configured and invoked

## 5. Process the Response

- Certain versions of request methods (get(), head(), etc.) returns "javax.ws.rs.core.Response" object
- This is the same Response class that is used on the server side. This gives us more fine-grained control of the HTTP response on the client side.
- For Example

```
import javax.ws.rs.core.Response;

Response response
    = client
        .target("http://www.ecommerce.com/customers/123")
        .accept("application/json")
        .get();

try
{
    if (response.getStatus() == 200) {
        Customer customer = response.readEntity(Customer.class);
    }
}finally{
    response.close();
}
```

- In the above example, we invoke an HTTP GET to obtain a Response object. We check that the status is OK and if so, extract a Customer object from the returned JSON document by invoking Response.readEntity().
- The readEntity() method matches up the requested Java type and the response content with an appropriate MessageBodyReader
- Responses object can be used at Producer/Server side as well as at Consumer/Client code
  - @Producer side, it's used to send the Response with additional/metadata information
  - @Consumer side, it's used to get the Response to get more information about the response
- Responses object has lot of static methods and all of them returns object of "ResponseBuilder". These methods should be used for sending the Response as part of "Producer / Server" code
- Responses object also has lot of Getter methods (ex:- getCookies(), getHeaders(), getLastModified(), etc.) which helps us to get additional/metadata information from Request
- **NOTE:-** readEntity() fetches the "actual response" and other Getter methods provides "additional information" about the response
- readEntity() method should be invoked only once unless we buffer the response with the bufferEntity() method.

- For example:

```
Response response
    = client
        .target("http://www.ecommerce.com/customers/123")
        .accept("application/json")
        .get();

try
{
    if (response.getStatus() == 200) {
        response.bufferEntity();
        Customer customer = response.readEntity(Customer.class);
        Map rawJson = response.readEntity(Map.class);
    }
} finally {
    response.close();
}
```

- In the above example, the call to `bufferEntity()` allows us to extract the HTTP response content into different Java types, the first type being a `Customer` and the second a `java.util.Map` that represents raw JSON data
- If we didn't buffer the entity, the second `readEntity()` call would result in an `IllegalStateException`
- **Closing Response Object:-**
  - Once we're done with Response object we MUST close() it
  - Response objects reference open socket streams. If we do not close them, we're leaking system resources
  - While most JAX-RS implementations implement a `finalize()` method for Response, it is not a good idea to rely on the garbage collector to clean up
  - The default behaviour of the RESTEasy JAX-RS implementation actually only lets us to have one open Response per Client instance. This forces you to write responsible client code.