**4. Internals of request() or Building and Invoking Requests:**

4.1 Invoking request() using Invocation.Builder style

|-i) Working with Headres, Cookies, Content Negotiation Headers (Accepting Content-Type)

|-ii) Working with Entity and List<Entity> using GenericType<T> and Response

|-iii) Reading Response body data from multiple times

|-iv) Working with Form

4.2 Invoking request() using Invocation style

|-Purpose of Invoking request() using Invocation style

|-Use case (Getting stock price or details)

4.3 Document to Build the Client

**4. Internals of request() or Building and Invoking Requests:**

Once we have a WebTarget that represents the exact URI we want to invoke on, we can begin building

and invoking HTTP requests through one of its request() methods:

public interface WebTarget extends Configurable<WebTarget> {

public abstract URI getUri();

public abstract UriBuilder getUriBuilder();

// So when we call getUriBuilder() on WebTarget obj we will get UriBuilder so we can access all the methods that are there in the UriBuilder to build complete URI.

...

public Invocation.Builder request();

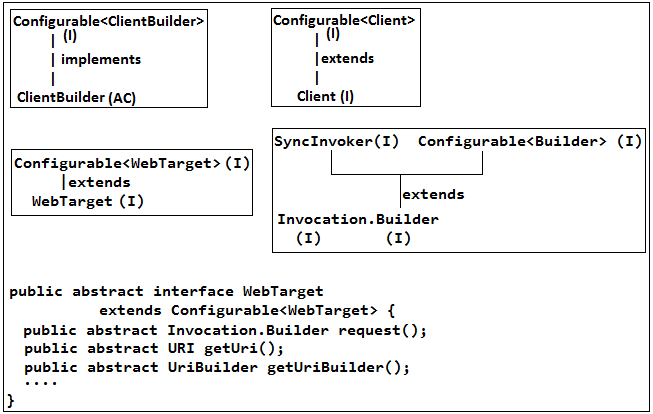
public Invocation.Builder request(String... acceptedResponseTypes);

public Invocation.Builder request(MediaType... acceptedResponseTypes);

}

**4.1 Invoking request() using Invocation.Builder style:**

**i) Working with Headres, Cookies, Content Negotiation Headers (Accepting Content-Type):**



The Invocation.Builder interface hierarchy is a bit convoluted, so now we need to understand how to build requests using examples and code fragments:

package javax.ws.rs.client;

public interface Invocation {

...

public interface Builder extends SyncInvoker, Configurable<Builder> {

...

public Builder accept(String... types);

public Builder accept(MediaType... types);

public Builder acceptLanguage(Locale... locales);

public Builder acceptLanguage(String... locales);

public Builder acceptEncoding(String... encodings);

public Builder cookie(Cookie cookie);

public Builder cookie(String name, String value);

public Builder cacheControl(CacheControl cacheControl);

public Builder header(String name, Object value);

public Builder headers(MultivaluedMap<String, Object> headers);

}

}

Invocation.Builder has a bunch of methods that allow you to set different types of request headers. The various acceptXXX() methods are for content negotiation. The cookie() methods allow you to set HTTP cookies we want to return to the server. And then there are the more generic header() and headers() methods that cover the more esoteric HTTP headers and any custom ones our application might have.

Content Negotiation Headers (Accepting Content-Type):

Content Negotiation Headers means we can ask the sever to give xml/json by using accept(MediaType.XML) or accept(MediaType.JSON) if the server is capable of send the response both XML/JSON which is called as Content Negotiation.

**ii) Working with Entity and List<Entity> using GenericType<T> and Response:**

After setting the headers the request requires, we can then invoke a specific HTTP method to get back a response from the server. GET requests have two flavors:

public abstract interface SyncInvoker {

Response get();

<T> T get(Class<T> responseType);

<T> T get(GenericType<T> responseType);

<T> T put(Entity<?> entity, Class<T> responseType);

<T> T put(Entity<?> entity, GenericType<T> responseType);

<T> T post(Entity<?> entity, Class<T> responseType);

<T> T post(Entity<?> entity, GenericType<T> responseType);

Response post(Entity<?> entity);

Response put(Entity<?> entity);

// Similarly for Delete, options etc as well methods are there

......

}

package javax.ws.rs.client;

public interface Invocation {

....

public interface Builder extends SyncInvoker, Configurable<Builder> {

....

}

}

As Builder is extends from SyncInvoker all the methods of SyncInvoker will be available in Builder.

The first two generic get() methods will convert successful HTTP requests to specific Java types. Let’s look at these in action:

Customer customer = client.target("http://commerce.com/customers/123")

.accept("application/json")

.get(Customer.class);

List<Customer> customer = client.target("http://commerce.com/customers")

.accept("application/xml")

.get(new GenericType<List<Customer>>() {});

In the first request we want JSON from the server, so we set the Accept header with the accept() method. We want the JAX-RS client to grab this JSON from the server and convert it to a Customer Java type using one of the registered MessageBodyReader components.

The second request is a little more complicated. We have a special MessageBodyReader that knows how to convert XML into List<Customer>. The reader is very sensitive to the generic type of the Java object, so it uses the javax.ws.rs.core.GenericType class to obtain information about the type.

GenericType is a sneaky trick that bypasses Java type erasure to obtain generic type information at runtime. To use it, you create an anonymous inner class that implements GenericType and fill in the Java generic type you want to pass information about to the template parameter. I know this is a little weird (unnatural), but there’s no other way around the Java type system.

Tip WebTarget has additional request() methods whose parameters take one or more String or MediaType parameters. These parameters are media types we want to include in an Accept header. I think it makes the code more readable if you use the Invocation.Builder.accept() method instead. But this generally is a matter of personal preference.

There’s also a get() method that returns a Response object. This is the same Response class that is used on the server side. This gives you more fine-grained control of the HTTP response on the client side.

Here’s an example:

import javax.ws.rs.core.Response;

Response response = client.target("http://commerce.com/customers/123")

.accept("application/json")

.get();

try {

if (response.getStatus() == 200) {

Customer customer = response.readEntity(Customer.class);

}

} finally {

response.close();

}

In this 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. This method can be invoked only once unless we buffer the response with the bufferEntity() method.

**iii) Reading response body data from multiple times:**

try {

if (response.getStatus() == 200) {

Customer customer = response.readEntity(Customer.class);

// Automatically InputStream will closes hence response will not contains data once we red

}

} finally {

response.close();

}

It we red the data from the response it is not possible to read once again if read already bcz once we call response.readEntity() the response will be taken from the from response body InputStream once we red the response body from the InputStream it will automatically closes, so if we try to read once again it will throws IllegalStateException will come that is the reason if we wanted to read the data from the response body multiple times then we need take one variable and store in that variable so that we can read multiple times instead of we taking a variable the JAX-RS has provided one more method called as response.bufferEntity(); which will read the data and stores in the buffer so if read once again from the response it will not reads form the response body rather it will reads form the buffer.

Response response = client.target("http://commerce.com/customers/123")

.accept("application/json")

.get();

try {

if (response.getStatus() == 200) {

response.bufferEntity();

// Reading first time

Customer customer = response.readEntity(Customer.class);

// Reading data 2nd time

Map rawJson = response.readEntity(Map.class);

}

} finally {

response.close();

}

In this 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.

Always remember to close() your Response objects. Response objects reference open socket streams. If you do not close them, you are 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 poorly written code. The default behavior of the RESTEasy JAX-RS implementation actually only lets you have one open Response per Client instance. This forces you to write responsible client code.

So far we haven’t discussed PUT and POST requests that are there in SyncInvoker that submit a representation to the server. These types of requests have similar method styles to GET but also specify an entity parameter:

<T> T put(Entity<?> entity, Class<T> responseType);

<T> T put(Entity<?> entity, GenericType<T> responseType);

<T> T post(Entity<?> entity, Class<T> responseType);

<T> T post(Entity<?> entity, GenericType<T> responseType);

Response post(Entity<?> entity);

Response put(Entity<?> entity);

The Entity class encapsulates the Java object we want to send with the POST or GET request:

package javax.ws.rs.client;

public final class Entity<T> {

public Variant getVariant() {}

public MediaType getMediaType() {

public String getEncoding() {

public Locale getLanguage() {

public T getEntity() {

public Annotation[] getAnnotations() { }

...

}

The Entity class does not have a public constructor. You instead have to invoke one of the static convenience methods to instantiate one:

package javax.ws.rs.client;

import javax.ws.rs.core.Form;

public final class Entity<T> {

public static <T> Entity<T> xml(final T entity) { }

public static <T> Entity<T> json(final T entity) { }

public static Entity<Form> form(final Form form) { }

...

}

The xml() method takes a Java object as a parameter. It sets the MediaType to application/xml. The json() method acts similarly, except with JSON. The form() method deals with form parameters and application/x-www-form-urlencoded, and requires using the Form type. There’s a few other helper methods, but for brevity we won’t cover them here.

Let’s look at two different examples that use the POST create pattern to create two different customer resources on the server. One will use JSON, while the other will use form parameters:

Customer customer = new Customer("Bill", "Burke");

Response response = client.target("http://commerce.com/customers")

.request().

.post(Entity.json(customer));

If at all we wanted access the MediaType or language the use below code bcz we cannot create obj for Entity<T> bcz it doesn't have constructor hence we need to call one of the static method on Entity which will returns Entity obj on that we can call the non-static methods.

MediaType type=Entity.json(new Customer()).getMediaType();

Locale locale=Entity.json(new Customer()).getLanguage();

response.close();

Here we pass in an Entity instance to the post() method using the Entity.json() method. This method will automatically set the Content-Type header to application/json.

To submit form parameters, we must use the Form class:

**iv) Working with Form:**

package javax.ws.rs.core;

public class Form {

public Form() { }

public Form(final String parameterName, final String parameterValue) { }

public Form(final MultivaluedMap<String, String> store) { }

public Form param(final String name, final String value) { }

public MultivaluedMap<String, String> asMap() { }

}

This class represents application/x-www-form-urlencoded in a request. Here’s an example of it in use:

Form form = new Form().param("first", "Bill")

.param("last", "Burke);

response = client.target("http://commerce.com/customers")

.request()

.post(Entity.form(form));

**4.2 Invoking request() using Invocation style:**

The previous examples are how you’re going to typically interact with the Client API. JAX-RS has an additional invocation model that is slightly different. You can create full Invocation objects that represent the entire HTTP request without invoking it. There’s a few additional methods on Invocation.Builder that help you do this:

public interface Invocation {

...

public interface Builder extends SyncInvoker, Configurable<Builder> {

Invocation build(String method);

Invocation build(String method, Entity<?> entity);

Invocation buildGet();

Invocation buildDelete();

Invocation buildPost(Entity<?> entity);

Invocation buildPut(Entity<?> entity);

...

}

}

The buildXXX() methods fill in the HTTP method you want to use and finish up building the request by returning an Invocation instance. You can then execute the HTTP request by calling one of the invoke() methods:

package javax.ws.rs.client;

public interface Invocation {

public Response invoke();

public <T> T invoke(Class<T> responseType);

public <T> T invoke(GenericType<T> responseType);

...

}

**Purpose of Invoking request() using Invocation style:**

So what is the use of this invocation style? For one, the same Invocation object can be used for multiple requests. Just prebuild your Invocation instances and reuse them as needed. Also, since invoke() is a generic method, we could queue up Invocation instances or use them with the execute pattern. Let’s see an example:

Response response = client.target("http://commerce.com/orders/report")

.queryParam("start", "now - 5 minutes")

.queryParam("end", "now")

.request()

.accept("application/json")

.get();

If we send req the request obj will be created if we call get() the req will be sent to the srver and gets the response, if again we send the req by creating for loop for 5-times then for each and every time the request obj will be created and sends the req to the server to get the response but creating every time req obj eventhough the req we are sending containing same req data, same headers and same cookies will not makes sence and leads to performnace issues bcz creating un-necessarily the obj's in the JVM that's where we nedd to go for Invoking request() using Invocation style.

**Use case (Getting stock price or details):**

We need to send same req containing same req data, same headers and cookies to get the stock price of stock info bcz stocks are going to change with in fraction of seconds so we need to display the stock details latest for each and every fraction of seconds that means we need to send req's containing same data multiple time to get the stock info. In this case if we use above procedure to send the req and to get the stock info then memory issues will come that' where to re-use same req obj to send multiple times we need to use Invoking request() using Invocation style which is given below.

Invocation generateReport = client.target("http://stock.com/report")

.queryParam("start", "now - 5 minutes")

.queryParam("end", "now")

.request()

.accept("application/json")

.buildGet();

// Till here invocation obj will be created/builded but req will not send when we call invoke() then only req will be send to get the response that means we can send same req obj multiple times to get the response which improves application performance.

while (true) {

Report report = generateReport.invoke(Report.class);

renderReport(report);

Thread.sleep(300000);

}

The example code prebuilds a GET Invocation that will fetch a JSON report summary of orders made in the last five minutes. We then loop every five minutes and re-execute the invocation. Sure, this example is a little bit contrived (forced, strained), but I think you get the idea.

Similarly we can write for POST req as follows

Response response = target.request()

.accept(MediaType.APPLICATION\_XML)

.post(Entity.xml(subscriber));

With the above code if we send multiple req's performnace issues will come that's where we need to use below code

Invocation invocation = target.request()

.accept(MediaType.APPLICATION\_XML)

.buildPost(Entity.xml(subscriber));

Response response = invocation.invoke();

if (response.getStatus() == 200) {

receipt = response.readEntity(Receipt.class);

}

**4.3 Document to Build the Client:**

The Resource provider will gives a document to develop the client what input need to send, for example if we need to send xml then they will provide structure of xml and details about elements which required and which are optional etc based on this structure the client developers need to write xsd and need validate against xsd and then need to develop the Binding classes by passing xsd as input and then send the req by registering MessageBody Readers and Writers.