JAX-RS (JSR 311)

POJOs with Annotations

JAX-RS

Makes the developer focus on URLs, HTTP methods and Media Types.

Lets code…

@Path

```
@Path("location")
public class LocationResouce {
}
```

@GET, @PUT, @POST

```
@Path("location")
public class LocationResouce {

    @GET
    @Produces("application/json")
public String getLocation() {
        return "I'm in Bucharest;
    }
}
```

@PathParam, @QueryParam

```
@Path("location")
public class LocationResouce {


    @GET
    @Path("{user}")
    @Produces("application/xml")
    public String getLocation(@PathParam("user") String user) {
        return "<location>” + user + " is in Bucharest</location>";
    }
}
```

Server API

By default, request processing on the server works in a synchronous mode, which means that each request is processed in a single HTTP thread. When request processing is finished, the thread is returned to the thread pool. This is not very significant when resource method execution takes a short time and the number of concurrent connections is relatively not very high. In asynchronous mode the thread is returned to the thread pool before request processing is completed. Request processing is then continued in another thread, called a Worker. The released I/O thread can be used to accept new incoming requests. In many cases, just a few threads can handle lots of requests simultaneously, so the number of threads needed to handle incoming requests can be reduced significantly. By using a lot fewer threads we both save memory and improve performance (by reducing thread context switching) and we gain more resistance to [cascading failures]
(http://en.wikipedia.org/wiki/Cascading_failure)

Async Server API

```
import javax.ws.rs.container.AsyncResponse;
import javax.ws.rs.container.Suspended;
import java.util.concurrent.Executor;

@Path("/")
public class Resource {

@Inject
private Executor executor;

@GET
public void asyncGet(@Suspended final AsyncResponse asyncResponse) {
      executor.execute(() -> {
              String result = service.veryExpensiveOperation();
              asyncResponse.resume(result);
           });

    }
 }
```

@Context, UriInfo and UriBuilder

```
@Path("location")
public class LocationResouce {
    @Context UriInfo uriInfo;
    . . .
    @POST
    @Consumes("application/x-www-form-urlencoded")
    public Response saveLocation(@FormParam("user") String user) {
        locations.put(user, new Location("59.3", "18"));
        UriBuilder uriBuilder = uriInfo.getAbsolutePathBuilder();
        URI userUri = uriBuilder.path(user).build();
        return Response.created(userUri).build();
    }
}
```

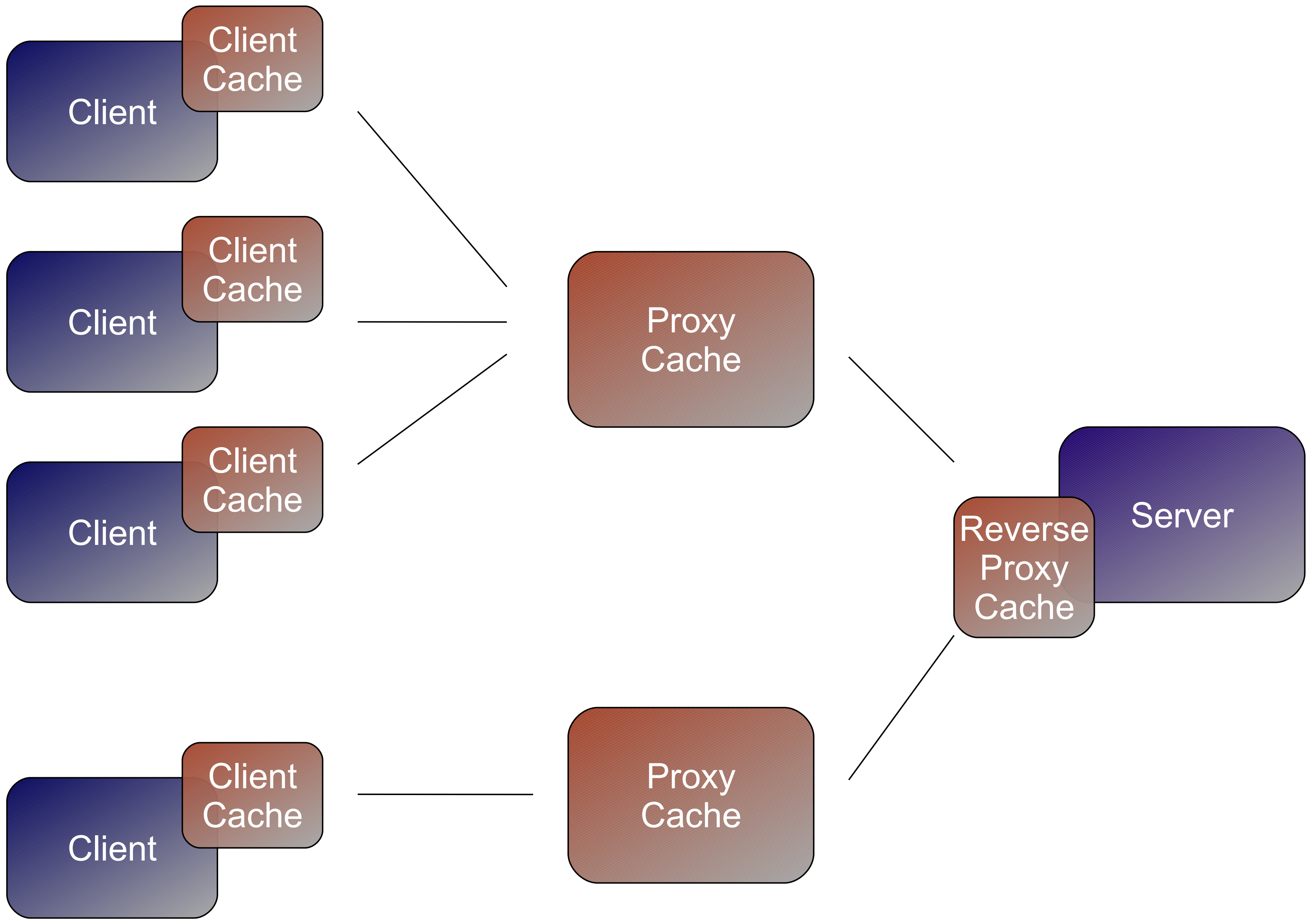
Cache Control

```
@GET
@Path("{user}")
@Produces({"application/xml", "application/json"})
public Response getLocation(@PathParam("user") String user) {
    Location l = locations.get(user);
    CacheControl cc = new CacheControl();
    cc.setMaxAge(500);
    Response.ResponseBuilder builder = Response.ok(l);
    builder.cacheControl(cc);
    return builder.build();
}
```

Statelessness

- Better scaling
  - No session storage on server
  - Any server in a cluster can handle any request
  - All the result pages can safely be cached
- More reliable
  - A client can safely re-send a request
- Better resource reuse
  - The resources can safely be linked to

Caching


Client
Client Cache
Client
Client Cache
Client
Client Cache
Client
Client Cache
Proxy Cache
Proxy Cache
Reverse Proxy Cache
Server

“The best requests are those that not even reach me.”

- Anonymous overloaded Server

50 requests/second
=
3000 requests/minute

setting max-age=60 (seconds)
can then save you 2999 requests

Caching

- GET can be cached while POST can't (safely)
- Specify max-age
  - Use max-age cache control header
- Split data according to freshness requirement
- Support conditional GET
  - Get if modified since
  - E-tags

Jax-RS Client

```
public Response test() throws Exception{
   Client client = ClientBuilder.newBuilder().build();
   WebTarget target = client.target("https://api.github.com/search/users?q=abhckzz");
   Invocation.Builder reqBuilder = target.request();
   AsyncInvoker asyncInvoker = reqBuilder.async();
   Future<Response> futureResp = asyncInvoker.get();
   Response response = futureResp.get(); //blocks until client responds or times out
   String responseBody = response.readEntity(String.class);
   return Response.status(response.getStatus()).entity(responseBody).build();
}
```

```
Customer customer = client.target("http://commerce.com/customers/123")
                      .accept("application/json")
                      .get(Customer.class);
```

Async Client

```
Client client = ClientBuilder.newClient();

Future<Response> future1 = client.target("http://example.com/service")
                                 .request()
                                 .async().get();
```