Most of the information resources will return a 200 when a GET request is issued. Obviously, PURLs override that behavior to return 302, 303, 307, 404, etc. The interesting resource-oriented tidbit is revealed when we inspect the PURL-oriented implementation of the `resStorage.getResource()` method:

```
INKFRequest req = context.createSubRequest("active:purl-storage-query-purl");
req.addArgument("uri", uri);
IURRepresentation res = context.issueSubRequest(req);
return context.transrept(res, IAspectXDA.class);
```

In essence, we are issuing a logical request through the `active:purl-storage-query-purl` URI with an argument of `ffcpl:/purl/employee/briansletten`. Ignore the unusual URI scheme; it is simply used to represent an internal request in NetKernel. We do not know what code is actually going to be invoked to retrieve the PURL in the requested form, nor do we actually care. In a resource-oriented environment, we simply are saying, "The thing that responds to this URI will generate a response for me." We are now free to get things going quickly by serving static files to clients of the module while we design and build something like a Hibernate-based mapping to our relational database. We can make this transition by rewriting what responds to the `active:purl-storage-query-purl` URI. The client code never needs to know the difference. If we change the PURL resolution away from a local persistence layer to a remote fetch, the client code can still not care. These are the benefits we have discussed in the larger notion of resource-oriented Enterprise computing made concrete in a powerful software environment.

Not only are our layers loosely coupled like this, but we get the benefit of an idempotent, stateless request in this environment as well. The earlier code snippet that fetches the PURL definition gets flattened internally to an asynchronously scheduled request to the URI `active:purl-storage-query-purl+uri@ffcpl:/purl/employee/briansletten`. As we discussed earlier, this becomes a compound hash key representing the result of querying our persistence layer for the generated result. Even though we know nothing about the code that gets invoked, NetKernel is able to cache the result nonetheless. This is the architectural memoization that I mentioned before. The actual process is slightly more nuanced, but in spirit, this is what is going on. If someone else tries to resolve the same PURL either internally or through the HTTP RESTful interface, we could pull the result from the cache. Though this may not impress anyone who has built caching into their web pages, it is actually a far more compelling result when you dig deeper. Any potential URI request is cacheable in this way, whether we are reading files in from disk, fetching them via HTTP, transforming an XML document through an XSLT file, or calculating pi to 10,000 digits. Each of these invocations is done through a logical, stateless, asynchronous result, and each has the potential to be cached. This resource-oriented architectural style gives us software that scales, is efficient, is cacheable, and works through uniform, logical interfaces. This results in substantially less brittle, more flexible architectures that scale, just like the Web and for the same reasons.

Download at WoweBook.Com