Taking Control of WCF

Patrick Roeper & Jay Hill

|  |  |  |
| --- | --- | --- |
| **Your Presenters**  Patrick Roeper   * patrick.roeper@gmail.com * github.com/broccliman   Jay Hill   * jay@codequota.com * twitter: @jittery * blog: codequota.com * feedback: speakerrate.com/jayhill | **ELMAH**  Error Logging Modules And Handlers  Powerful error-logging facility for ASP.NET  We can leverage this to provide a view into our WCF exceptions  code.google.com/p/elmah | **Object-Oriented Principles and Design Patterns**   * DRY: don’t repeat yourself * Single responsibility * Open-closed principle   These are the key principles that we touched on, designing extensibility, maintainability, and testability into our WCF application |
| **Bypassing the Service API**  We identify lots of duplication and multiplicity of responsibilities when implementing a service in a .svc file.  We modularize operations into their own classes and redirect WCF to use these. | | |
| **Problems**   * Generated client code is clunky and is a duplication of server-side contracts; easy to get out of sync * WCF services have many responsibilities; .svc file makes it hard to modularize operation handling * Client endpoints scale in parallel to server endpoints | **Solutions**   * Say “goodbye” to * Move service definition and data contract components to own assembly * Package client wrapper API for invoking service * Modularize operations as separate RequestHandler classes * Hook into WCF pipeline to bypass .svc file and reroute to RequestHandlers * Isolate infrastructural, cross-cutting concerns from business logic * Create a gateway or “portal” with a single operation to handle any request, either instead of or in addition to a suite of services | |
| |  |  | | --- | --- | | **Reigning in the Endpoints**  As services grow in number, there is a cost associated with managing those services on the client side. In many cases, the creation of all of these services is also an unnecessary effort.  A single gateway, or “Portal,” allows us to achieve a highly maintainable, distributed architecture for our internal service consumers, either instead of or in addition to “standard” service endpoints, for which there may be barriers to rapid versioning.  The Portal also greatly simplifies service consumption for the client, and grows along with other RequestHandler-based services with no code changes required. |  | | | |
| **Applied OOP : Using the Decorator Pattern to compose functionality around WCF operation invocation**   * Each piece of infrastructural functionality is implemented as an IOperationInvoker * Each piece accepts an “inner” IOperationInvoker that is invoked within its own implementation of Invoke( ) * By using a compositional technique rather than inheritance, we can apply these in arbitrary combinations according to configuration, attributes, runtime conditions, etc.   ErrorHandlingInvoker : IOperationInvoker  Invoke()  {  try  {      }  catch { … }  }  AuditInvoker: IOperationInvoker  Invoke()  {  AuditRequest();    }  WcfInvoker : IOperationInvoker  Invoke()   * Typically, the framework-provided invoker would be at the innermost position – last to be actually invoked * When using the RequestHandler<> technique, we never invoke this, as we are instead resolving a handler object that will process the request | | **WCF Extensibility**  While WCF provides many extensibility points throughout the stack, our solutions focused on these in particular.   * IOperationBehavior – allows us to attach run-time behaviors to a service operation * IServiceBehavior – allows us to attach run-time behaviors to an entire service * IOperationInvoker – object that will actually call the method in our service implementation   Operation and service behaviors can be applied through configuration or implemented as Attributes that can be applied to a service operation or the entire service, respectively.  We created a *single* Attribute, ServiceExtensionsAttribute, which can be applied to an operation or to an entire service. Our implementation of the IServiceBehavior interface consists only of applying our IOperationBehavior implementation to all operations in the service for all configured endpoints (except metadata exchange endpoints).  Implementing IOperationBehavior is what gives us access to the IOperationInvoker. This is the key piece that allows us to divert processing from the .svc file to our RequestHandler objects, and add infrastructural functionality by wrapping this invocation with decorators. |
| Today’s slide deck and demo code will be made available on github.com/broccliman. | | |