MESSAGING WITH WEB SERVICES

Message-based design is enabled in ServiceStack by capturing the Services Request Query into a Request DTO that's completely de-coupled from any one implementation. You can think of making a ServiceStack request as a Smalltalk runtime method dispatch at a Macro scale, where the ServiceStack host is the Receiver, the HTTP Verb is the selector and Request DTO is the message.

It doesn't matter on which of the endpoints the Request is sent to as the request can be populated with any combination of PathInfo, QueryString and Request Body. After the Request binding, the request travels through all user-defined filters and pre-processors for inspection where it can optionally be handled before it is able to reach the service implementation. Once the request reaches the service it invokes the best matching selector, by default it will look for a method with the same name as a HTTP Verb, if it doesn't exist it falls back to using a catch-all 'Any' method that can be used to handle the request on any endpoint or route in any format. Even when inside the Service implementation it is able to further delegate the request to an alternative service or easily proxy the request to a remote sharded instance if it needs to.

Conceptually in ServiceStack you're just sending a message to a ServiceStack instance, the client is not concerned with what ultimately handles it, only that a response is returned for reply requests or that the request is successfully accepted for oneway messages. In a RPC API you're conceptually invoking a remote method which has the request tightly coupled to its remote implementation method signature.

Benefits of adopting a Message-based design

There are many natural benefits gained when adopting a message-based design they offer better resilience, flexibility, versionability than their RPC cousins. An example of one of the benefits possible is when you send a request to its one-way endpoint, if the ServiceStack instance has a MQ Host configured, the request is automatically deferred to the configured MQ Broker and processed in the background. So even if

the ServiceStack Host goes down none of the pending messages are lost and are automatically processed the next time the Host starts up. This is the type of behavior that is enabled for free in ServiceStack. When no MQ host is enabled the request is just processed normally, i.e. synchronously by a HTTP web worker.

Most of the benefits of message-based designs are gained over time as you're developing and evolving your existing services and adding support for more clients. **One immediate benefit is being able to provide an end-to-end typed API without the use of code-gen**. This is impossible to achieve without a message-based design which ensures the essence of your Service Contract is captured in re-usable DTOs. Being able to share your server DTOs you defined your web services with on the client completely by-pass the normal development workflow required in re-generating your clients proxies from your services interim WSDL/XSD schemas.

Typed, Native SDK's provide maximum end-user value

Typed clients are the under-pinnings for **Native SDK's which provide the most value to end-users of your service** as they reduce the most of the burden required in order to consume your API. This approach is popular for companies that really, really want you to use their APIs, i.e. where their businesses success depends on its popular use. This is the preferred approach taken by Amazon EC2, Google App Engine, Azure, Facebook, Ebay, Stripe, Braintree, etc.

More importantly, message-based designs encourage the design of coarse-grained and more re-usable services. By contrast RPC method signatures are generally designed to serve a single-purpose, i.e. Rather than adding more RPC methods for every client requirement (which introduces a new external endpoint each time), message-based designs instead encourages enhancing existing services with extra functionality since they can be added without friction. This additionally has the benefit of providing instant utility to existing clients already consuming existing services, since they can easily access the extra features without introducing a new code-path to call a new external endpoint.

Ideal for SOA

This is an especially important approach to take whenever implementing service-intensive systems like SOA platforms, as services routinely end up out-living and serving more clients that the original client that consume them, so it's important not to have your service APIs driven by adhoc client-specific requirements. It's more useful to think about designing APIs from the system's perspective with the goal of

exposing the underlying systems capabilities in a generically re-usable API. This is the main reason why I now only ever adopt message-based designs for all my services endpoints as Coarse-grained APIs naturally encourage the design of more re-usable and feature-rich APIs.