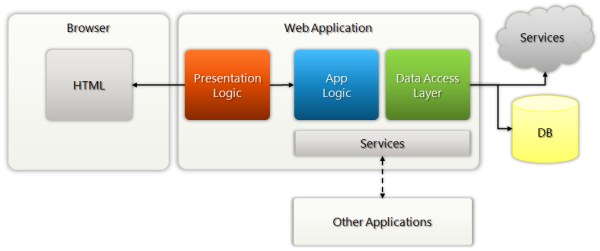
# .NET RIA Services

http://www.nikhilk.net/NET-RIA-Services-Vision-Architecture.aspx

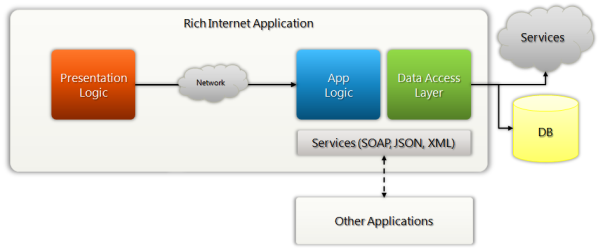
Microsoft .NET RIA Services simplifies the traditional n-tier application pattern by bringing together the ASP.NET and Silverlight platforms. RIA Services provides a pattern to write application logic that runs on the mid-tier and controls access to data for queries, changes and custom operations. It also provides end-to-end support for common tasks such as data validation, authentication and roles by integrating with Silverlight components on the client and ASP.NET on the mid-tier.

Simply speaking, we wanted to make development as simple as 2-tier apps, *without* sacrificing good n-tier design or compromising on fundamentals.

Lets look at the anatomy of a classic Web application (and most Ajax apps aren't significantly different).



With a RIA, most of the presentation logic moves to the client to improve the UX and to make effective use of local state.The result is additional work for the developer: defining a full-fledged services layer, new types and contracts to share etc. and continually evolving them as the app evolves on either end.



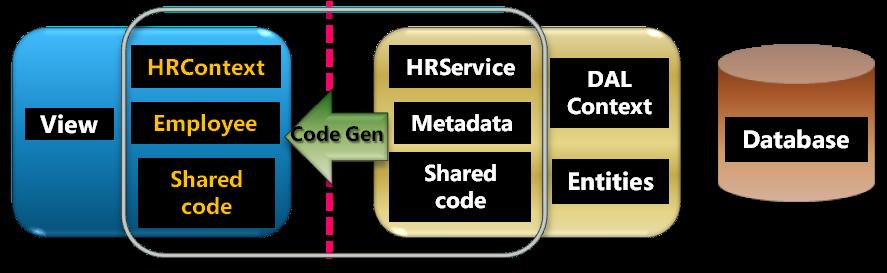
The presentation tier is separated by a network, **but .NET RIA Services steps in to provide the plumbing behind the scenes to preserve a 2-tier feel to the overall codebase.** Peel back a layer, and underneath the covers, you'll see familiar building blocks: the goodness of services, and open/standard protocols (more on that below). They're simply implementation detail, that you don't have to focus on as the high order bit. Instead you get to focus on your application's logic and user interface.

We want to address that trust boundary and point of latency by i**ncorporating those factors(Network, Services, Contracts...) into the programming model and making them natural, and simplifying the associated tooling.**

What this fresh look at RIA lets us do:

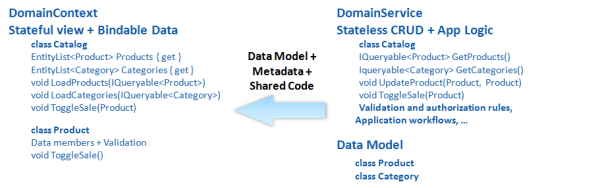
* **Share behavior**, application semantics, metadata, and even implementation rather than just contract between tiers. Its all within the bounds of the same app. Why do things twice?
* **Gear our tooling to address what you're building** - a RIA. There is a lot more we have to do to get to nirvana here. Over time...

**In RIA, some code and logic should be shared**, like operation permission, authentication, data validation (on client side, quick response, on server side, correct data status). This is supported by .NET RIA.



**A Concrete, End-to-End, Application Pattern**

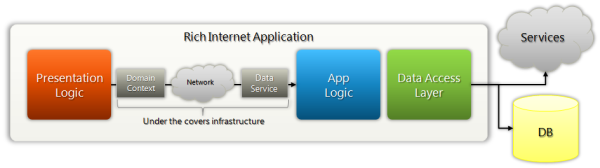
Using the example from my [MIX demo](http://www.nikhilk.net/RIA-Services-MIX09.aspx), I have a simple data model with Product and Category entities.



The first part of our pattern is to write a **DomainService** class. **This represents your application domain, your application logic, or business logic**. It surfaces the set of data a client can see, which might be DAL types, or projection types invented specifically for use by the presentation tier, or a combination thereof. It also surfaces a set of operations: both CRUD-based and custom domain-specific operations. It also specifies rules around the data and the operations such as authorization and validation. Finally it encapsulates any DAL-specific nuances. A DomainService is optimized to be **stateless** - to respond to query requests and to change set processing requests.

The second part of the pattern is what we generate - the client-side data model - in a **DomainContext** class. **This class represents the view of what the client can see. It contains lists of objects, one for each type exposed to the client, and it contains a set of load methods that roughly correspond to the queries exposed on the service**. These load methods can be used to load objects into the corresponding lists. The individual objects and lists track changes and raise change notifications. The DomainContext can extract all the changes, create a change set and submit it to the service for processing and commiting. A DomainContext is optimized for taking advantage of the **stateful** environment and to be a well-behaved citizen in a binding-centric presentation technology such as Silverlight or WPF.

hen you peel a layer, you'll see that the infrastructure is built on familiar building blocks of services and proxies and deriving the qualities of service-based design.



**Summary**

Despite being prescriptive, I think we have a very flexible technology built on building blocks that can be replaced. A small set of examples:

* For example, you could write your own explicit service on top of the application logic or DomainService pattern, and you could create an explicit service reference in your client, if you prefer to be explicit across tiers, rather than using all of the underlying infrastructure.
* You could plug in your own back-end store if the ones we support out-of-the box don't match your needs. For example, you could plug in a rich domain model.
* You could decide how much of the DAL you want to expose or how little, by instead constructing an explicit set of presentation types.