

TEST PDF 5:

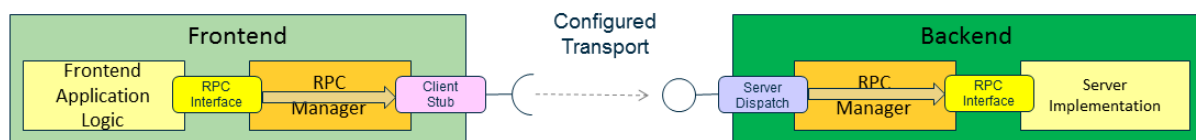
DRAWING PRODUCTION JS

This article discusses RPC communication in iTwin.js.

This article contains test for bold and italics as well test for sub script and super script

Overview

the functionality of an iTwin.js app is typically implemented in separate components that run in different processes, potentially on different machines. These components communicate through interfaces. These interfaces can either be implemented as [Rpc](#) or [Ipc](#). For web applications, iTwin.js uses *RpcInterfaces* or [RPC](#).



The diagram above shows an app frontend requesting operations from some backend. The terms *client* and *server* specify the two *roles* of an *RpcInterface*:

- *client*-- the code that runs on the frontend, and calls methods on an *RpcInterface*.
- *server*-- the code that runs on the backend, and implements the *RpcInterface*.

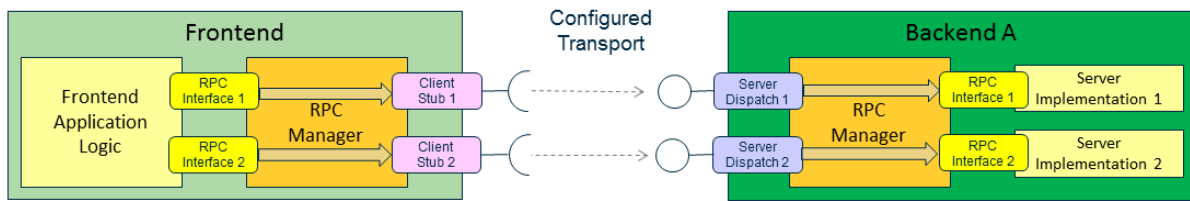
Classes that derive from [RpcInterface](#) define a set of operations implemented by a server, callable from a client.

As shown, client and server work with the [RpcManager](#) to manage the available *RpcInterfaces*. *RpcManager* exposes a client "stub" on the client side that forwards *RPC* requests. On the other end, *RpcManager* uses a server dispatch mechanism to relay the request to the implementation in the server. In between the two is a transport mechanism that marshalls the data passed from the client to the server over an appropriate communications channel. The transport mechanism is encapsulated in a *configuration* that is applied at runtime.

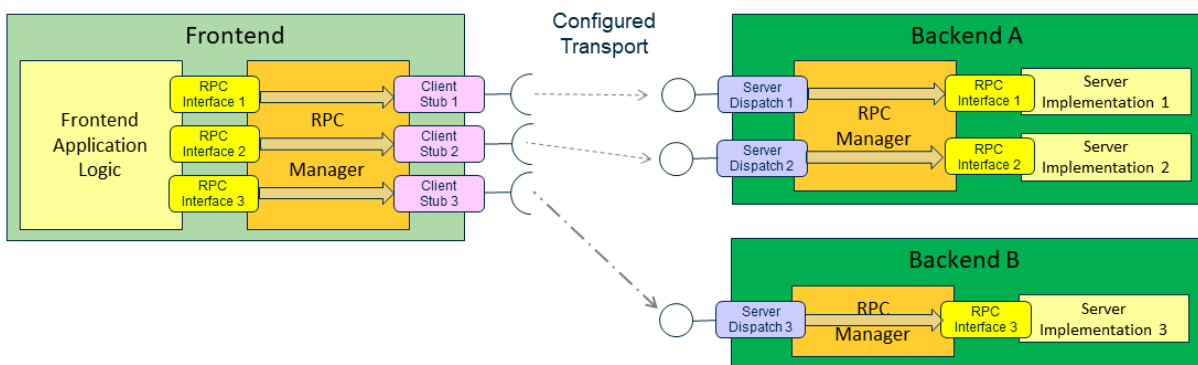
A typical app frontend will use more than one remote component. Likewise, a server can contain and expose more than one component. For example, the app frontend might need two interfaces, Interface 1 and Interface 2. In this example, both are implemented in Backend A.

Test for Sub_{script} and Super^{script} and **bold Sub_{script} and Super^{script}**

And *italics Sub_{script} and Super^{script}* as well as bold and italics ***Sub_{script} and Super^{script}***



An app frontend can just as easily work with multiple backends to obtain the services that it needs. One of the configuration parameters for an `RpcInterface` is the identity of the backend that provides it. For example, suppose that the frontend also needs to use Interface 3, which is served out by Backend B.



The RPC transport configuration that the frontend uses for Backend B can be different from the configuration it uses for Backend A. In fact, that is the common case. If Backend A is the app's own backend and Backend B is a remote service, then the app will use an [RPC configuration](#) that matches its own configuration for A, while it uses a `Web` configuration for B.

As noted above, the client of an RPC interface can be frontend or backend code. That means that backends can call on the services of other backends. In other words, a backend can be a server and a client at the same time. A backend configures the `RpcInterfaces` that it *implements* by calling the `initializeImpl` method on `RpcManager`, and it configures the `RpcInterfaces` that it *consumes* by calling `initializeClient`. For example, suppose Backend B needs the services of Backend C.

