

Jiffybus Technical Overview

Jiffybus is a Java framework that is designed primarily to facilitate the development of n-tier Java applications with rich, interactive, client-server style user interfaces. It was developed in response to an identified business need to deploy Java applets in the enterprise without deviating too far from the basic web application architecture that has become popular over the past few years. It is attractive to application users because it transcends the many shortcomings of HTML as a user interface toolkit; at the same time, it is attractive to IT because it retains the many strengths of the web application model, like low-cost deployment, scalability, and ubiquity.

In the following sections, we will examine the components of the Jiffybus framework in detail.

Fundamental Request/Response: A review of basic web architecture

The basic idea of *request/response* should be familiar to anyone who has developed a web application. This technique is the fundamental architectural notion behind HTTP and, therefore, the Web. The notion is this: a client (sometimes a browser, sometimes a custom application) formulates a **request** of some kind and sends the request to a server (in our case, an HTTP or "Web" server) that is equipped to handle the request. Generally the server examines the contents of the request, looking for things it "understands" in terms of generating data to send back to the client. In the most basic and familiar case, the request consists of a request for a *resource* like an HTML file or JPG image that the user has asked for via the browser. The server takes the request, converts the URL to a local file location, and pipes the data from the file into the network socket that is connected to the browser on the other end.

The format of the request can also indicate that the server is to utilize an external application or server plug-in to generate the response. This technique is what allows data to be dynamically generated on the server, and is the next evolutionary step forward beyond static file serving. The external application can be a Java servlet, an Active Server Page, or a script written in Perl, Python, or PHP. (These are just some typical examples...many technologies exist for generating content dynamically.) However sophisticated they seem, these server-side technologies are, architecturally speaking, minor twists on the basic idea of a server receiving a request from a client and generating an appropriate response.

Clearly, the wide acceptance of the web as an enterprise application platform has been fostered by the simplicity and openness of the architecture. At the core of this architecture is the HTTP protocol itself. (While HTML has been important, it is far less standardized and has become more complex, while HTTP *is* the core standard and has remained fairly simple.) The W3C [RFCs](#) for HTTP describe the protocol in detail. It is worth noting here that the protocol is simple, text-based, and easily extensible to a wide range of uses. In particular, while the vast majority of worldwide use of HTTP is for communicating requests from browsers and responding with HTML, this is by no means

the only use. Both request and response packets can be used to convey arbitrary payloads from the client to the server. This notion is in fact at the core of how Jiffybus works.

Jiffybus as an application of HTTP

It is probably easiest to approach Jiffybus from a standpoint in which we view the framework as enhancements to the basic browser/server web architecture reviewed above.

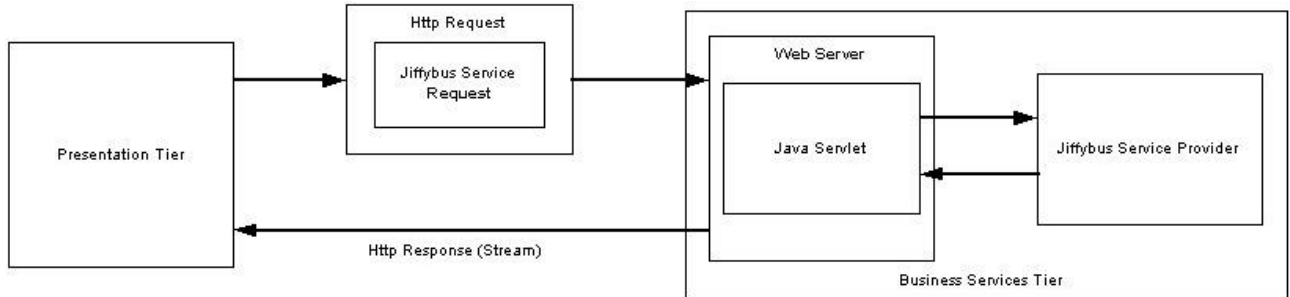
We begin by partitioning application functionality into tiers, which has become a common practice among web developers. The *presentation tier* is the set of functionality that deals directly with the user. The presentation tier consists of the user interface itself, plus any business rules and application logic that functions to give the user a better experience. Most applications built with Jiffybus will use a rich, interactive user interface built with Java (and its Swing UI toolkit) to deliver the presentation tier, rather than HTML. The *business service tier* is the set of functionality that performs all of the key business functions of the application, independent of the user interface. Generally, the business service tier deals with persistence (e.g., making calls to a database to store and retrieve information), data validation, and accessing enterprise services (e.g., email, messaging, legacy systems, etc.) Finally, the *persistence tier* deals with managing the physical storage of information, if the application requires it. Typically the persistence tier will consist of a third-party relational database solution, but could also incorporate Enterprise Java Beans (EJBs).

As we begin to develop the application in these tiers, we find we need a way to communicate between them. Communication between the business service tier and the persistence tier is generally handled by JDBC and other Java enterprise technologies, including application servers that provide high scalability and failover as well as access to particular legacy systems. As Jiffybus is not an application server, it offers little in the way of facilitating the communication between the business service and persistence tiers. Enterprise Java developers use the tools they are used to using in their existing development environment. It is in this way that Jiffybus is not an alternative to Java 2, Enterprise Edition (J2EE) environments, but rather is complimentary to them.

Where Jiffybus provides an advantage is in the space between the presentation tier and the business services tier. In a traditional web application, this function is supplied by HTML. But in a Jiffybus application, we want to use a rich, interactive user interface that exceeds what HTML can do. The challenge, then, is to come up with a way of communicating requests for business services from the presentation layer to the business services layer, handing the request off to a business service component that can handle the request, and finally communicating the response back to the client in a way that makes it easy for the client to digest.

Jiffybus accomplishes this by piggybacking on HTTP. HTTP works because it is simple to deal with, is ubiquitous, and has excellent support in Java. It is also designed for this very thing: client sends request to server, server sends back a response. Two types of objects participate in the Jiffybus request/response cycle: *Service Request* objects

represent the request itself, while *Service Provider* objects generate the server-side response and send the response back to the client. This picture provides a high-level overview of the pieces and how they fit together:



The presentation tier formulates a Jiffybus service request, serializes it, and adds it to the message area of the HTTP request packet. A socket is opened to the web server, and the HTTP request is sent. The web server receives the request and sees (by its URL) that it is destined to be handled by a servlet, so the servlet is invoked and is passed the request packet. The servlet extracts the Jiffybus service request object from the packet and examines it. The request contains information about the Jiffybus service provider that is to handle the request; the servlet finds the service provider (instantiating it if necessary) and passes it the request object. The service provider performs the requested service and builds a response object (which can be anything that is serializable, though Jiffybus provides some convenient specialty response classes.) The servlet in turn serializes the response object and writes it into the socket stream that is connected to the presentation tier code on the other end. The presentation tier receives the response and adjusts the user interface as appropriate based on its contents.

In the rest of this overview, we will look at the details of this round-trip from client to server and back. Specifically, we will examine:

- The databus: moving data between the tiers
- Service Providers: Performing business services
- Middle tier proxies
- Client-side infrastructure: AbstractClientLogin and ServiceHub
- Metadata, case data, and the Swing model adapters
- The util package: Logging and miscellaneous utilities
- An example