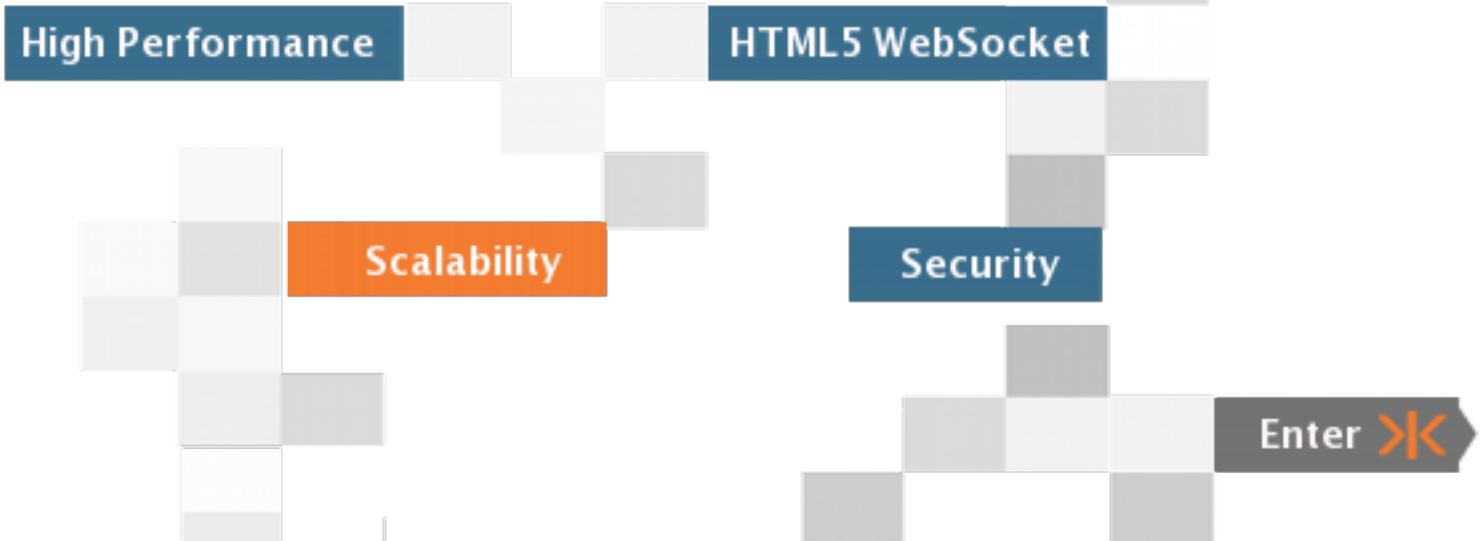
KAAZING

BUILDING LIVING WEB APPLICATIONS

A TECHNICAL WHITEPAPER



Copyright © 2012 Kaazing Corporation. All rights reserved.

kaazing.com

Building Living Web Applications

HTML5 WebSocket represents the most important upgrade to the Web in its history. By leveraging this new standard along with Kaazing WebSocket Gateway and a set of software design patterns well-suited to dynamic data flows, enterprises can build Living Web applications that tap new sources of revenue, cut expenses and astound users. The patterns described in this whitepaper form a set of best practices for building those Living Web applications.

OVERVIEW

Today's Web is a dynamic fabric of users, applications and devices – a constantly evolving platform on which we conduct an ever-growing share of our daily lives. This is a far cry from the original Web, which was a network of static text pages hyperlinked by URLs. As the legacy Web has morphed into today's Living Web, users have become considerably more demanding of online applications, insisting on real-time information delivery, fully interactive user experiences, and speed – lots and lots of speed.

Yet the fundamental architecture of the Web has barely changed. Until recently, every Web interaction relied on the very same HTTP request/response mechanism that was used to execute the world's first Web transfer more than twenty years ago. Since the mid-1990s, developers have employed a series of clever workarounds to simulate real-time interactivity; but these techniques have proven costly, complex and only partially effective. Living Web applications require a very different fundamental architecture.

Enter HTML5 WebSocket, the first major upgrade in the history of the Web. While static resources will continue to rely on HTTP, dynamic data will now flow freely over WebSocket connections that are persistent (always on), full duplex (simultaneously bi-directional) and blazingly fast. Revolutionary on purely technological grounds, WebSocket's most profound impact will be on the quality of online life and the economics of online business. Combined with an enterprise-grade WebSocket gateway like the one developed by Kaazing, the WebSocket standard will enable entirely new kinds of applications, just as the Web itself did. Businesses will quickly leverage this new technology to tap novel sources of revenue, reduce time-to-market and slash the costs and complexity of their IT infrastructure.

This whitepaper describes some key software strategies that businesses can use with WebSockets and Kaazing WebSocket Gateway to build Living Web applications which capture this host of top-line and bottom-line benefits.

Copyright© 2012 Kaazing Corporation

LIVING WEB ARCHITECTURE

The seven design patterns described below form a set of best practices for building Living Web applications. Here, a "pattern" refers – in the spirit of Christopher Alexander's "pattern language" – to a general, reusable solution to a commonly occurring problem within a given context. (For a good discussion of design patterns for software development, see http://www.patternlanguage.com/leveltwo/caframe.htm?/leveltwo/../bios/designpatterns.htm.)

Patterns are not mutually-exclusive but combine to form a single system – in this case, a single Living Web application. Patterns are also not rigid requirements; an application can lack one or more of those outlined below and still warrant the title "Living Web." Yet when a developer building a Living Web application decides to forego a given pattern – an activity stream, for example – chances are that the final product will nonetheless feature something very similar. So it is generally best to consider all of these patterns explicitly from the outset.

The patterns discussed below differ in their degree of specificity (they are listed from most general to most specific). Yet the theme common to all of them is that Living Web applications are fueled by messaging. The messaging paradigm enables software systems that are fully asynchronous and component-based; so a messaging approach removes the shackles of both time and space and enables applications that are not only dynamic and interactive but simple and inexpensive to build, deploy and scale. This approach has the added benefit of enterprise-readiness: the abundance of highly scalable, industry-standard protocols designed for asynchronous messaging means that developers need not reinvent the wheel, provided their development and deployment environments include infrastructure elements such as Kaazing WebSocket Gateway that supports these protocols.

It is important to note that a focus on messaging does not preclude the use of other interaction patterns needed to build out the solution. In particular, access to back-end data will often rely on database patterns. With an asynchronous, non-blocking messaging architecture, this presents no problem at all: back-end servers are responsible for accessing data stores to retrieve data, which is then passed (asynchronously) to the de-coupled messaging system, allowing all interaction with clients to be accomplished via messages. The integrity of back-end data is preserved while clients receive a full-fledged Living Web experience.

Pattern I: Browser-side MVC

Most server-side platforms such as Java, Ruby, PHP and others provide out-of-the-box MVC frameworks that treat the browser HTML page as a basic view and consign the processing to the server. Java developers are undoubtedly familiar with the concept of a Java Servlet and will remember the plethora of MVC frameworks that abstracted the Java

developer away from the browser. Whether it is struts or JSF or Wicket, what these frameworks all had in common was that they provided a way for the developer to think of the client application in terms of views without having to worry much about what was going on in the browser. All widgets and UI elements were taken care of by the framework, and virtually the only person who needed to look at the browser was the designer who applied the visual styles (CSS) to the pages created by the framework.

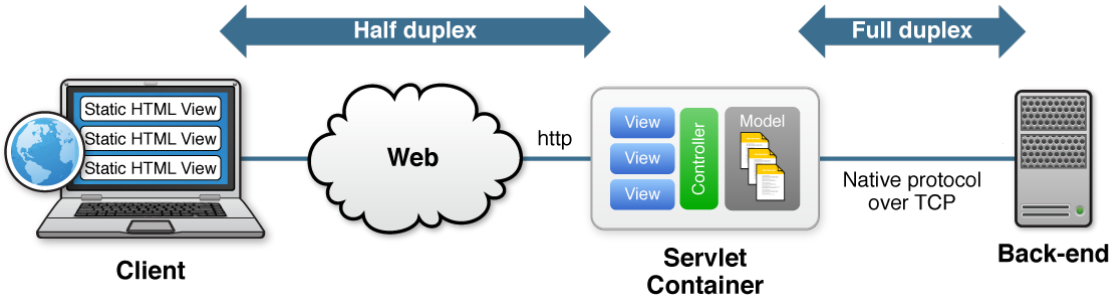


Figure 1

In a legacy Web application, the MVC framework resides in the Servlet engine and serves up static HTML views via http request/response

With HTML5 and a robust WebSocket gateway such as Kaazing, these frameworks will slowly become obsolete as a good portion of the MVC framework can be offloaded to the browser. In fact, the entire Servlet container may become obsolete. Because native protocols can reach the browser via WebSocket, the entire client can be built in the browser as a single page HTML5 application, since both the view and the controller can run on the client. The model is now the bridge between the client and the server, as it should be.

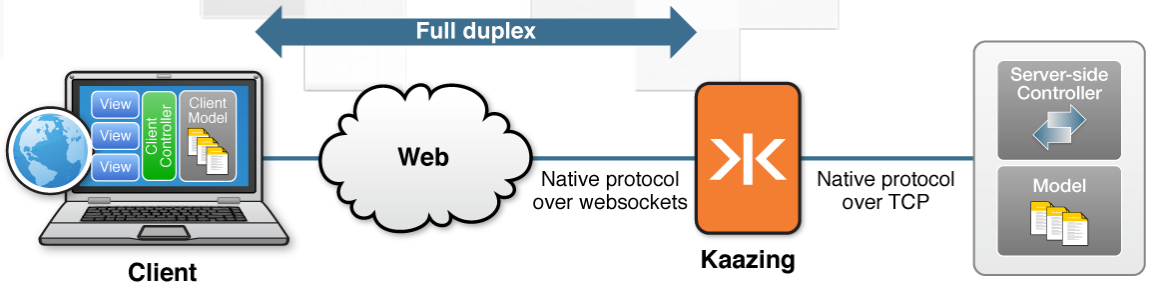


Figure 2

In a Living Web application, the MVC framework runs in the client; for example, in the Google Chrome browser, it would be written in JavaScript and run in the V8 engine. Communications are now handled via native protocols that run over WebSockets. Examples of appropriate protocols are discussed in the next section

One might also argue that some things should still be computed on the server to take weight off the client. Perhaps that might have been the case back in the days of x386 processors, but the argument no longer holds true in light of the massive computing power now available on the client. Google Chrome's JavaScript V8 engine is a case in point: it is such a powerful engine that it is now available to run as a server as well. So why offload anything to the server unless it has to do with persisting data objects on the server or collaborating in real time with other users or other devices? Even computational items that require knowledge of other data that might not be present in the browser can still be performed in the browser thanks to efficient messaging that can quickly bring the data to the client.

HTML5 has so many new UI elements and new ways of accessing the DOM via JavaScript that UI frameworks such as jQuery become less important to writing powerful and dynamic clients. But it is usually a good idea to write a client via an MVC pattern so the resulting code is easy to maintain. Developers can write their own, or they can actually leverage a number of open source client frameworks that provide an MVC framework out of the box. These are outlined in the Appendix.

JavaScript is a dynamically typed language; and while it is incredibly easy to write code with JavaScript, some developers still prefer strongly typed languages. There are a number of such languages available to run in the browser. They do this by compiling into JavaScript that can run in the browser's JavaScript engine. The Google Closure library is one such language; meanwhile others are surfacing, such as Google's DART language.

Writing rich client applications is a topic that merits its own discussion, but a few references are included in the Appendix. Note that some controller functions must continue to reside on the back end, though the controller in the browser and the controller in the back end are now intertwined via asynchronous messaging, and views are entirely controlled within the browser-side client framework. The next section delves into more details of how this is accomplished.

Pattern II: Asynchronous Messaging

A rich client in the browser now needs to be able to talk to the server. Unlike HTTP, where the only protocol of choice is http-request-response, it is now possible to choose the protocol that is best suited to any specific type of application. At the most basic level, if all that is needed is to pull some data from the server to display on the client side, it is tempting to use a simple database protocol to perform queries on a database. While there may be some value in doing this, such as making it easier to provide offline application behavior and auto-resync with transactional consistency, this approach is not evolving past the old request-response paradigm. Effectively, one type of request-response (http) is being replaced with another (JDBC for example) and thus inherits all the limitations associated with the general request-response approach to building applications.

The major limiting factor of the request-response paradigm is that it is designed for users looking for data. But in today's world, the data is changing so fast that this approach is no longer suitable. By the time the data is pulled from the server, it has already changed. In a Living Web enterprise, it is no longer just the users who are looking for data; instead, the data comes looking for any user who might be interested in it. The bottom line is that to evolve to the Living Web, another approach becomes necessary.

In a real-time application, the client and server need to be able to send messages to each other to express events. Each event is represented by a message with a particular header and a specific payload. The most effective way to handle the message traffic is to rely on a specific message protocol. There are myriad protocols from which to select. Kaazing WebSocket Gateway provides complete freedom to choose the protocol that best matches specific requirements. However, there is a specific class of protocols that will be most appropriate for today's real-time applications.

There are two very basic classes of messaging protocols. The most primitive class features a remote procedure call pattern, which essentially allows calls to be made to a server to run a function or method on the server, and then waits for the return call. IIOP and RMI are examples of this class of protocol in Java. The problem with these protocols is that they force the usage of what is essentially a blocking pattern of request/response and are thus not well-suited to the fluid and asynchronous pattern of data transfer associated with real-time applications.

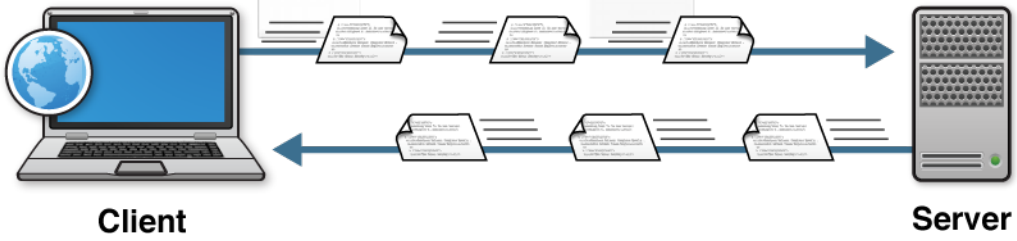


Figure 3

With an asynchronous messaging pattern, the client and server are loosely coupled via messages that move in both directions. The client is not idling for returns on server requests, and the server is also able to push messages out to the client at will. Both sides continue to execute in a non-blocking fashion, and network latency is no longer a limiting factor on execution.

The second class of messaging protocols uses an asynchronous message passing pattern with a loose relationship between requests and responses. One can still do basic request-response calls, but it is now possible to leverage the power of an entire system that is designed from the ground up to be asynchronous from end-to-end. The advantages of such a system make the asynchronous pattern incredibly compelling. Hardware resources are maximized by minimizing the number of threads blocking on IO operations, and the network bandwidth scales to its full capacity.

By contrast, with an RPC approach, each request requires a wait time before receiving a response, and thus the application performance is limited by the round trip time (or latency) of the network. In an asynchronous system, message flows can be pipelined in different directions, and the limiting factor becomes the network bandwidth as opposed to the latency. The key with WebSocket is to leverage the asynchronous advantage all the way from the client to the data storage layer in all aspects of the application. The result is higher performance and more efficient network throughput across the board.

The classic reference protocols for the asynchronous message pattern are AMQP (Advanced Messaging Queuing Protocol) and STOMP (Streaming Text Orientated Messaging Protocol.). AMQP is the most up-to-date open standard; it has a very flexible structure that includes a number of elements that make it suitable for a large number of messaging patterns. The protocol itself is specified with a compressed binary format. In contrast, STOMP is more basic in terms of the elements it provides, and it is based on an uncompressed, open text standard – making it possible even to talk to a broker using STOMP directly via a telnet client. Depending on the back-end architecture, it may be

best to leverage a proprietary protocol from a leading vendor that can offer additional advantages not available with the open standard protocols. In general, all of these protocols not only support asynchronous calls, they also support a clean decoupling between the senders of messages and the consumers of messages.

```
MESSAGE
destination:/queue/FOO.BAR
receipt:
timestamp:1190811984837
priority:0
expires:0
message-id:ID:fboltond820-2290-1190810591249-3:0:-1:1:1

Hello, queue FOO.BAR
```

Figure 4

Example of a STOMP message frame, all in open text format. Since STOMP is text based, it is possible to run a telnet session and speak STOMP to a message broker directly via the keyboard.

Pattern III: Real Time Service Oriented Architecture

On the server side, a message broker becomes a natural fit for processing messages. The broker will support a specific messaging protocol and ensure that senders are clearly decoupled from receivers and that multiple senders and receivers can be supported on the same wire. For example, in a JMS (Java Message Service) broker, this is accomplished via the creation of topics and queues from which both client and server components can freely publish and consume. The brokers essentially support the creation of flexible and loosely coupled application components, while the protocols ensure the co-existence of large volumes of message traffic for multiple publishers and consumers on the same wire. The elements that make up the structure of the publishers and subscribers of messages are often referred to as a message bus.

Of course, there are many different ways to structure a message bus; it all depends on the goals of the application. A healthy message bus implementation will also include a set of modular back-end services that are as decoupled from the rest of the message bus and one another as possible. Not only does this allow the system to scale based on

which service has the most load, it also allows easy, plug-and-play-style swapping out, upgrading or improving of individual services without affecting the overall functionality of the application.

Ideally, all of the services are coupled only loosely via messaging; as such, it is possible to have a back end that is serviced by multiple platforms and languages. This structure is akin to a message-driven Service Oriented Architecture BUS; more broadly, this structure is the default architecture for any real-time non-blocking back end that is based on true concurrency. Unlike standard SOA services, the services here are 100% asynchronous and message-based. One of the ideal advantages of the loose coupling is that it allows the hot swap of services without affecting the application.

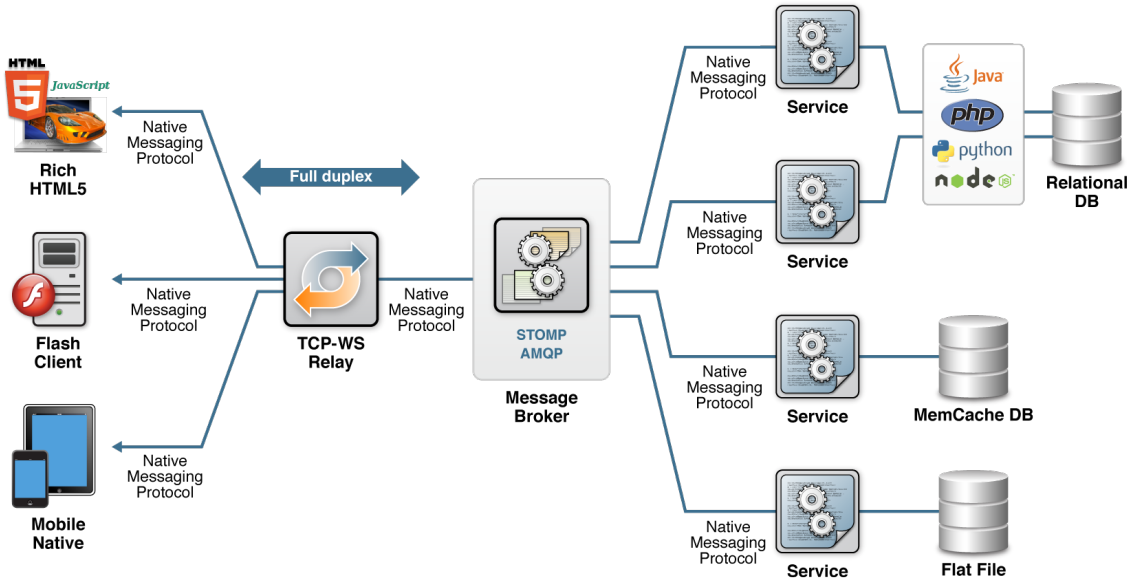


Figure 5

The overall message bus architecture, which assumes that the back end is decomposed into a set of loosely-coupled services. The coupling is done entirely via asynchronous messaging to maximize concurrency and enable the seamless replacement of individual components. Because of the large degree of decoupling, it is possible to have a back end written in multiple languages.

Finally, the message bus is not complete without a proper client service strategy. Unfortunately, most message brokers were never designed to service millions of browser clients. This is precisely where a solid WebSocket gateway such as Kaazing's comes into the picture. In general, it is undesirable for millions of users to have their

own dedicated connections to the back-end message broker. A gateway can effectively function to extend a message broker to millions of client Web browsers by providing a number of services to scale a broker to handle the load. Examples of basic services needed include connection offloading, load balancing, single sign-on, security and encryption.

Pattern IV: Decoupled Asynchronous Business Logic

Because deployment requirements change rapidly in the enterprise, it is crucial to decouple the scaling of the underlying software platform from the business logic. This way, as a business grows and applications must service more end users, the underlying messaging architecture can be evolved and extended without affecting the business logic.

The decoupling is accomplished by abstracting the messaging layer in such a way that the core functions such as topic/queue/exchange creation, along with user authentication and load balancing of message workers, are not visible to the business developer, who is then free to concentrate on the required business functionality. In an ideal case, the business developer just initializes the application context and lets the framework handle all lower level functions.

The favored programming pattern follows RPC type calls, so that a business developer can easily identify business objects and their associated methods. However, it is important to note here that in the case of Living Web applications, the RPC is asynchronous and will always have non-blocking callbacks that can represent either a single return or multiple returns, the latter representing a message stream. Caution is advised in how the syntax is implemented. The asynchronous aspects of the RPC calls need to be emphasized to avoid misleading a developer into thinking of the code from a blocking perspective.

In both client and server code, RPC function calls should always provide their return value in the form of non-blocking callbacks.

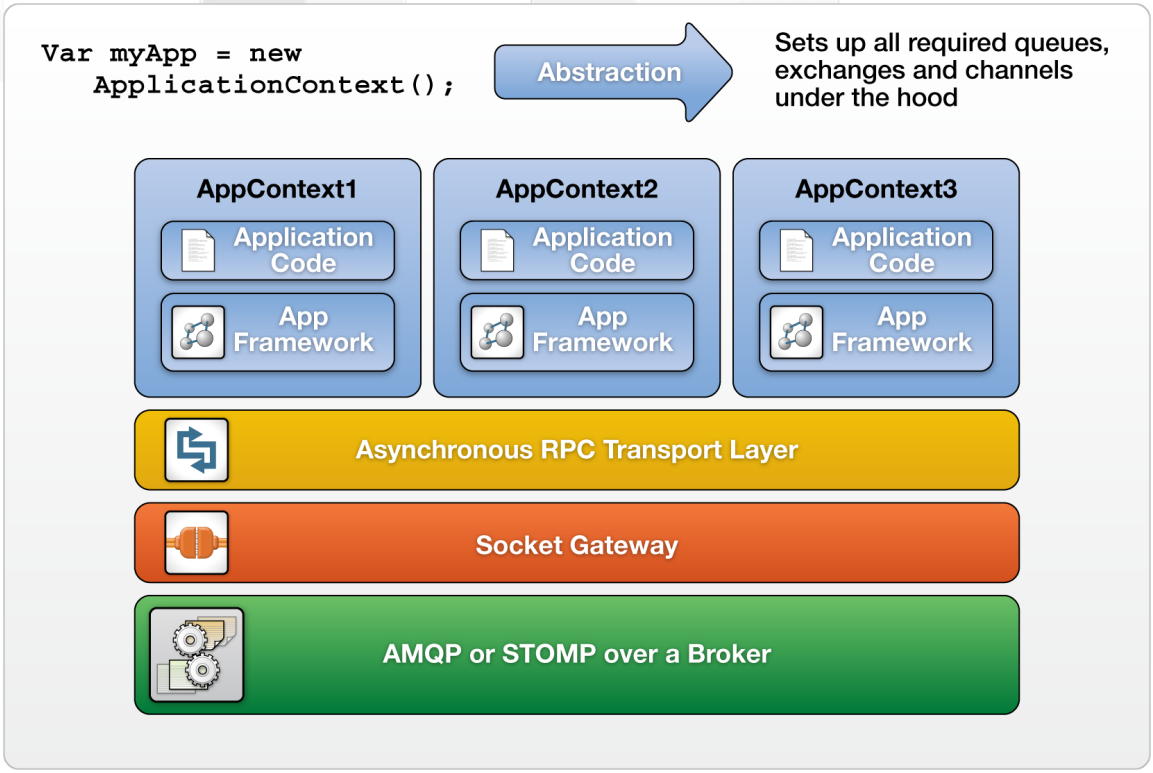


Figure 6

Layered approach that emphasizes separation of the message bus transport layer and the business logic. The syntax for the business developer is such that the entire "plumbing" is abstracted into a single AppContext so that the developer can focus on just the required business logic.

A large development team may choose to create a home grown, proprietary framework that best reflects the application needs, while smaller teams can leverage existing frameworks. But the choice is also limited by the desired programming language for the back-end message services or the front-end client services, since not all languages have readily available frameworks to leverage.

To illustrate the principle, a sample JavaScript client code for monitoring business objects within an enterprise activity stream is provided below:

```
// what to do once user is authenticated

function onEventLogin(userData){

        myAppContext = newAppContext(userData.user, onAppContextCreation);

}

// What to do once the application context is created

function onAppContextCreation(){

        //open the business object channel

        myAppContext.openBusinessObjectNotificationChannel(getHistory,onBOEvent);

}

// what to do when the business object arrives

function on BOEvent (businessObject){

        //

        displayStream(businessObject);

}
```

An example of an RPC framework using Python is QUAM; it sits on top of AMQP and is compatible with any broker that supports AMQP. There are frameworks for other languages as well. Apache Camel allows the use of declarative descriptors to bind together a wide array of back-end services into a unified message bus and provides some basic RFC support.

The preferred payload for asynchronous RPC in Java used to be JAX-RPC, which encoded the RPC in an XML object. But XML has fallen out of favor and is rapidly being replaced by JSON-RPC (see http://jsonrpc.org/spec.html). There are many advantages to using JSON. It is natively compatible with JavaScript in the client and removes the complexities associated with XML. In addition, JSON is now supported in practically all other languages, most notably Java, PHP and Python. However a back end is implemented, if the language used does not have concurrency built in (Java for example), it will be necessary to make liberal use of a concurrency library (Java.util.concurrent in the case of Java) to make sure all code executes in a non-blocking fashion.

Because Kaazing's gateway is built on open standards, it will support any standards-based framework and language that is compatible with messaging.

Pattern V: Living Business Objects (or Observer or Pub/Sub) Pattern

In legacy systems, business objects are designed for request-response queries. This means that unless a read or update query comes in from a client, the business object remains static. In a Living Web application, the business object can initiate a two-way dialogue. In essence, the business object will now send updates to whoever is interested. In the old days of request-response, anyone who was interested in changes in the business object would query it on a regular basis, leading to a huge number of empty requests: "Have you changed? No, I have not changed…." With a two-way dialogue, the living business object now informs anyone who is interested when a change has occurred.

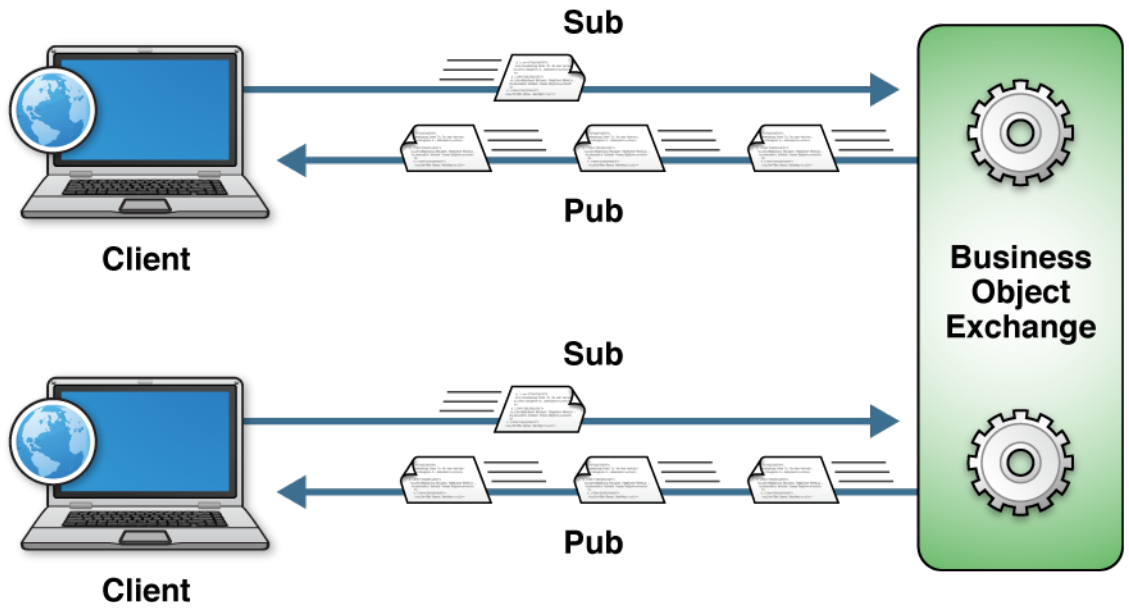


Figure 7

In the observer pattern, clients will subscribe to a virtual exchange that represents a particular business object and will then receive all updates related to the activities around that business object. In that sense, business objects come to life, and clients can see them change in real time. Notice that the bulk of the traffic is actually from the server to the client, as opposed to the client making all the requests.

The pub/sub pattern is ideally suited to supporting living business objects. All client services that are interested in changes in a particular object subscribe to those changes; when changes occur, the business object notifies all interested parties. This means living business objects are essentially asynchronously coupled with one another. Rather

than single objects calling on the methods of other objects in a blocking mode, an object will instead subscribe to a specific task or activity of another object and will be notified when that task or activity occurs. The subscribers are referred to as “observers” and the object being observed as the “publisher” (or the “subject”). Publishers notify subscribers when events occur.

There are modern out-of-the-box solutions such as Redis that accomplish this feat without the need of additional coding. But companies with a large investment in a back-end office can leverage a broker to write pub/sub service facades for their business objects. In this way, the business objects get plugged into the message bus in a natural way. It is advisable to write the façade in Java or whatever language the legacy back end is written in (ABAP or COBOL) and to leverage a caching database like Memcache to temporarily manage pub/sub states.

Pattern VI: Transaction Percolation

The pub/sub layer is essentially the real-time layer for living business objects. In general, data at this layer (such as updates and notifications) are non-critical, and the loss of such data is generally acceptable under general system failure. Any persistence requirement that serves as the single source of truth will be offloaded asynchronously to a persistence service that is generally separate from the pub/sub layer.

However, one exception applies to business transactions. The very nature of business transactions requires that they be stored with a commit. But in most instances, back office systems where such transactions need to be committed are too slow. Therefore, there is a need to persist any transactions that take place within the pub/sub system before trying to commit them to the back office.

Copyright© 2012 Kaazing Corporation

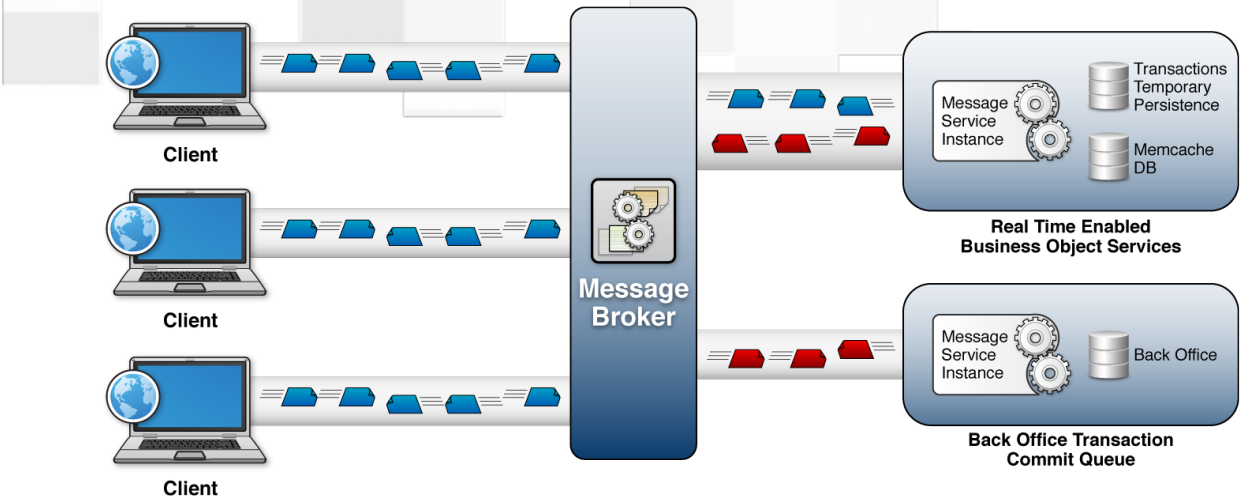


Figure 8

In the percolation pattern, clients will interact primarily with the real-time business objects and even be able to commit transactions at that level; but transactions are synched asynchronously via a separate and correspondingly slower back office service.

If the application leverages the message bus architecture pattern, it will not be hard to add a long-term persistence agent to the bus. The message layer can then be used to percolate transactions from the dynamic living front end to the core back office once they are vested. They can be queued up to be processed asynchronously and will populate the back office over a longer period of time.

Pattern VII: The Activity Stream Pattern

An activity stream is at the core of any real time Living Web application. The "activity stream" here is the stream of all data activities relevant to that user, not to be confused with an "Activity Stream UI" such as those found in Facebook and other social networks. In a generalized way, each user has his/her own activity stream; as the name implies, the activity stream is a stream of Activities usually associated with business objects that can be generated either by the system agents, by the user himself/herself or by other users.

As previously noted, the activity stream itself is not to be confused with the user interface representation of an activity stream. Rather, it is a core system object that ties together all the parts of a Living Web application. All business object activities are published into that user's activity stream, and the activity stream then publishes into various user interfaces such as a Web representation of the activity stream (e.g., Google Plus or Facebook Wall for social activities),e-mail notifications, SMS notifications or – more subtle yet – some notification feature within a larger Web user interface.

The activity stream is a useful programming structure to tie together the fabric of real-time interactions between the user and the system. It has become such a common structure that a standard now exists to represent it (see http://activitystrea.ms/specs/json/1.0/). JSON is a natural way to represent the activity stream since one of the most common expressions of an activity stream comes in the form of a JavaScript-driven Web UI. While the standard was originally created to support social networking activities, it is equally relevant to any enterprise setting.

The core of the activity stream specification is the activity object, shown in the Appendix. While it looks complicated, the object essentially describes what the activity is, who did it and when they did it. The real-time aspect is built into the spec by noting that the time stamp is described by the publish date.

The most natural fit for implementing an activity stream is the pub/sub pattern. In a business setting, the activity stream observes various business objects on the back end, and the business objects publish changes in the form of activities. These are then available from the activity stream to be published to wherever the user needs them.

SUMMARY

The arrival of HTML5 along with enterprise-grade infrastructure such as Kaazing WebSocket Gateway enables a completely new class of dynamic, interactive Living Web applications. The seven design patterns described in this whitepaper constitute a set of best practices for building those applications. The patterns themselves – unlike the applications they enable – are far from revolutionary, but are instead twists on well-established designs that will be familiar to any software architect or developer.

APPENDICES

Appendix I: Browser based GUI Frameworks

On the client front end, JavaScript is the most versatile language for writing applications; but trying to write to the DOM in the browser can be tedious and hard to maintain. This is where the use of a good framework can go a long way in helping to build a robust, maintainable client. This appendix provides summaries for a few such frameworks.

Backbone

Backbone makes it easy to represent data as models; and the models can readily be created, validated and destroyed. There is no need to write the glue code that looks into the DOM to find an element with a specific id, and update the HTML manually — when the model changes, the views simply update themselves. Out of the box, Backbone is designed to work with a RESTful JSON interface, but it is fairly easy to write some glue that will tie Backbone models to a WebSocket connection.Backbone is open source and can be downloaded from http://documentcloud.github.com/backbone/.

Knockout

Knockout is similar in functionality to Backbone, but slightly lighter weight (29kb) and more declarative in nature. Knockout is also open source (MIT License) and can be downloaded at http://knockoutjs.com/.

Sencha

Sencha (http://www.sencha.com) provides a number of rich client frameworks under various licenses, from open source to basic commercial licenses. Their offerings include libraries that provide MVC functionality and support for multiple browsers. They also have a designer called the Sencha Animator, at http://www.sencha.com/products/animator/. Their Ext JS 4 library is at the core of their offering and provides a lot of features. Please see http://www.sencha.com for more information.

Dojo

Dojo is an extensive open source toolkit (dojotoolkit.org) that provides a number of advanced features for building rich browser applications. This toolkit is supported by the Dojo Foundation (dojofoundation.org), which itself is supported by a number of large enterprise companies such as Google, TIBCO and a few others.

Appendix II: The Activity Stream Object Specification Version 1.0

Object actor	Describes the entity that performed the activity. An activity MUST contain one actorproperty whose value is a single Object.
JSON String content	Natural-language description of the activity encoded as a single JSON String containing HTML markup. Visual elements such as thumbnail images MAY be included. An activity MAY contain a content property.
Object generator	Describes the application that generated the activity. An activity MAY contain a generatorproperty whose value is a single Object.
Media Link icon	Description of a resource providing a visual representation of the object, intended for human consumption. The image SHOULD have an aspect ratio of one (horizontal) to one (vertical) and SHOULD be suitable for presentation at a small size. An activity MAY have anicon property.
JSON String id	Provides a permanent, universally unique identifier for the activity in the form of an absolute IRI [RFC3987]. An activity SHOULD contain a single id property. If an activity does not contain an id property, consumers MAY use the value of the url property as a less-reliable, non-unique identifier.
Object object	Describes the primary object of the activity. For instance, in the activity, "John saved a movie to his wishlist", the object of the activity is "movie". An activity SHOULD contain an object property whose value is a single Object. If the object property is not contained, the primary object of the activity MAY be implied by context.
date-time published	The date and time at which the activity was published. An activity MUST contain a published property.
Object provider	Describes the application that published the activity. Note that this is not necessarily the same entity that generated the activity. An activity MAY contain a provider property whose value is a single Object.
Object target	Describes the target of the activity. The precise meaning of the activity's target is dependent on the activities verb, but will often be the object the English preposition "to". For instance, in the activity, "John saved a movie to his wishlist", the target of the activity is "wishlist". The activity target MUST NOT be used to identity an indirect object that is not a target of the activity. An activity MAY contain a target property whose value is a single Object.
JSON String title	Natural-language title or headline for the activity encoded as a single JSON String containing HTML markup. An activity MAY contain a title property.
date-time updated	The date and time at which a previously published activity has been modified. An Activity MAY contain an updated property.

Copyright© 2012 Kaazing Corporation

JSON String url	An IRI [RFC3987] identifying a resource providing an HTML representation of the activity. An activity MAY contain a url property.
JSON String verb	Identifies the action that the activity describes. An activity SHOULD contain a verb property whose value is a JSON String that is non-empty and matches either the "isegment-nz-nc" or the "IRI" production in [RFC3339]. Note that the use of a relative reference other than a simple name is not allowed. If the verb is not specified, or if the value is null, the verb is assumed to be "post".