An Introduction to Implicit Invocation Architectures

Benjamin Edwards (ben@ben-edwards.com)

Form follows function - that has been misunderstood. Form and function should be one….

-Frank Lloyd Wright

ColdFusion's initial appeal was to "webmasters" who wanted to make their sites more dynamic. It succeeded admirably. But just as the term, webmaster, is an anachronism, the call for more dynamic websites has been succeeded by the need for true web applications. As these applications become more involved and more ambitious in scope, ColdFusion developers find that a thorough knowledge of tags and functions isn't enough. To build scalable, robust applications—especially applications that can evolve successfully—developers must involve themselves in developing an appropriate *software architecture*.

In this paper, I will discuss some of the issues involved in a software architecture and go into some detail on one particularly flexible software architecture, labeled *event-based, implicit invocation*.

Software Architectures

In "An Introduction to Software Architecture"[1], David Garlan and Mary Shaw of Carnegie Mellon University describe software architectures as going beyond algorithms and data structures that make up an application. Software architecture addresses "Structural issues [that] include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives."

Application design occurs at several levels. In an object-oriented system, decisions must be made about the interfaces of an application, the classes that implement these interfaces, the choice of ways that classes communicate and are related to each other (this last is usually the domain of design patterns). Each level of design represents a new level of organization. The architecture of an application is the most encompassing of all design decisions.

In practice, software architectures are commonly treated as a collection of components and connectors. Components are the system's functional elements. For example, a shopping cart, a contact manager, and a database could be components of a software architecture. Connectors are the protocols for communication between components. Examples of connectors include method calls, SQL queries, and HTTP requests. A system's chosen architecture determines both the vocabulary of components and connectors that can be used as well as the set of constraints defining how they are combined.

The choice of a particular software architecture is made on the basis of an overall system organization—which is to say that there is no single-fit, perfect architecture. Over time, several different software architectural styles have been created—each having strong points and weaknesses. The more popular architectures include:

- Pipes and filters

[1] Available at: www.cs.cmu.edu/afs/cs/project/able/ftp/intro_softarch/intro_softarch.pdf

- Data abstraction and object-oriented organization
- Layered systems
- Repositories
- Event-based, implicit invocation

Two metrics important for consideration in defining the publicly exposed interfaces of an architecture's components and connectors are a system's *cohesion* and *coupling*. Cohesion is a measure of the degree to which a component has a singular purpose. The greater cohesion a component exhibits, the more focused is the component and the fewer are the assumptions about contexts for reuse.

Coupling is the degree of interdependence between components. The less a component relies on other components (the looser its coupling), the more independent and reusable it is. Maximized cohesion (simple components) and minimized coupling (fewer connectors) are hallmarks of a flexible, maintainable architecture.

Event-based, Implicit Invocation

Event-based, implicit invocation is an example of a well-crafted architectural style with high cohesion and loose coupling. As such, it is one of the more broadly accepted architectural styles in software engineering. Examples of implicit invocation systems abound, including virtually all modern operating systems, integrated development environments, and database management systems.

Garland and Shaw describe implicit invocation systems: "The idea behind implicit invocation is that instead of invoking a procedure directly, a component can announce (or broadcast) one or more events. Other components in the system can register an interest in an event by associating a procedure with the event. When the event is announced the system itself invokes all of the procedures that have been registered for the event. Thus an event 'implicitly' causes the invocation of procedures in other modules."

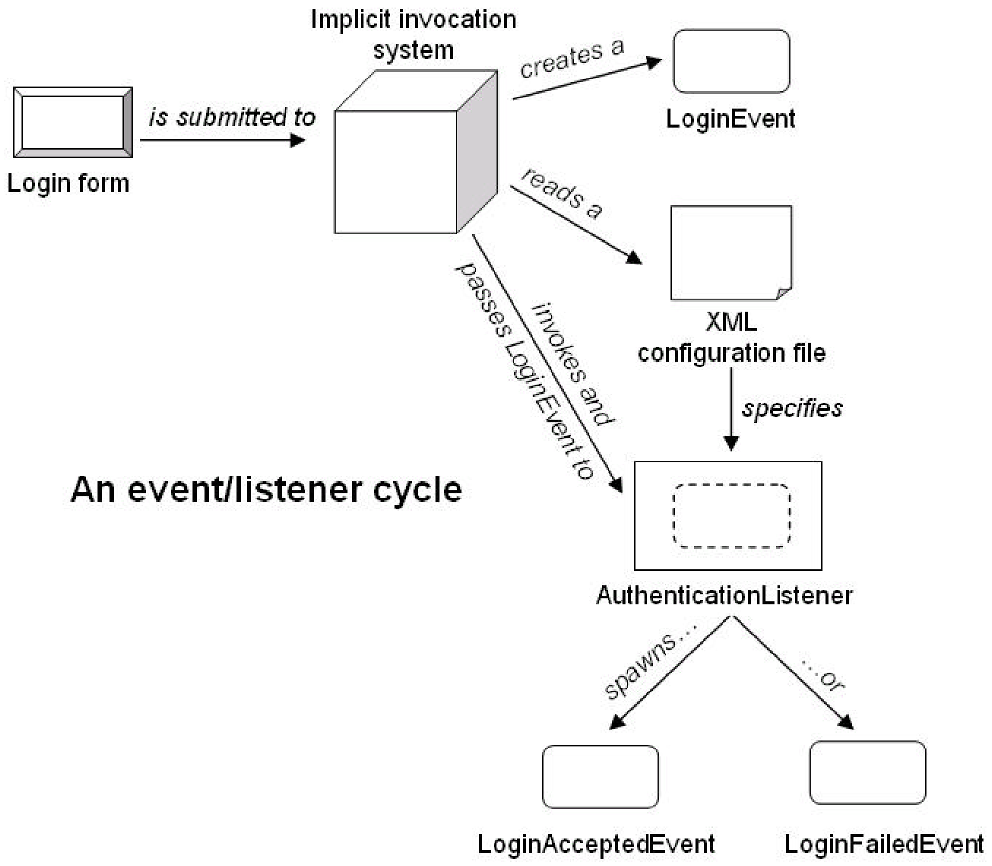
Implicit invocation systems are driven by events. Events are triggered whenever the system needs to do something—such as respond to an incoming request. Events can take many forms across different types of implementations; often for object-based systems an event is an object whose properties contain any contextual information needed to process the event (similar to how a HTTP request carries with it all its form and query-string variables).

When an event is announced, the system looks up listener components for that event. Listeners fit the same criteria for components that we've already discussed—they are functional modules of the system. Components that wish to act as listeners are registered to listen for certain events at configuration time (by specification in an XML file, for instance). When an event is triggered, all registered listeners of that event are passed the event by means of a dynamically-determined method call. In this way, functions are implicitly invoked. This process of notifying listeners of an event is called event announcement.

Events and listeners can themselves trigger other events. Let's consider a how a common login/authentication scenario can be represented in terms of events and listeners. In this example, a login form is filled out by a user and the form submitted. The incoming HTTP request triggers the creation of a LoginEvent, and the system populates the event with information in the request.

Next, the system determines the listeners for LoginEvent; in this case there is only one—the AuthenticationListener. Determined by a configuration file, the system invokes the

AuthenticationListener's tryLogin() method, passing to it the event. Based on information in the event, the tryLogin() method will seek to authenticate the user. If the authentication succeeds, a new LoginAcceptedEvent is triggered. If authentication fails, a new LoginFailedEvent is triggered. The cycle then continues, with any listeners of the new event being notified.



Implicit invocation architectures differ from *explicit invocation* systems in that implicit invocation system components use *events* to communicate with each other. Connectors in such architectures are bindings between events and component methods. Because these bindings are determined dynamically at runtime, components are loosely coupled; there is no compile-time determination of which method calls will be made. Loose coupling offers software architects the great benefit of increased flexibility and maintainability: new components can be added by simply registering them as event listeners.

Loosely coupled components work together, but do not rely on each other to do their own job. The interaction policy is separate from the interacting components, providing flexibility. Components can be introduced into a system simply by registering them for events of the system, aiding greatly in reusability. Introduction of new components does not require change in other component interfaces, providing scalability as new features are added. Overall, implicit invocation eases system evolution.

Architecture and Design Patterns

Software design begins with selection of an architectural style that will provide the proper framework to meet a system's non-functional requirements. Non-functional requirements are criteria the system has to meet that are separate from the requisite functions the system must perform. For example, a system might have for functional requirements the ability to log in a user, for that user to add items to a shopping cart or to checkout their cart. Examples of non-functional requirements that implicit invocation addresses include reusability, flexibility, and scalability.

Architectural styles are not the same as *design patterns*. While architectures define the components and connectors possible within the system, design patterns define the implementation strategies of those same components and connectors. A good architecture will make use of design patterns but has a broader goal. We may take, for an example, the popular design pattern, Model-View-Controller (MVC)[2]. MVC is not an architecture; it is a design strategy for implementing an architecture. Different architectures will find different design patterns more or less helpful. In the case of event-based, implicit architectures, the Model-View-Controller design pattern seems to be perfectly suited.

We said earlier that implicit invocation comes into play when replacing and adding other components. Under implicit invocation the controller notifies model and view components of updates to the system not by explicit method calls, but through the creation of events. Elements of the model, the view, and the controller can all create events that the controller uses to notify listeners—all without the listening components needing to be aware of each other.

Conclusion

Event-based, implicit invocation architectures provide benefits of reusability, robustness and—especially—maintainability to software code. By combining high cohesion of software components with loose coupling of these components by means of dynamic notification through the use of events, applications that rely on implicit invocation can adapt flexibly to the changing requirements and new uses to which so much corporate software is subject.

While implicit invocation has found wide use in other areas of software engineering, it has not been used to any great degree with web applications. But this is changing. Work is now underway on an event-based, implicit invocation architecture named "Mach-II". Its first incarnation uses ColdFusion components (CFCs) to implement a true MVC, event-based architecture. Mach-II provides a framework that can combine ColdFusion components, Java classes, web services, and Flash clients to create an application that is at once robust and easy to maintain.

For more information, visit www.mach-ii.com.

References

- www.cs.cmu.edu/afs/cs/project/able/ftp/intro_softarch/intro_softarch.pdf
- www.mach-ii.com
- www.halhelms.com

[2] While all good software engineering practice calls for separation of display and logic, not all such systems conform to a true Model-View-Controller design. The key to MVC is the controller. Used properly, the controller decouples the model from the view in a way that increases flexibility and reuse. View and model components are independent of each other and can literally be swapped out without affecting each other.

Author

Benjamin Edwards is a Sun Certified Java Programmer and holds a degree in Computer Science from the Georgia Institute of Technology with specializations in Software Engineering and Educational Technology. Ben co-founded Synthis Corporation while at Georgia Tech. Ben currently trains developers on software engineering practices with a focus on Java and Object-Oriented programming.

Special thanks to Hal Helms for his contributions to this article.