Explain  Event- based, Implicit Invocation with exampleTop of Form

**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 compiletime 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.