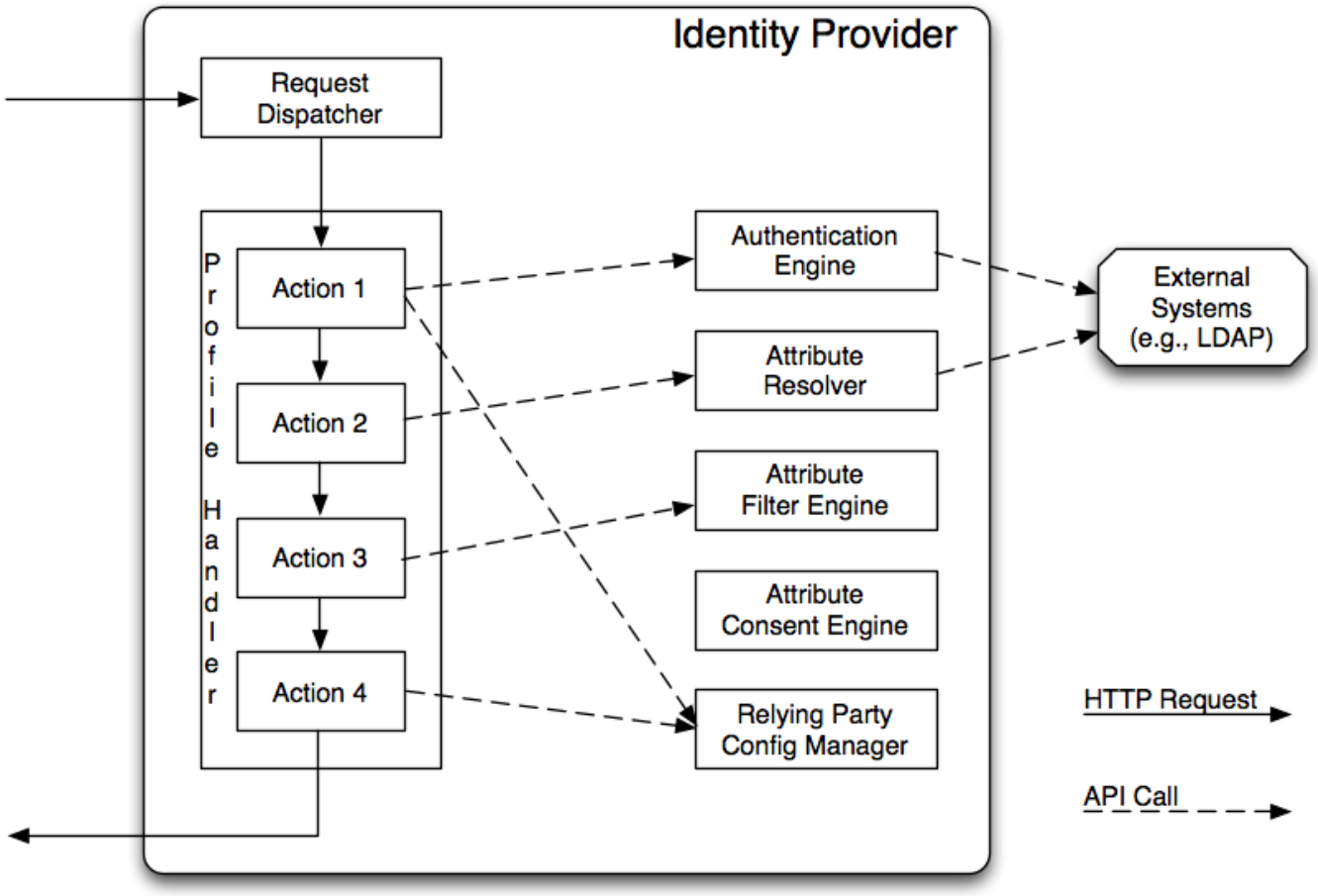
GeneralArchitecture

Overall Architecture



The overall architecture of the IdP is relatively straightforward. An HTTP request comes in to a *Request Dispatcher*. The dispatcher inspects the request and, based on the request's properties, sends it along to a *Profile Handler*. A *Profile Handler*, as its name implies, is designed to handle a particular protocol profile request (e.g., SAML 1 Attribute Query, SAML 2 Single SignOn).

In our particular design, the functions of request dispatch and profile handling are actually implemented by Spring Web Flow, which itself sits on top of the Spring MVC layer. Each request is mapped to a profile flow, which are the top level units of processing in the software.

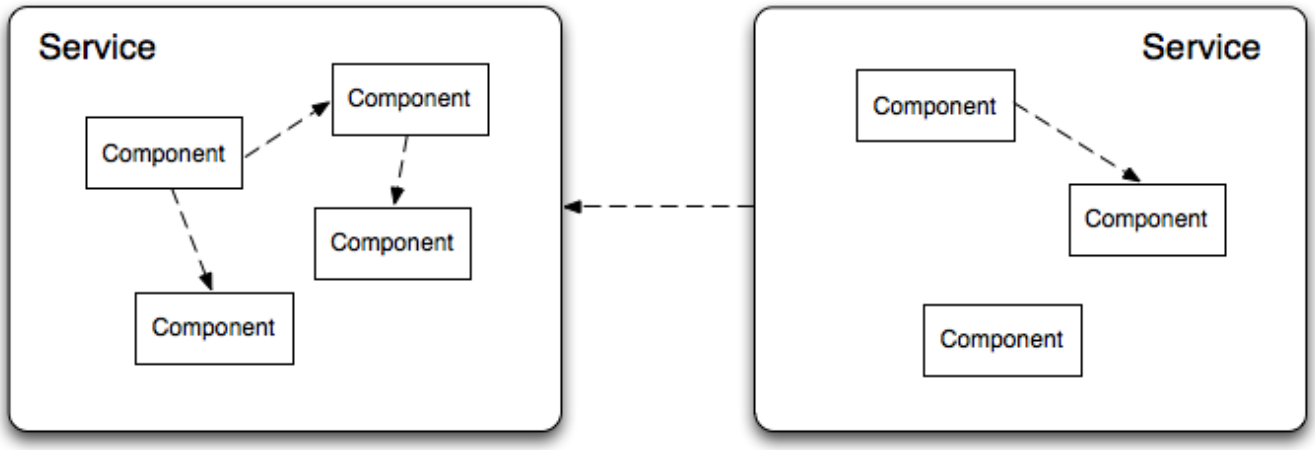
Each profile flow is composed of a set of actions which perform some part of the overall processing necessary to generate the appropriate response. Requests pass from action to action until the response is complete and returned to the requester. While at the high level, there are "actions" to perform user authentication, attribute resolution, and response signing, the actual implementation of each action is much more fine grained.

When an action is performing its work, it may in turn make calls to one or more of the IdP's services.

Components And Services

The IdP is mostly just a collection of individual bits of code wired together with Spring. Each discrete bit of code is termed a *Component*. Most components have a unique identifier (mostly used in logging and by Spring). A *Component* might be something small, like the LDAP authentication action, or something larger, such as the whole attribute resolver.

The example of the attribute resolver also demonstrates the ability for a bunch of smaller components to be composed in to a larger *Component*. A larger *Component* that hides the individual components that make it up and also contains configuration logic is known as a *Service*. Note that components in different services should never communicate with each other. However, components within the same service often communicate, as may two different services. Typically when services need other, one is injected into the other in a declarative fashion using Spring.



Use of Spring and Web Flow

Spring is used in two different ways by the Identity Provider. First, as mentioned above, the IdP uses Spring in its configuration. In a very real sense Spring is used to build the IdP from all the discrete components and services. The Spring-based configuration currently uses two different types of files.

Configuration files that are not expected to be touched by most deployers tend to use the generalized Spring bean file format. These configuration files are mostly about wiring the various services together and handling the "plumbing" of the IdP. Configuration files that are meant to be modified by deployers are a mix of native Spring and some custom schema configuration files.

Going forward we expect to make broader use of Spring's native configuration format, and less of custom files, but backward compatibility demands that we continue to support the older formats. While native Spring syntax can be more complex in some ways, it's also more consistent, better documented, and allows for a lot of syntactic tricks that make advanced features much easier to take advantage of.

The other use of Spring is the use of Spring Web Flow as the mechanism used to manage IdP profile logic, authentication processing, and essentially all of the orchestration of function in the system. This allows the profile and authentication processes to be more easily extended with unforeseen capabilities with less code and less risk of regressions.

The Spring Web Flow mechanism is used to execute ProfileActions, which make up the bulk of the IdP's processing model.

Contexts

The state of an executing Web Flow is managed in the IdP using a concept called the "context tree", a graph of objects each descended from a common class. Each context can have a parent, and maintains a class-indexed hashtable of children, limiting a tree to contain a single instance of a context type at a given level. Each context is typically of a derived type that particularizes it to store information specific to a single category of information. A context tree tends to be shallow, and is designed to support type-safe access to conversational state by ProfileActions.

At the root of the tree will be a context type that is specific to a particular pattern of interaction. The most common one in the IdP is a request/response pattern that is managed by the ProfileRequestContext context type. This context type chiefly just manages a slot for an inbound and outbound MessageContext, and then will contain child contexts reflecting the needs of the various actions that make up a particular profile flow.

Some general design principles:

- Major services within the IdP typically will rely on a specific context type as the root of a subtree that is used for input to the service and for communicating results back out. For example, the attribute resolver component relies on an AttributeResolutionContext.
- Most of the time, services only read or modify information within the subtree they are defined to operate on. The exceptions may be certain well-defined context types that are standardized across services. An example would be a SessionContext that makes a subject's session available. Otherwise, specialized actions exist to move information from service-specific subtrees into other subtrees or into more generic contexts.
- Protocol-specific actions such as code that processes SAML messages will typically read and write contexts designed to carry information specific to a given protocol. When services must operate on information generically, actions are provided to translate the information into generic form so that the boundary between protocol-specific parts of a flow and generic parts is respected. This facilitates the use of "subflows" to capture reusable sets of actions across multiple protocols.