Dependency Injection and Configuration

Background

Plugins in Log4j have long supported a primitive form of dependency injection through the use of annotations that indicate where in the configuration to obtain the injected data. This initially began with the static plugin factory method whose arguments could be *attributes* (keys with scalar values; in XML, this is an element attribute; in JSON and YAML, this is a field with a scalar value), *elements* (further-defined plugin objects to inject; in XML, this is an element; in JSON and YAML, this is a field with a non-scalar value), and *values* (a special kind of attribute; in XML, this is the text contents of an element; in JSON and YAML, this is another attribute typically named "value" or similar), along with bindings for the `Configuration` being processed (usually used for obtaining the configuration's `StrSubstitutor` for variable interpolation) and the currently processed `Node`.

Configurations are parsed as a tree of nodes where each node contains a name, list of attributes, child nodes, an optional value, which are processed recursively unless a plugin sets its `deferChildren` option to true which defers loading and binding of the children nodes of that plugin's node. This lazy loading option is primarily useful for advanced plugins like `RoutingAppender` which selectively load plugin configurations based on processed `LogEvent` data.

Later on, support for creating and configuring plugins via builder classes was added for simplifying default value strategies, adding new configuration options over time without maintaining several deprecated static factory methods, and making tests easier to write when directly referencing plugin instances. This added injection through fields of the builder class along with a separate static factory method on the plugin class to construct fresh builder instances.

While this strategy has worked well enough for configurations where all necessary plugin objects are configured through a config file, this has left much duplicate functionality around log4j-core for loading other types of plugins, and it has made it difficult to inject broader-scoped objects into narrower-scoped ones (e.g., injecting a singleton-scoped instance into a configuration-scoped instance). In order to simplify plugin handling and configuration loading, these related plugin loading systems should be unified into a consistent dependency injection API.

API Changes

Inject Annotations, Scopes, Qualifiers, and Factories

Plugin annotations are updated to support a dependency injection API similar to `javax.inject` which is sufficiently simple to model both existing injection strategies as well as new ones introduced by this system (such as method-based and constructor-based injection). To avoid third party dependencies, Log4j provides its own equivalent annotations to those in `javax.inject` for better integration with how Log4j already works. More specifically, log4j-plugins adds the following annotations to the `org.apache.logging.log4j.plugins` package:

- `@Inject` - marks a constructor, field, or method for injection. Classes that use this annotation can be obtained from an Injector which handles binding runtime instances to each of the injection points. Marking a no-arg method with `@Inject` is effectively an initialization method as these are invoked last. Injection starts with the constructor, then fields, then methods (methods with arguments executed first before no-arg methods). Inject methods must not be abstract.

    -
      ```
      class InjectExample {
              DependencyA dependencyA;
          @Inject DependencyB dependencyB;
               DependencyC dependencyC;

              @Inject
              InjectExample(DependencyA a) {
                      dependencyA = a;
              }

              @Inject
              void setC(DependencyC c) {
                      dependencyC = c;
              }

              @Inject
              void postConstruct() {
                      StatusLogger.getLogger().debug("Finished injecting everything");
              }
      }
      ```
- `@QualifierType` - marks an annotation as a *qualifier* annotation. Qualifiers are used for providing multiple variants of the same type which can be identified by the qualifier annotation. Qualifiers can be specified on a class, an inject method parameter, a factory method, and an inject field.
- `@Named` - qualifier for a named instance. This annotation can specify alias names, too (similar to the existing `@PluginAliases` annotation).

    -
      ```
      class QualifierExample {
              @Named String foo; // uses field name by default for value in @Named
              @Named({"primary", "legacy-name"}) String bar; // inline use of name plus aliases
      }
      ```

- `@ScopeType` - marks an annotation as a *scope* annotation. Scopes provide a strategy for determining the lifecycle of instances in its scope. Without a scope, instances are created fresh each time they're injected or obtained from `Injector`.
- `@Singleton` - scope type that maintains at most one instance for each qualified type. That is, every injected singleton instance and every singleton instance obtained from `Injector` will be the same instance.
- `@FactoryType` - marks an annotation as a *factory* type. Factory annotations are used for marking methods as factories for the returned instance of that method. This is intended to support existing plugin factory related annotations like `@PluginFactory` and `@PluginBuilderFactory`.
- `@Factory` - marks a method as a factory for instances it returns. Factory methods can be static or not, though instance factory methods may only be installed by installing an instance of the class with said factory method.

    -
      ```
      class FactoryExample {
              @Inject Dependency dep;

              @Factory
              ProducedInstance newInstance(@Named String name) {
                      return new ProducedInstance(name);
              }
      }
      ```

      Then, the load and install using an anonymous class for an installation module.

      ```
      Injector injector = DI.createInjector(new Object() {
              @Factory
              @Named
              String getName() { return "example"; }
      }, FactoryExample.class);

      var producedInstance = injector.getInstance(ProducedInstance.class);
      ```

Another API change is replacing existing `Builder<T>` classes with `java.util.function.Supplier<T>`. To support backward compatibility, when a static factory method returns an object that implements `Builder<T>` or `Supplier<T>`, then that instance is injected and used as a factory for its type `T`.

The Core plugin type is updated to use this injection API so that in addition to the existing support for injecting `@PluginElement`, `@PluginAttribute`, `@PluginBuilderAttribute`, `@PluginValue`, `@PluginNode`, and `@PluginConfiguration` instances, this is extended to support for injection via fields, methods, and constructors, along with injection of any other instances Injector has bindings for or knows how to create bindings on demand for. Classes that can have on-demand bindings are injectable classes which are classes with either one `@Inject` constructor or a no-args constructor. Implementations of `LogEventFactory` are a good example of injectable classes where the choice of class is configurable at runtime, though the dependency chain involved can be made explicit while removing boilerplate dependency injection code to where this class is relevant. An abbreviated example of what a `Supplier<LoggerConfig>` class may look like:

```
public class Builder implements java.util.function.Supplier<LoggerConfig> {
    // ...

    // note that methods with qualified parameters are implicitly @Inject
    public Builder withLevel(@PluginAttribute Level level) {
        this.level = level;
        return this;
    }

    public Builder withLoggerName(
            @Required(message = "Loggers cannot be configured without a name") @PluginAttribute String name) {
        this.loggerName = name;
        return this;
    }

    public Builder withRefs(@PluginElement AppenderRef[] refs) {
        this.refs = refs;
        return this;
    }

    public Builder withConfig(@PluginConfiguration Configuration config) {
        this.config = config;
        return this;
    }

    public Builder withFilter(@PluginElement Filter filter) {
        this.filter = filter;
        return this;
    }

    // need to specify @Inject here because LogEventFactory is an unqualified bean
    @Inject
    public Builder setLogEventFactory(LogEventFactory logEventFactory) {
        this.logEventFactory = logEventFactory;
        return this;
    }

    @Override
    public LoggerConfig get() {
        // ...
    }
}
```

Keys and Bindings

The `Key<T>` class provides a way to identify plugins by type, optional qualifier, and a name. Existing `@PluginAttribute` and `@PluginValue` annotations are natural qualifier types, though given that these annotations are duplicated in log4j-plugins and log4j-core, a more generic mechanism is needed to treat these as equivalent. This is where the name aspect of a `Key` comes in; all named-style qualifier annotations are treated as `@Named` qualifiers. The `ConfigurationInjector` and `ConfigurationBinder` API in log4j-plugins is replaced with a simpler strategy for supplying a (possibly converted) value for a particular `Node` instance. `Nodes` can be configured via `Injector` which is where general dependency injection occurs along with binding of provided configuration attributes in the `Node` instance. Annotation-handling strategies for configuration injection are replaced with a parsing strategy while strategies for binding are inlined into the general logic of `Injector`.

When processing a `Configuration` with a `Node` tree, the general logic for combining plugin instances is handled specially by different `Configuration` implementations. `AbstractConfiguration` should be updated to rely on dependency injection to form the full graph of plugin instances so that different configuration formats would only be responsible for transforming input configuration data into a tree of `Nodes` which get processed by dependency injection. Plugins that inject a `Configuration` can be refactored to inject whichever relevant instances are needed from the configuration instead of looking them up manually. For backward compatibility, the `Configuration` can be obtained from `Injector`.

No annotation processing updates are required here as the existing plugin annotations provide enough useful metadata on plugins to avoid loading them all at runtime. Plugin classes relying on `@Inject` are supported as soon as that class is referenced in the injection system as the injection constructor is discovered on demand.

An additional API is added for users to provide custom injection modules to allow programmatic configuration of other injectable instances. This may be useful for unifying some `ServiceLoader`-related APIs in Log4j. These modules can set up bindings for configurable classes outside the normal configuration system. For example, if injection begins from `Log4jLoggerContext`, then these modules can be used for programmatically configuring a `ContextSelector` and `ShutdownCallbackRegistry` instead of relying on optional system properties to specify a class name. Programmatic configuration from a `ServiceLoader`-created class simplifies class loading as direct references to classes can be compiled in directly. When injection begins from a `LoggerContext`, modules can customize classes that are global to a configuration or logger context. This also provides a natural place to define or override `ConfigurationFactory` instances. Other programmatic instances needed for various plugins can be more easily specified through this API.