Asynchronous Web Clients

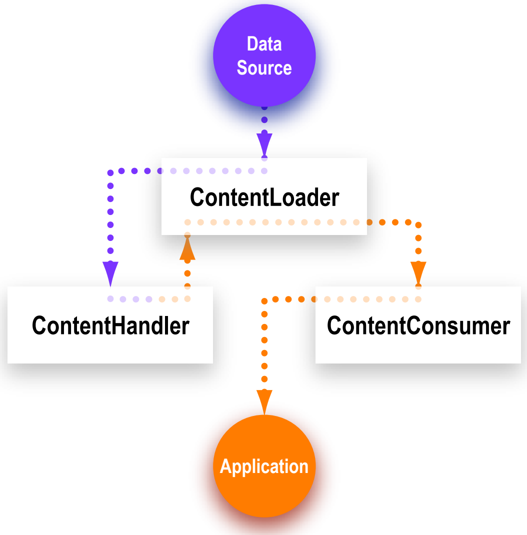
This article describes an approach to building web-facing applications that permit lightweight AJAX-style interactions, but use Java technology instead of JavaScript for increased richness, robustness, and control.

Overview

In an asynchronous web client, raw content is loaded and processed in the background using a different thread from the one that initiates the load. While raw content is processed in the background thread, processed chunks of it are asynchronously consumed by the client in the event handling thread. The basic design involves three cooperating objects, each responsible for a specific part of the interaction. First, a **content loader** is responsible for fetching content from a web service or data source after passing request parameters, if any, to the server. Second, a **content handler** is responsible for interpreting, filtering, and manipulating the fetched content to realize domain-specific objects from the content. Finally, a **content consumer** is responsible for assimilating the realized objects into the presentation.

Figure 1 shows the flow of content from a data source to an application through a content loader, content handler, and content consumer.

Figure 1: Content loader, handler, and consumer



The Content Troika

The content loader, content handler, and content consumer implement the `ContentLoader`, `ContentHandler`, and `ContentConsumer` interfaces, respectively. The classes that implement these interfaces hide considerable complexity behind their simple interfaces, and are largely independent of each other. For example, whether a content loader uses HTTP or loads content from a local file system has no bearing on how a content handler interprets that data, or how a content consumer uses that data.

Similarly, a content handler might process textual content using regular expressions, or it might process XML content using an XML processor, perhaps relying on prefabricated object realization components to extract domain-specific objects from the data stream, but the details don't affect the content loader and content consumer.

Finally, the content consumer is responsible for the ultimate disposition of the output produced from the fetched content by the content handler. How it chooses to incorporate that output into the presentation does not matter to the other two. Unlike the other two classes of objects, which can often leverage pre-fabricated components, a content consumer class almost invariably consists of application-specific code that interacts with user interface components, and feeds processed content to their data models.

Orchestration

In addition to loading content, a content loader is also responsible for the orchestration of its associated content handler and content consumer objects. Whenever the content loader is triggered, it tries to asynchronously fetch the content that it is responsible for, and hand it off to the content handler as soon as the content becomes available.

As the content handler processes the fetched data, it may incrementally realize *elementary* objects from that data and collect them into an *aggregate* object. Through a mechanism to be described later, the content loader periodically gets notified of the incremental progress along with chunks of output produced by the content handler. The content loader then hands over these chunks to the content consumer as they become available.

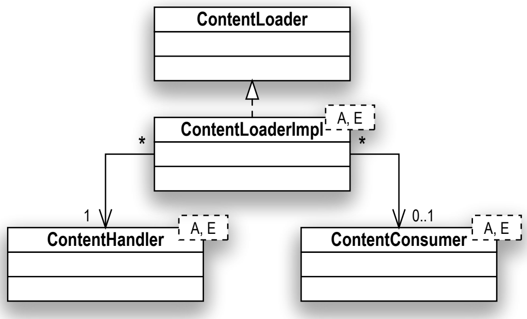
Finally, after the content handler has finished processing the fetched data, the content loader takes the aggregate object produced by the content handler, and hands it over to the content consumer for assimilation. The content handler and content consumer never communicate directly with each other, and are not even aware of each other's existence.

For this to work, the content loader, content handler, and content consumer must all agree on the *types* of the elementary and aggregate objects produced by the content handler, even though they are otherwise independent of each other.

Classifier Relationships

Of the three classes of objects in the content troika, only an implementation of the `ContentLoader` interface has references to the other two, while the rest are not even aware of the existence of the other objects. This is shown in Figure 2, where `ContentHandler` and `ContentConsumer` are navigable from `ContentLoaderImpl`, a fictitious class that implements the `ContentLoader` interface.

Figure 2: ContentLoader and related interfaces



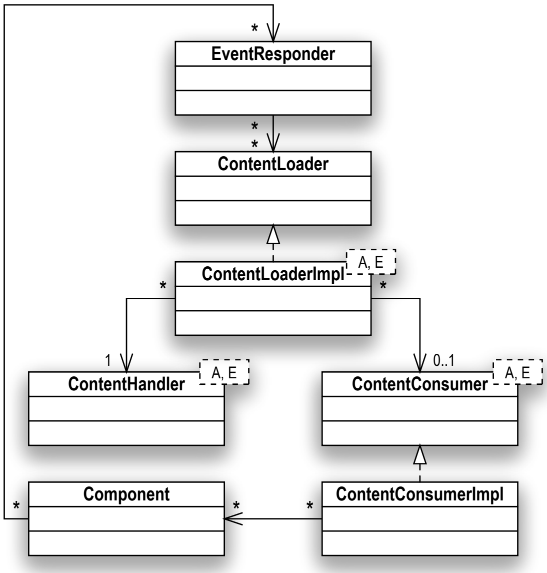
The figure also indicates that while a `ContentHandler` is required, a `ContentConsumer` is optional. The generic type parameters `A` and `E` indicate[1] the types of the *aggregate* and *elementary* objects, respectively, that may be realized by a `ContentHandler`.

Peripheral Interactions

Before a content loader can load anything, something must trigger its `load` method. A typical arrangement is to have an **event responder** object respond to a stimulus from one or more user interface components by triggering the content loader's `load` method. The trigger sets off a chain reaction that usually culminates in the content consumer interacting with some user interface component, and updating its data model.

Figure 3 shows the classes involved in these peripheral interactions. Note that components may be associated with any number of event responders, which, in turn, may be associated with any number of content loaders. Similarly, a content consumer may also update any number of components.

Figure 3: ContentLoader and peripheral classes



The component that is updated by the content consumer might or might not be the same instance that provided the initial stimulus to the event responder. Complex cascading arrangements and feedback loops are possible simply by linking different component instances to different event responder instances. We will see an example of a cascading arrangement later on in the article.

---

[1] The meaning and order of these generic type parameters are the same as those in the `SwingWorker` class. For the purpose of this discussion, you don't need to understand how `SwingWorker` works. See http://download.java.net/jdk6/docs/api/javax/swing/SwingWorker.html if you are curious. `ContentLoader`, `ContentHandler`, and `ContentConsumer` effectively distribute the various aspects of `SwingWorker` over three separate abstractions, each of which may evolve independently.

ContentLoader

The `ContentLoader` interface is very simple, consisting of only two methods as shown in Listing 1.

Listing 1: ContentLoader interface

```
public interface ContentLoader {
    void load(String... dynamicParams);
    boolean isLoading();
}
```

A content loader may only load content from a pre-designated data source. A content loader may pass zero or more parameters to the data source with each invocation of the `load` method. The values of these parameters are determined dynamically by the event responder that triggers the `load` method. For example, an event responder that responds to combo box interactions may fetch the value of the item selected in the combo box, and pass it to the `load` method. The content loader then starts a new thread to fetch data asynchronously from the designated data source, passing to it the parameters received by its `load` method.

The `isLoading` method may be used to ascertain if the content loader is currently loading its content. If the `load` method is invoked while the content loader is still loading content from an earlier invocation, the content loader cancels the pending request and initiates a new one. Event responders that automatically trigger the `load` method in response to timer events, for example, may use the `isLoading` method to throttle their request rate. This is to avoid overwhelming the content loader with frequent new requests, each of which might cancel the previous unfinished request. Without this protection, if new requests *always* came in before previous ones completed, nothing would *ever* get loaded!

HttpContentLoader

A variety of `ContentLoader` implementations are possible. In this section, we discuss `HttpContentLoader`, which is responsible for making Http requests and fielding responses to those requests.

`HttpContentLoader` always conducts Http interactions in a different thread from the one in which its `load` method was called. Applications that wish to monitor the pulse of these asynchronous interactions may configure an instance of `HttpContentLoader` with an object that implements the `ProgressTracker` interface described later. Listing 2 shows the constructor declaration for `HttpContentLoader`.

Listing 2: Constructor declaration for HttpContentLoader

```
public class HttpContentLoader<A,E> implements ContentLoader {
   public HttpContentLoader(
      ContentHandler<A,E> handler, ContentConsumer<A,E> consumer,
      ProgressTracker progTracker, String serviceURI, HttpMethod method,
      String ...staticParams) {
      // ...
   }

   // ...
}
```

In addition to a `ContentHandler` and a `ContentConsumer` object, the constructor takes an instance of `ProgressTracker`, the request URI, the `HttpMethod` (GET, POST, and so on), and request parameters that remain unchanged for every invocation of the `load` method. Of these arguments, a non-`null` value is required only for the content handler, the service URI, and the http method. The formal type parameters `A` and `E` in the declaration match those of `ContentHandler` and `ContentConsumer` to ensure type safety as realized objects flow from content handler to content loader to content consumer.

The `ProgressTracker` interface has a single method called `completed`, which indicates the percent of work that has been completed. `HttpContentLoader` calls this method with the value *zero* when it starts an asynchronous interaction, and again with the value *100* when the interaction is completed. This allows applications to display a progress indicator in the user interface to inform users that an asynchronous task of indefinite duration is currently under way. Later, we describe `JFlowIndicator`, a prefabricated component, that can be used to unobtrusively inform the user whenever a `ContentLoader` attempts to load some content asynchronously.

Using `HttpContentLoader` is quite simple. Listing 3 shows a tiny application that uses `HttpContentLoader` to load some content, and hand it over to a simple content handler called `Echo` (described later) that just echoes the loaded content. This is an utterly useless application, but it does show how little is required to use `HttpContentLoader`.

Listing 3: Simple demo of Http GET with HttpContentLoader

```
public class HttpGetDemo {
   public static void main(String[] args) {
      ContentLoader contentLoader = new HttpContentLoader<String, String>(
         new Echo(), null, null, “http://www.openmarkup.net”, GET);
      contentLoader.load();
   }
}
```

The application first instantiates `HttpContentLoader`, configuring it with a `ContentHandler` called `Echo`, the URI string pointing to the location from which to fetch content, and the enumeration constant `GET`, which actually describes an `HttpMethod`. This example does not have a content consumer, and does not care about tracking progress.

Static request parameters that never change from one invocation of `load` to the next are typically specified in the constructor for `HttpContentLoader` following the `HttpMethod` parameter. Additional dynamic parameters, whose values might change from one invocation of `load` to the next, are passed in through the `load` method. In this example, there are no request parameters, either static or dynamic.

The content handler class `Echo`, described in detail later, implements the parameterized type `ContentHandler<String, String>`. The first actual parameter used in this invocation of the generic `ContentHandler` declaration promises that the type of object, if any is realized from the fetched content, is guaranteed to be `String`. The second actual parameter declares that the type of objects, if any are realized *incrementally* from the fetched content, is also guaranteed to be `String`. To accommodate an instance of `Echo`, the declaration of `HttpContentLoader` must itself be invoked with the same actual type parameters `<String, String>`.

Listing 4 shows a simple demo that uses Http POST with `HttpContentLoader`. This example fetches XML content from a web service, passing in a single request parameter to the `load` method; the parameter value may vary with each invocation of the method.

Listing 4: Simple demo of Http POST with HttpContentLoader

```
public class HttpPostDemo {
  public static void main(String[] args) {
    ContentLoader contentLoader = new HttpContentLoader<String, String>(
      new Echo(), null, null,
      "http://www.webservicex.net/uszip.asmx/GetInfoByState", POST);
    contentLoader.load("USState=AK"); // params may change with each invocation

    try {Thread.sleep(10000);} catch (Exception ex) {}   // sleep 10 seconds
    System.out.println("==========");

    contentLoader.load("USState=AS"); // params may change with each invocation
  }
}
```

The call to `Thread.sleep` tries to ensure that the second invocation of the `load` method is not made until the first one completes. Without this guard, the first request might be cancelled if it has not already completed when the second request is made. This is only an issue in this contrived example. In real applications, where the `load` method is triggered in response to user actions, there is no need to throttle load requests like this.

ContentHandler

A `ContentHandler` is responsible for interpreting, filtering, and manipulating arbitrary content, and publishing the results of its computations to the outside world. The most common use of a `ContentHandler` is to process textual content and realize domain-specific objects from that content. The `ContentHandler` interface declares a single method called `getObject`, as shown in Listing 5.

Listing 5: ContentHandler interface

```
public interface ContentHandler<A,E> {
  A getObject(String systemID, MediaType type, int size, Map<String,String> parm,
    InputStream stream, OutputChannel<E> out);
}
```

`ContentHandler` is a generic interface that takes two type parameters, the first of which indicates the type of object that its `getObject` method may return when it has finished processing the supplied content. The second type parameter indicates the type of objects that the `getObject` method may periodically output in chunks as it processes the content.

The `getObject` method takes six arguments — the base URI or system ID against which relative URI references may be resolved, the `MediaType` of the received content, the size of content, a `Map` of additional <key, value> parameters (for future expansion), the `InputStream` from which the content may be read, and an `OutputChannel` to which progress information and intermediate results may be written.

Not all implementations of `ContentHandler` incrementally publish the results of their computation in chunks, or even return an actual object once the processing is finished. The `ContentHandler` interface only represents that *if* anything is returned from the `getObject` method, it will be of the same type as the first *actual* type parameter, and that *if* any chunks are incrementally provided to the progress observer, the type of each chunk will match the second *actual* type parameter.

Listing 6 shows the `Echo` class implementing the `ContentHandler` interface.

Listing 6: Simple implementation of ContentHandler

```java
public class Echo implements ContentHandler<String, String> {
    public String getObject(String systemID, MediaType contentType, int length,
        Map<String,String> params, InputStream stream, OutputChannel<String> o) {
        if (contentType.isText()) {   // don’t bother echoing non-textual content
            try {
                BufferedReader reader = new BufferedReader(
                                                    new InputStreamReader(stream));
                String line;
                while ((line = reader.readLine()) != null) {
                    System.out.println(line);  // “Echo” the line to System.out
                }
                reader.close();
            }
            catch (Exception ex) {
                System.out.println(ex);
            }
        }
        return null;
    }
}
```

Classes implementing the `ContentHandler` interface have *intimate* knowledge of the format of the content they expect to receive for interpretation and processing. Typically, they use that knowledge to processes the supplied data in application-specific ways in order to realize domain-specific object representations of that data.

MediaType

`MediaType` represents a media type, which is used in several Internet protocols to identify the type of content such as text, image, audio, and video. The `MediaType` class provides methods to get the type, subtype, and parameters, if any, for the content type. For added convenience, it also includes the `isText` method to check the textuality of the received content, and the `isXML` method to check whether it is an XML derivative.

Listing 7 shows the implementation of the `isText` and `isXML` methods in a snippet from the `MediaType` class.

Listing 7: isText() method of MediaType

```java
public boolean isText() {
    return getType().toLowerCase().equals("text") || isXML();
}

public boolean isXML() {
    return XML_SUBTYPE_PATTERN.matcher(getSubtype()).matches();
}

private final static Pattern XML_SUBTYPE_PATTERN = Pattern.compile(
        "^(?:(?:(?:.+\\+)?xml)|(?:xml-.+))$", CASE_INSENSITIVE); // per RFC 3023
```

OutputChannel

An `OutputChannel` passively waits for the `getObject` method to provide progress updates and chunks of data as they become available A `ContentHandler` knows nothing about the provenance of the `OutputChannel`. All it knows is that the channel is capable of receiving incremental chunks of output, if any, as well as periodic updates on the percentage of work completed, if known. Listing 8 shows the `OutputChannel` interface.

Listing 8: OutputChannel

```
public interface OutputChannel<E> extends ProgressTracker {
    void produced(E... chunks);
}
```

We now develop a `ContentHandler` that filters the received content and publishes incremental chunks of "accepted" data to the output channel as the input data is being processed.

Example: Screen Scraper

The example shown in Listing 4 calls the `load` method of a `ContentLoader` with a hard-coded query string that passes a two-letter USA state code to a web service to obtain basic information about that state. What we want to do now is build a list of state names and abbreviations that we can show in a Swing component such as `JList` or `JComboBox`.

Before we start working on our `ContentHandler`, we have a minor detail to discuss. Recall that the query parameter sent to the web service contains a two-letter state code, but we want to display the full state name in the list or combo box. This means that a list item must encode both, the state name and the state code. For this, we use a prefabricated class called `TitledItem` that allows us to attach a *value* in addition to a *title* for an item.

For the input data to our `ContentHandler`, we look to the United States Postal Service web site at http://www.usps.com/ncsc/lookups/usps_abbreviations.html.

A quick examination of the content loaded from the Postal Service web site using a browser's "View Source" feature reveals that it consists of capricious HTML, with dangling elements (for example, META elements have no closing tag), and elements whose start and end tags don't use the same case consistently. This means that we can't process this with an XML processor, and must resort to old-fashioned regular expressions text processing to extract anything useful for a list or combo box data model. Had the content been some kind of XML document, additional interesting processing options would have opened up immediately.

Recall that a `ContentHandler` has *intimate* knowledge of the format of the content it expects to receive for interpretation and processing. It uses that knowledge to processes the supplied data in application-specific ways. In this example, the job of our `ContentHandler` is to filter the received content, extracting state names and abbreviations from it and throwing away everything else. The `getObject` method of the `ContentHandler` must create `TitledItem` instances from the extracted information, and provide them to the `OutputChannel` as soon as they are extracted. The `getObject` method must also return an array containing *all* of the `TitledItem` instances when the entire input has been processed.

Listing 9 shows the definition of our class called `StateFinder`, which implements the parameterized type `ContentHandler<TitledItem[], TitledItem>`.

Listing 9: Complete implementation of StateFinder

```
public class StateFinder implements ContentHandler<TitledItem[], TitledItem> {
   public TitledItem[] getObject(String systemID, MediaType type, int length,
      Map<String,String> parms, InputStream stream, OutputChannel<TitledItem> o) {
      try {
         BufferedReader reader = new BufferedReader(
                                             new InputStreamReader(stream));
         List<TitledItem> states = new ArrayList<TitledItem>();
         String line;
         while ((line = reader.readLine()) != null) {
            Matcher m = STATE_NAME_PATTERN.matcher(line);
            if (m.find()) {     // state name is in group 1
               String stateName = m.group(1);
               line = reader.readLine();
               m = STATE_ABBR_PATTERN.matcher(line);
               if (m.find()) {  // now state abbreviation is in group 1
                  TitledItem   item = new TitledItem(m.group(1), stateName);
                  o.produced(item);      // drop it in the OutputChannel
                  states.add(item);
               }
            }
         }
         reader.close();
         return states.toArray(new TitledItem[states.size()]);
      }
      catch (Exception ex) {
         System.out.println(ex);
      }
      return null;
   }
   private final static Pattern STATE_NAME_PATTERN = Pattern.compile(
                  "headers=\\x22st\\x22.+<TT>(.+)</TT>", Pattern.CASE_INSENSITIVE);
   private final static Pattern STATE_ABBR_PATTERN = Pattern.compile(
                  "<TT>(.+)</TT>", Pattern.CASE_INSENSITIVE);
}
```

At the heart of this `ContentHandler` are two `Pattern` objects shown at the bottom. The first one looks for a line containing the string `headers=“st”`, followed by one or more characters, followed by the start and end tags `<TT>` and `</TT>` with the state name embedded between the tags. The second one is similar, but less restrictive, and tries to capture the state abbreviation embedded between the `<TT>` and `</TT>` tags.

Before it starts reading the content, the `getObject` method allocates an instance of the parameterized type `List<TitledItem>` to accumulate instances of `TitledItem` that are realized from the information we scrape together from the HTML content.

The `getObject` method then enters a loop, reading one line at a time, trying to find sets of two consecutive lines, where `STATE_NAME_PATTERN` matches in the first line and `STATE_ABBR_PATTERN` matches in the second line. Once both matches are found, it creates a new instance of `TitledItem`, drops it in the output channel, while simultaneously adding it to the list of states allocated earlier.

Once the entire content has been processed, the `getObject` method converts the item list from a `List<TitledItem>` to a `TitledItem[]` array and returns that array as its result, as promised. That is all there is to it!

This example can easily be adapted to handle any kind of textual content, such as tab- or comma-separated tabular data. When input data consists of well-structured markup (as in XHTML documents), additional options become available for processing that data, but the general approach remains the same.

Note that the class that implements `ContentHandler` does not know where the `OutputChannel` comes from, nor does it know anything about the `ContentLoader` interface. The mystery behind `OutputChannel` is easy to explain.

Since the class implementing the `ContentLoader` interface mediates between a content handler object and a content consumer object, it is natural for the content loader to set up the output channel as a temporary staging area from where it can grab chunks of output and give them to the content consumer in a thread-safe manner. Once the content loader has set up the output channel, it passes that object's reference into the `getObject` method of the content handler. Then, as the content handler drops objects into the channel, the content loader cascades them to the content consumer.

We now discuss content consumers, and show how the `StateFinder` class that we developed in this section can be used with a content consumer we develop in the next section.

ContentConsumer

A content consumer is responsible for the ultimate disposition of the output produced by the content handler, but it knows nothing about the existence of the content handler. Neither does it know about who invokes its services. The `ContentConsumer` interface is shown in Listing 10.

Listing 10: ContentConsumer interface

```
public interface ContentConsumer<A,E> {
    void prepare();
    void consume(E... chunks);
    void assimilate(boolean cancelled, A aggregate);
}
```

Unbeknown to a `ContentConsumer`, implementations of `ContentLoader` invoke the `prepare` method of a `ContentConsumer`, *before* invoking the `getObject` method of a `ContentHandler`.

The `prepare` method alerts the `ContentConsumer` to prepare itself for consuming chunks of processed data. Typically, the `ContentConsumer` initializes its internal state, if necessary, at this time. This is also the place where a content consumer must record its internal state in a memento, for example, to implement **checkpointing** in an optimistic design.

The `consume` method incrementally brings in one or more chunks of processed data, which the `ContentConsumer` must incorporate into the application as it deems fit. A content consumer may choose to ignore the periodic calls to its `consume` method, and simply consume the cumulative date in one shot in its `assimilate` method.

The `assimilate` method marks the final step in the content consumption process. The first parameter indicates whether upstream content processing was prematurely cancelled or not. If the value of the first parameter is `true`, the method may try to perform a rollback to the last checkpoint, or take evasive action, such as feeding some default values to the application. If all is well, the second parameter refers to an object containing the cumulative results of upstream processing since the `prepare` method was last called.

An implementation of `ContentConsumer` that uses both, the `consume` method and the `assimilate` method, must be careful not to consume the same data twice — once in the `consume` method, and again in the `assimilate` method.

To make it easier to define implementations of the `ContentConsumer` interface, we provide an abstract class called `AbstractContentConsumer`, which provides empty bodies for each of the methods of the `ContentConsumer` interface.

We now develop a simple implementation of `ContentConsumer` by extending the `AbstractContentConsumer` class to populate a `JComboBox` with objects of a generic type. So we define a class, called `ComboBoxUpdater`, whose declaration contains a formal type parameter `T` which allows us to specify the actual type of objects that the associated `JComboBox` instance may contain.

`ComboBoxUpdater<T>` extends `AbstractContentConsumer<T[], T>`, declaring that its `assimilate` method can accept an aggregate object of type `T[]`, while its `consume` method can accept zero or more chunks of type `T`. Listing 11 shows the implementation of this class.

Listing 11: Simple implementation of ContentConsumer

```
public class ComboBoxUpdater<T> extends AbstractContentConsumer<T[], T> {
    public ComboBoxUpdater(JComboBox comboBox) {
        this.comboBox = comboBox;
    }

    @Override public void assimilate(boolean cancelled, T[] items) {
        if (!cancelled && (items != null) && (items.length > 0)) {
            comboBox.setModel(new DefaultComboBoxModel(items));
        }
    }

    private final JComboBox comboBox;
}
```

The constructor for this class accepts an instance of `JComboBox`, and saves that in a private final member for the life of the content consumer instance. Later, whenever the `assimilate` method is called, it updates this combo box with a fresh data model constructed from the array of items received by that method.

Throughout our discussion we have explicitly maintained the separation of concerns between `ContentHandler` and `ContentConsumer` by having separate classes implement the interfaces. However, in some situations, the task of interpreting the raw data and the task of incorporating the processed data into the application may be inextricably intertwined. To cope with such situations, you may implement the `ContentHandler` and `ContentConsumer` interfaces in the same class.

We are now ready to build a small application that integrates the `ContentHandler` we developed in the last section and the `ContentConsumer` we developed in this section.

Integration

Once you understand how `ContentLoader`, `ContentHandler`, and `ContentConsumer` are implemented and used, it is surprisingly easy to integrate them into a working application.

Listing 12 shows a simple application that integrates the classes we have built so far. For additional variety, it also throws in a `JFlowIndicator`, although that is not strictly required for the application to function.

Listing 12: United States of America

```java
public class UnitedStates {
    public static void main(String[] args) {
        JComboBox        states = new JComboBox();
        JFlowIndicator   statesPulse = new JFlowIndicator();

        ContentLoader stateLoader = new HttpContentLoader<TitledItem[], TitledItem>(
            new StateFinder(), new ComboBoxUpdater<TitledItem>(states), statesPulse,
            “http://www.usps.com/ncsc/lookups/usps_abbreviations.html”, GET);

        stateLoader.load();

        // Build rest of the UI while content is loaded in a background thread
        JFrame frame = new JFrame("United States");
        JPanel statesPanel = new JPanel();
        statesPanel.add(statesPulse);
        statesPanel.add(states);
        frame.add(statesPanel);
        frame.setSize(640, 80);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

The code for the `UnitedStates` application should be pretty easy to understand. However, this is probably not what you had in mind if you were looking for AJAX-style interactions, where data is fetched and processed in the background, and delivered to the application *while the user is interacting with it*. The next section shows an example that does just that.

AJAX-style interactions

In this section, we construct an AJAX-style interactive **mashup** that combines data from the **Postal Service web site** shown in Listing 9 and data from the **USZip web service** shown in Listing 4.

The application sets up two combo boxes, first of which is populated using data from the Postal Service web site exactly as shown in Listing 12. The second combo box is populated using data from the USZip web service and is updated every time the user interacts with the first combo box. All interactions with the web service take place in a background thread without holding up the user interface.

To build this application, we already have all the components we need, except two. First, we need a `ContentHandler` that can extract city names from the content loaded from the USZip web service, just like the `StateFinder` class shown in Listing 9 was used to extract state names from the Postal Service web site. Let us call this class `CityFinder`. Second, we need an **event responder** that responds to a stimulus from the first combo box by triggering an update of the second combo box. Naturally, this event responder class will implement the `ActionListener` interface.

Example: Filtering content using regular expressions

We first build the `CityFinder` class, closely modeling it after the `StateFinder` class. As mentioned earlier, a `ContentHandler` has intimate knowledge of the input data. Running the example in Listing 4, we know that the content received from the USZip web service is legitimate XML data that describes a data set containing one or more records in a denormalized form.

Specifically, the records contain fields such as CITY, STATE, ZIP, and so on. The CITY field, for example, is marked up using <CITY> and </CITY> tags. We know that within a single data set, there may be multiple records for the same CITY name, but different ZIP codes.

The job of our `ContentHandler` is to filter this data set, and extract *unique* city names from it by ignoring the ZIP codes. The `getObject` method of the `ContentHandler` must provide the accepted city names to the `OutputChannel` as soon as they are discovered, while discarding the rest. The `getObject` method must also build a collection of unique city names, and return a sorted array containing those names when all of the input data has been processed.

Normally, filtering and sorting aspects are handled on the server. But it is quite common to apply this kind of processing to limited amounts of data on the client. This is also very useful when you don't have much control over the content you get back from the server (as in this example).

There are a variety of ways in which this problem can be approached. Because the data consists of well-formed XML, obviously one way to process this is to use an XML processor. However, a closer examination of the input data reveals some properties that allow us to use very simple text processing to accomplish our goals without resorting to an XML processor. Most important of these is that each city record can be found in its entirety on a separate line of text. Also, the input stream already contains records sorted by city name. As long as the `ContentHandler` can rely on these properties not changing, it can use simple regular expression processing on the input text, reading it in one line at a time.

Once we have decided on the overall approach, we can start defining the `CityFinder` class, a skeleton of which is shown in Listing 13.

Listing 13: Skeleton of CityFinder

```java
public class CityFinder implements ContentHandler<String[], String> {
   public String[] getObject(String systemID, MediaType contentType, int length,
      Map<String,String> params, InputStream stream, OutputChannel<String> o) {
      return null;  // temporary
   }
}
```

A notable difference between the `CityFinder` class and the `StateFinder` class is that they invoke the generic `ContentHandler` declaration with different *actual* parameters.

As shown in Listing 13, the `CityFinder` class implements the parameterized type `ContentHandler<String[], String>`, declaring that its `getObject` method will periodically publish `String` objects while processing the input, and return a `String[]` when it has finished processing the input.

It is now time to add meat to the skeleton. Listing 14 shows the full body of `CityFinder`.

Listing 14: Complete implementation of CityFinder

```java
public class CityFinder implements ContentHandler<String[], String> {
   public String[] getObject(String systemID, MediaType contentType, int length,
      Map<String,String> params, InputStream stream, OutputChannel<String> o) {
      try {
         BufferedReader reader = new BufferedReader(
                                                  new InputStreamReader(stream));
         Set<String> cities = new LinkedHashSet<String>();
         String line;
         while ((line = reader.readLine()) != null) {
            Matcher m = CITY_PATTERN.matcher(line);
            if (m.find()) {
               String city = m.group(1);

               // if not duplicate, notify progress observer of produced item
               if (cities.add(city)) {    // add item to aggregate; detect dup.
                  o.produced(city);   // provide unique item to observer
                  //o.completed(0); indeterminate -- can't set %completed
               }
            }
         }
         reader.close();
         return cities.toArray(new String[cities.size()]);
      }
      catch (Exception ex) {
         System.out.println(ex);
      }
      return null;
   }
   private final static Pattern CITY_PATTERN = Pattern.compile(
                  "<CITY>(.+)</CITY>");
}
```

The two main objects driving this `ContentHandler` are the `Pattern` object, which is used to extract city names from the input, and the `LinkedHashSet` object, which is used to accumulate unique city names. The pattern is designed to look for city names consisting of at least one character surrounded by `<CITY>` and `</CITY>` tags.

Next, we use a `Set<String>`, or more specifically, a `LinkedHashSet<String>` to store our city names because we want to exploit the fact that `Set` never allows duplicate items, as well as the fact that `LinkedHashSet` returns elements in the same order as they were added. This makes it very convenient to weed out duplicate city names while preserving the already sorted order of city names.

We process the input by reading it in one line at a time, and try to match it against `CITY_PATTERN`. If a match is found, we extract the city name from the first and only captured group of the `Matcher`. We then try to add the newly found name to the set of city names. If the name was never seen before, we notify the `OutputChannel` by calling its `produced` method.

After we have finished processing the input, we convert the set of city names into an array and return it as the result of the `getObject` method.

EventResponder

An event responder is the last missing piece in our quest for full round-trip asynchronous interaction between our rich client and a web service. Fortunately, it is also the simplest, if you already know how to write event listeners in Swing.

Recall that an event responder may trigger the `load` method in any number of content loaders. Since most event responders contain application-specific code anyway, you may either roll your own from scratch, or simply extend the abstract `EventResponder` class shown in Listing 15 if you are looking for an easy way to manage interactions with these content loaders.

Listing 15: EventResponder

```java
public abstract class EventResponder {
    protected EventResponder(ContentLoader... targets) {
        this.targets = targets;
    }

    public void trigger(boolean completePrevious, String... dynamicParams) {
        if (targets != null) {
            for (ContentLoader target : targets) {
                if (target != null) {
                    if (!completePrevious || !target.isLoading()) {
                        target.load(dynamicParams);
                    }
                }
            }
        }
    }

    private final ContentLoader[] targets;
}
```

The `boolean` argument to the `trigger` method tells it whether to wait for the completion of the previous request to a `ContentLoader`, or to go ahead and trigger a `load` right away. If the value of this argument is `true`, the `trigger` method will *not* trigger the `load` method for any content loader that is still loading content. The rest of the arguments, if any, are simply passed through to the `load` method on each invocation.

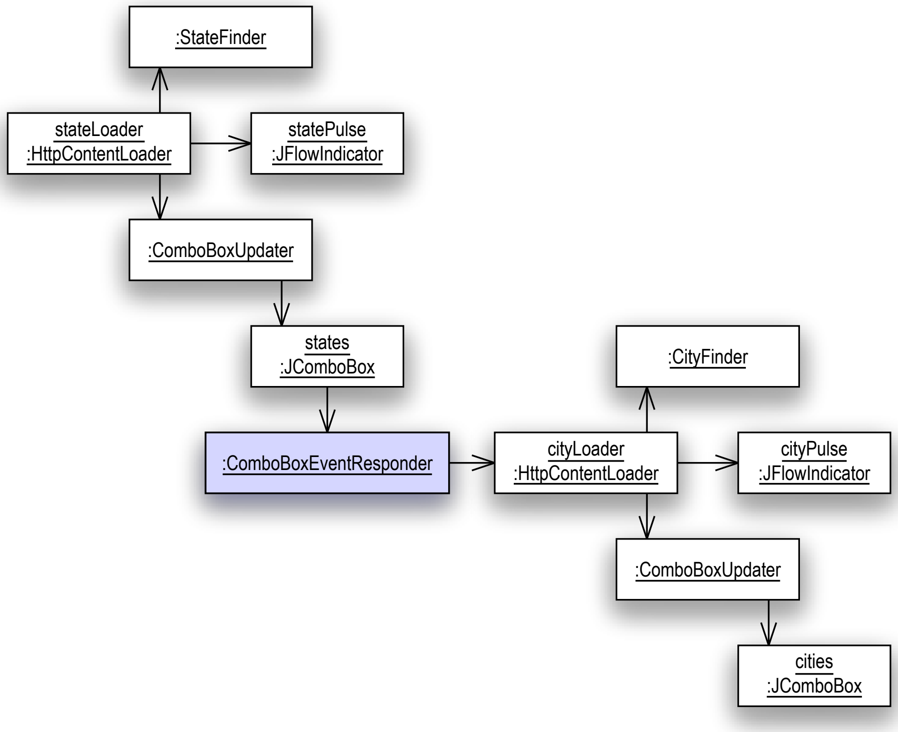
A concrete event responder may respond to stimuli from any number of components, not all of which need be of the same type. To accommodate a variety of components, an event responder may have to implement a variety of `EventListener` interfaces depending on the types of events that it responds to. The `ComboBoxEventResponder` class, for example, shown in Listing 16, extends the abstract `EventResponder` class and implements the `ActionListener` interface.

Listing 16: ComboBoxEventResponder

```
public class ComboBoxEventResponder extends EventResponder
    implements ActionListener {
    public ComboBoxEventResponder(String queryItemKey, ContentLoader ...targets) {
        super(targets);
        this.queryItemKey = queryItemKey;
    }
    public void actionPerformed(ActionEvent ev) {
        Object source = ev.getSource();
        if (source instanceof JComboBox) {
            Object item = ((JComboBox) source).getSelectedItem();
            if (item instanceof TitledItem){
                // no need to wait for the previous load request to complete!
                trigger(false, queryItemKey + "=" + ((TitledItem) item).getValue());
            }
        }
    }
    private final String queryItemKey;
}
```

Figure 4 shows a cascading arrangement in which a set of `ContentLoader`, `ContentHandler`, `ContentConsumer`, `ProgressTracker`, and `JComboBox` links to a completely different set of similar objects through an instance of our `ComboBoxEventResponder` class.

Figure 4: Object structure for the mashup



The object structure shown in Figure 4 reveals a repeatable pattern in which a `ContentLoader`, `ContentHandler`, `ContentConsumer`, and `ProgressTracker` collaborate to load raw content, process the content, integrate the processed content into a user interface component, and provide feedback to the user while all of this happening. Listing 17 shows how we translate this object structure into the code for our mashup.

Listing 17: UnitedStatesMashup

```
public class UnitedStatesMashup {
   public static void main(String[] args) {
      JComboBox        states = new JComboBox();
      JFlowIndicator   statePulse = new JFlowIndicator();
      ContentLoader    stLoader = new HttpContentLoader<TitledItem[], TitledItem>(
         new StateFinder(), new ComboBoxUpdater<TitledItem>(states), statePulse,
         “http://www.usps.com/ncsc/lookups/usps_abbreviations.html”, GET);
      stLoader.load(); // manually trigger this at startup

      // Build rest of the UI while states are loaded in another thread
      JComboBox        cities = new JComboBox();
      JFlowIndicator   cityPulse = new JFlowIndicator();
      ContentLoader    ctLoader = new HttpContentLoader<String[], String>(
         new CityFinder(), new ComboBoxUpdater<String>(cities), cityPulse,
         "http://www.webservicex.net/uszip.asmx/GetInfoByState", POST);

      ComboBoxEventResponder  responder = new ComboBoxEventResponder(
                                               "USState", ctLoader);
      states.addActionListener(responder); // responder implements ActionListener

      JFrame frame = new JFrame("United States Mashup");
      JPanel root = new JPanel();
      JPanel statesPanel = new JPanel();
      statesPanel.add(statePulse);
      statesPanel.add(states);
      JPanel citiesPanel = new JPanel();
      citiesPanel.add(cityPulse);
      citiesPanel.add(cities);
      root.add(statesPanel);
      root.add(citiesPanel);
      frame.add(root);
      frame.setSize(640, 80);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```

Note that `stLoader` and `ctLoader` invoke the generic `HttpContentLoader` declaration with different *actual* parameters because those are what their respective `ContentHandler` classes (`StateFinder` and `CityFinder`) require. The generic `ComboBoxUpdater` declaration is also invoked with different actual parameters for the same reason. Finally, `stLoader` loads content from an ordinary web site using the Http `GET` method, but `ctLoader` loads content from a REST-style web service using Http `POST` method.

The USZip web service used in this example expects a single request parameter whose name is `USState` and whose value is a two letter state code representing a state or territory of the United States. The parameter is passed to the web service in a query string of the form "*key*=*value*". Since the value of this parameter may change each time the `load` method of a `ContentLoader` is triggered, we construct the query string dynamically in the `actionPerformed` method of `ComboBoxEventResponder` using the selected `TitledItem` from combo box. The `load` method for each `ContentLoader` is itself triggered from within the `trigger` method of `ComboBoxEventResponder`.

For all of this to work, we construct the `ComboBoxEventResponder` instance with the query item key, and the content loader objects that will be triggered in response to a stimulus. In this example, there is only one content loader object, namely `ctLoader`, but there could be more.

The *raison d'être* for `statesPanel` and `citiesPanel` is to put their respective constituent components in close proximity to each other in order to *visually* indicate a link between a `JFlowIndicator` and the `JComboBox` instance that is being updated.

Threads

The code for this mashup is straightforward, and should be pretty easy to understand. What might not be obvious, though, is the number of threads that this example uses at various stages of program execution.

To begin with, the user interface is set up in the main application thread. The `load` method of `stLoader` is also invoked from the main application thread, but `stLoader` subsequently spawns a new background thread in which it loads raw content from the Postal Service web site. The `StateFinder` instance processes the raw content in the *same* background thread. Later, a `ComboBoxUpdater` instance updates its `JComboBox` instance in the AWT Event Dispatch Thread. While all of this is happening, `JFlowIndicator` performs its animation in yet another thread.

When a user interacts with the `states` `JComboBox`, the `actionPerformed` method of `ComboBoxEventResponder` is invoked in the Event Dispatching Thread. Unlike the `main` method, which invokes the `load` method of `stLoader` in the *main application thread*, the `trigger` method of `ComboBoxEventResponder` invokes the `load` method of `ctLoader` in the *Event Dispatch Thread*. The `ctLoader` object dutifully spawns a new background thread to load content from the USZip web service. Finally, the `cities` `JComboBox` instance is updated in the Event Dispatch Thread, while the associated `JFlowIndicator` instance displays its animated pulse indicator in yet another thread.

Fortunately, it is easy to avoid interference among the various threads because of the way the `ContentLoader`, `ContentHandler`, and `ContentConsumer` roles are factored into separate classes. An awareness of the threads in which the various objects participate is essential if you plan to write your own `ContentHandler` implementations.

Content Loader Factory

A content loader may load content from any of a variety of data sources. For example, one content loader may load resources from a local file system, another may load resources from a web server using the Http protocol, while a third may load content from *within* a local or remote Jar file. All of the examples we have seen so far are hard-wired to use the `HttpContentLoader` class. What we want is a content loader factory class that can automatically choose a content loader class and instantiate it for us based on the parameters that are supplied to the factory method. The `ContentLoaderFactory` class shown in Listing 18 does just that.

Listing 18: ContentLoaderFactory (public method declarations)

```
public class ContentLoaderFactory {
    public static <A, E> ContentLoader createContentLoader(
        ContentHandler<A, E> handler, ContentConsumer<A, E> consumer,
        ProgressTracker progTracker, String uriString) {
    }

    public static <A, E> ContentLoader createContentLoader(
        ContentHandler<A, E> handler, ContentConsumer<A, E> consumer,
        ProgressTracker progTracker, URI uri) {
    }

    public static <A, E> ContentLoader createContentLoader(
        ContentHandler<A, E> handler, ContentConsumer<A, E> consumer,
        ProgressTracker progTracker, URL url) {
    }

    public static <A, E> ContentLoader createContentLoader(
        ContentHandler<A, E> handler, ContentConsumer<A, E> consumer,
        ProgressTracker progTracker, File file) {
    }

    public static <A, E> ContentLoader createContentLoader(
        ContentHandler<A, E> handler, ContentConsumer<A, E> consumer,
        ProgressTracker progTracker, File file, MediaType mediaType) {
    }

    public static <A, E> ContentLoader createContentLoader(
        ContentHandler<A, E> handler, ContentConsumer<A, E> consumer,
        ProgressTracker progTracker, String uriString, HttpMethod method,
        String ...staticParams) {
    }

    public static <A, E> ContentLoader createContentLoader(
        ContentHandler<A, E> handler, ContentConsumer<A, E> consumer,
        ProgressTracker progTracker, URI uri, HttpMethod method,
        String ...staticParams) {
    }
}
```

`ContentLoaderFactory` has overloaded versions of the `createContentFactory` method, in which the requested protocol is usually obvious from the method signature. For example, versions that accept a `File` instantiate content loaders that load content from a local file system, whereas versions that accept an `HttpMethod` instantiate content loaders that load content using the Http protocol. In other versions, the method must inspect the value of certain parameters, such as the URI reference, to identify the protocol and instantiate an appropriate content loader class. The first three versions of the `createContentLoader` method shown in Listing 18 may be used to load content from a variety of data sources including a local file system, a remote web service using the Http protocol, or from inside a local or remote Jar file. When loading content over Http, the `GET` method is assumed in these versions of the `createContentLoader` method.

We conclude this article by showing a revised version of `UnitedStatesMashup` that uses `ContentLoaderFactory` instead of instantiating `HttpContentLoader` directly.

Listing 19: UnitedStatesMashup (using ContentLoaderFactory)

```
public class UnitedStatesMashup {
    public static void main(String[] args) {
        JComboBox        states = new JComboBox();
        JFlowIndicator   statePulse = new JFlowIndicator();
        ContentLoader    stLoader = ContentLoaderFactory.createContentLoader(
            new StateFinder(), new ComboBoxUpdater<TitledItem>(states), statePulse,
            “http://www.usps.com/ncsc/lookups/usps_abbreviations.html”);
        stLoader.load(); // manually trigger this at startup

        // Build rest of the UI while states are loaded in another thread
        JComboBox        cities = new JComboBox();
        JFlowIndicator   cityPulse = new JFlowIndicator();
        ContentLoader    ctLoader = new ContentLoaderFactory.createContentLoader(
            new CityFinder(), new ComboBoxUpdater<String>(cities), cityPulse,
            "http://www.webservicex.net/uszip.asmx/GetInfoByState", POST);

        // The rest of the code is unchanged from the original example
        // ...
    }
}
```

Although it might not be obvious, `ContentLoaderFactory` automatically instantiates the parameterized type `HttpContentLoader<TitledItem[], TitledItem>` for `stLoader`, but `HttpContentLoader<String[], String>` for `ctLoader` based on the actual arguments supplied to the two invocations of the `createContentLoader` method. Even though we no longer have to specify the type parameters for `HttpContentLoader` explicitly, type safety is never compromised.

Finally, note that it is no longer necessary to specify the Http method `GET` in the first invocation of `createContentLoader`, as that is the default method when content is loaded from a URI whose scheme is Http. As mentioned earlier, the biggest advantage of this factory method is that we can now change the URI string passed to this method to point to a local file, and the method will automatically instantiate a suitable content loader class.

Conclusion

This article shows one of many different approaches to building rich web clients. From a purely engineering standpoint, it is difficult to beat the type safety, robustness, degree of control, and sheer compactness in code size of such clients compared to browser-based solutions, despite the impressive strides made by the JavaScript community in recent years.