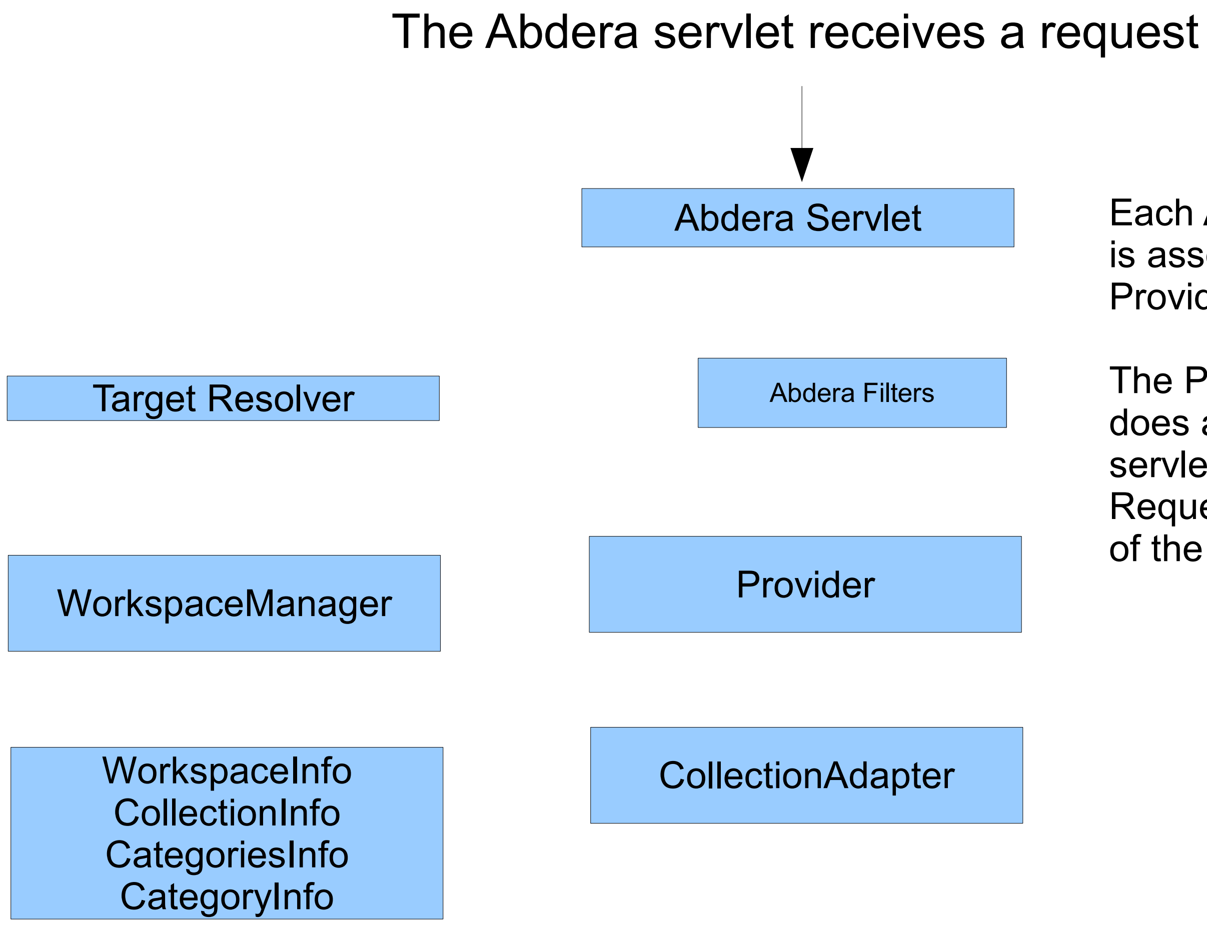
Implementing an Abdera Server

The 20-minute guide

The Key Components

- Abdera Servlet
- Provider
- Collection Adapter
- Workspace Manager
- Target Resolver
- Filter
- Workspace Metadata (WorkspaceInfo, CollectionInfo, CategoriesInfo, CategoryInfo)

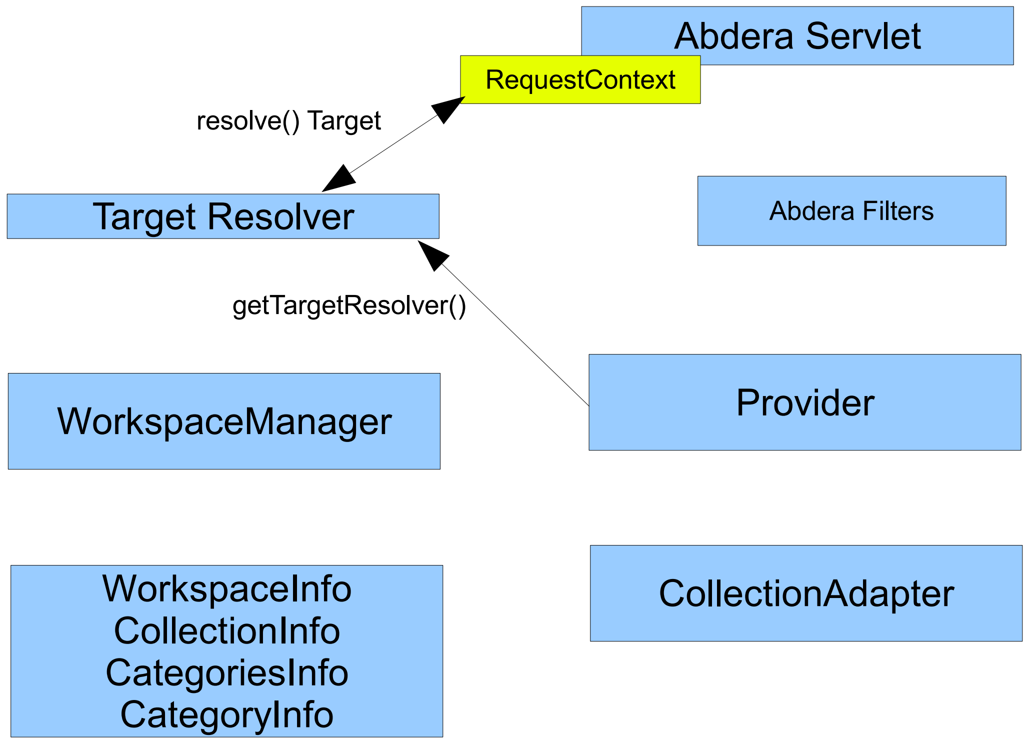
How requests are processed...



Each Abdera Servlet instance is associated with exactly one Provider implementation.

The Provider is the component that does all of the actual work. The servlet just handles the creation of the RequestContext and the serialization of the ResponseContext.

How requests are processed...

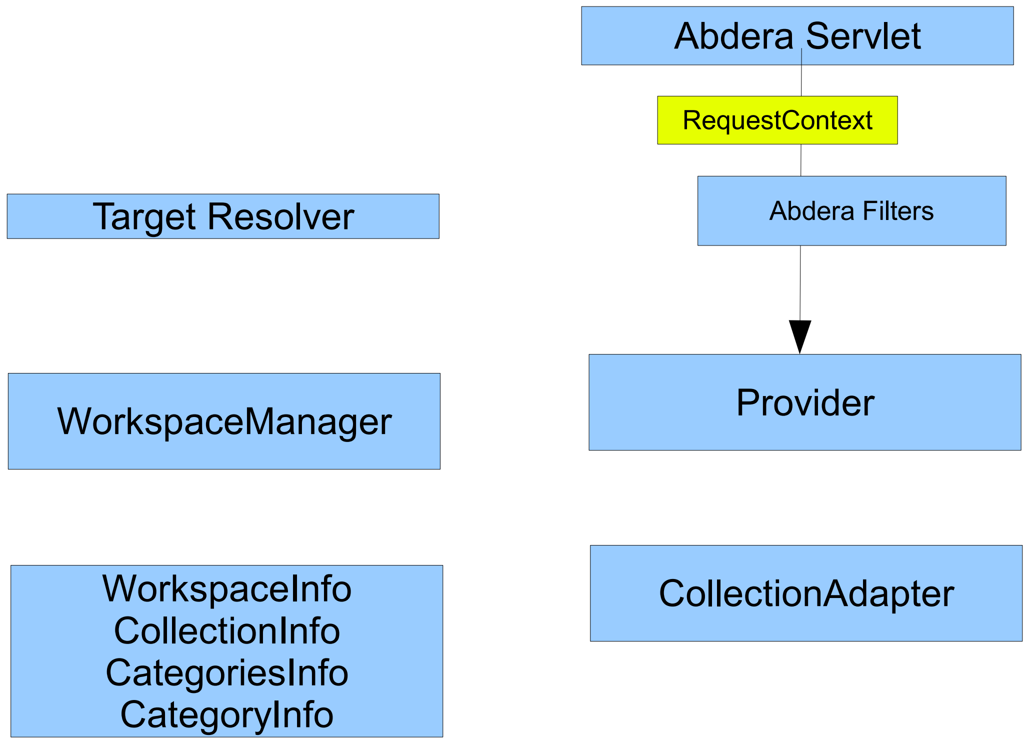


After receiving the request, the Abdera Servlet creates a RequestContext object, which is essentially a wrapper for the HttpServletRequest object.

The RequestContext asks the Provider for a “Target Resolver”, a special object that is used to determine the “Target” of the request.

Targets have an associated TargetType. These map to specifics kinds of Atompub artifacts like Collections, Entries, Service Documents, Categories Documents, etc.

How requests are processed...



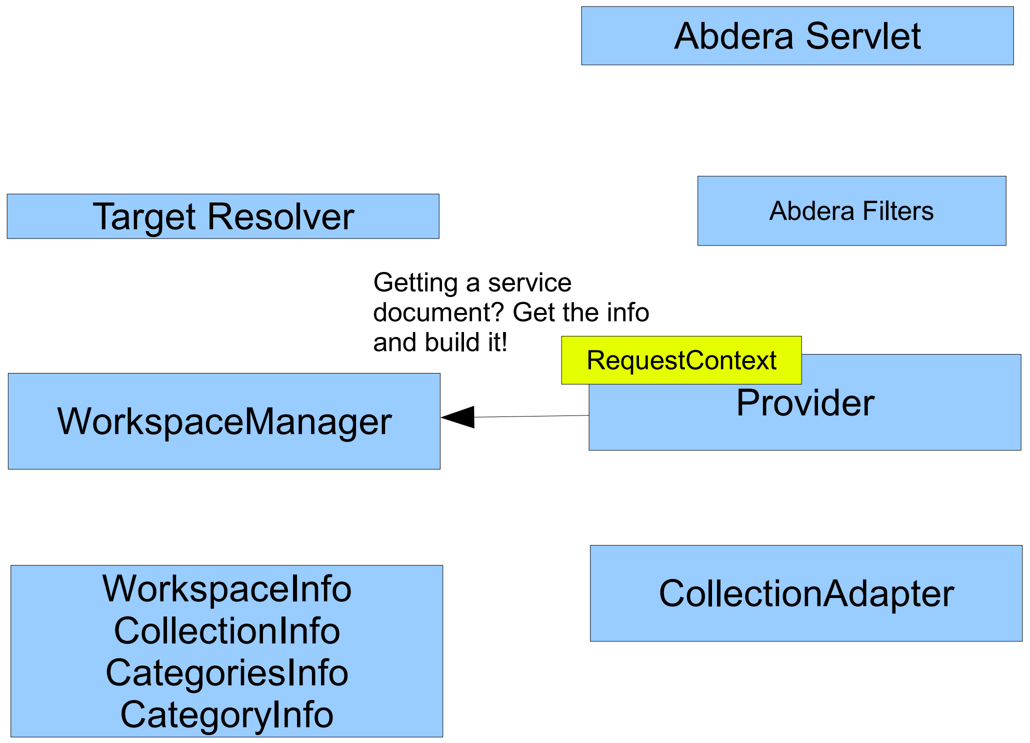
Once the RequestContext is created, the Abdera Servlet asks the Provider for a collection of Filters that should be invoked prior to passing the request off to the Provider.

The filters are invoked using a model identical to that used by Servlet Filters, with each filter in the chain forwarding the request on to the next filter in the chain until the last filter has been invoked.

Filters can either augment or intercept the request.

Once all of the filters have been invoked, the request is handed off to the Provider for processing

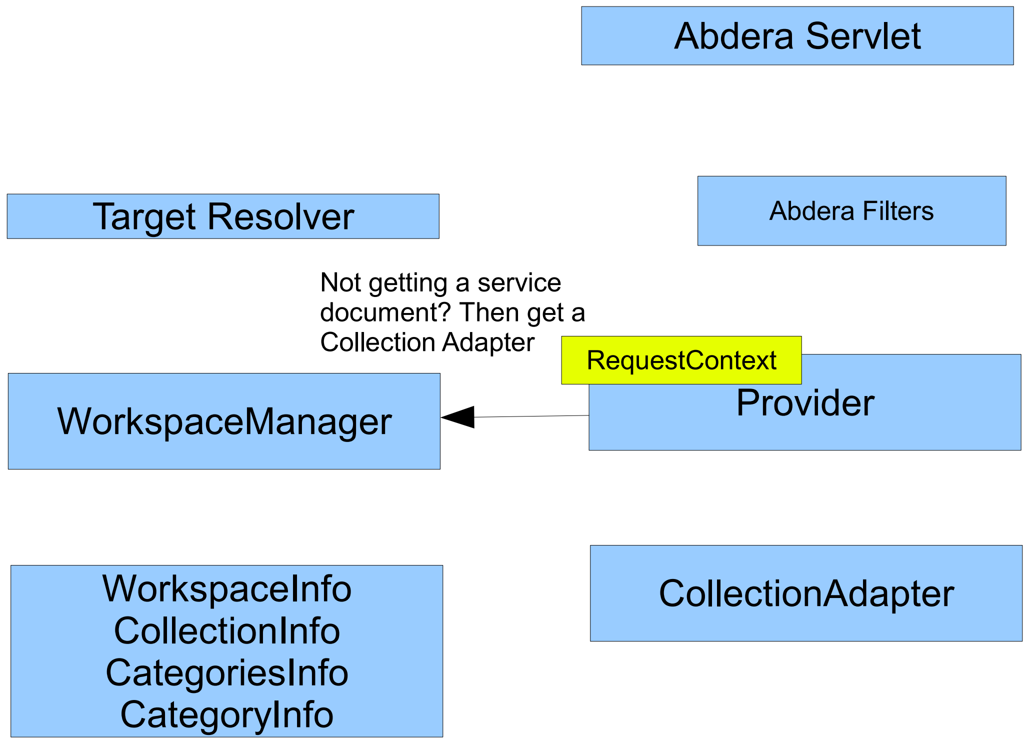
How requests are processed...



The Provider first determines whether the request is for an Atom Service Document.

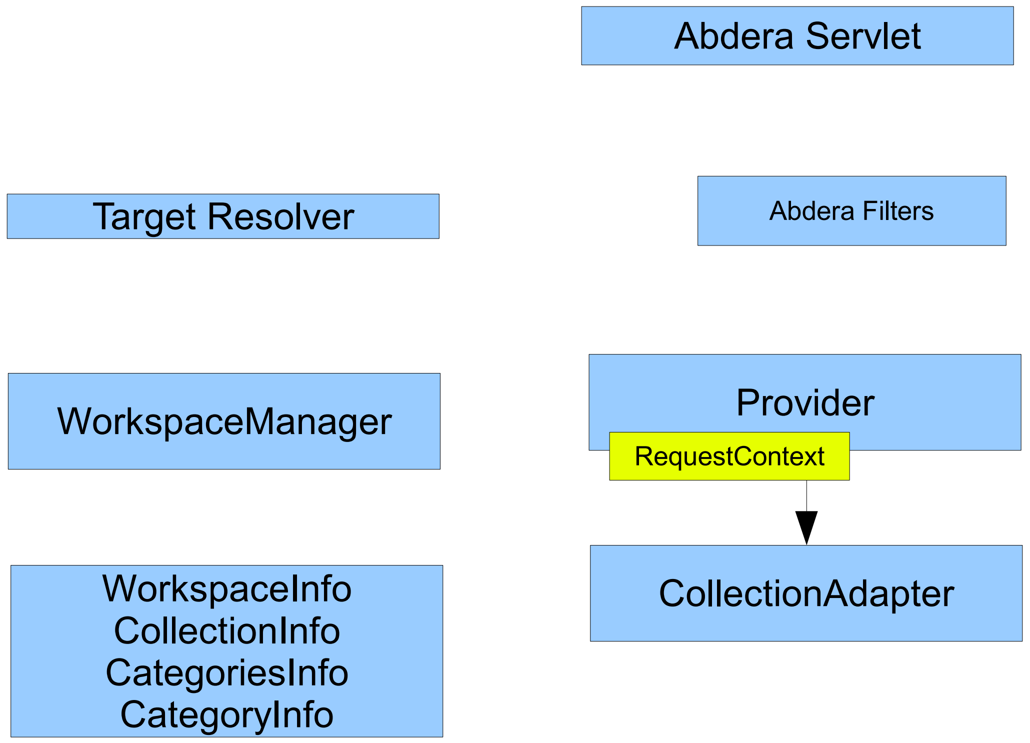
If a service document is being requested, the provider uses information provided by the Workspace Manager to build the service document. The data used comes from instances of the WorkspaceInfo, CollectionInfo, CategoriesInfo and CategoryInfo interfaces.

How requests are processed...



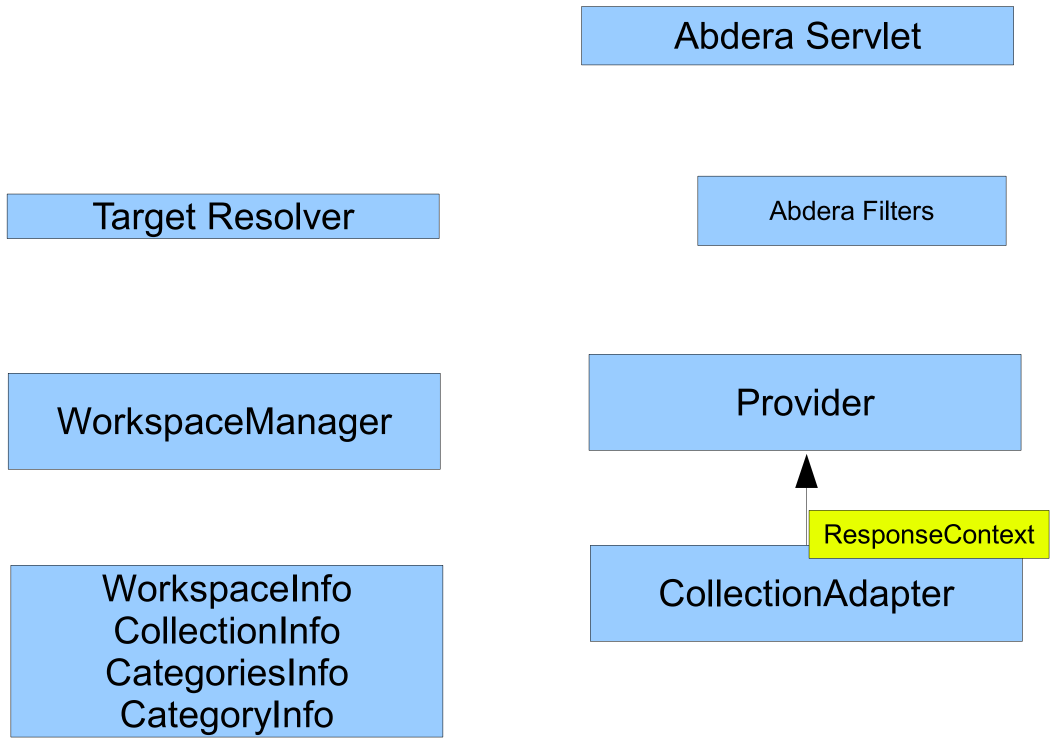
If the request is not for a service document, the Provider uses the Workspace Manager to select a Collection Adapter to handle the request. The Collection Adapter exposes methods like getFeed, getEntry, postEntry, etc, and is the interface that bridges the Atompub protocol to a specific back-end implementation.

How requests are processed...



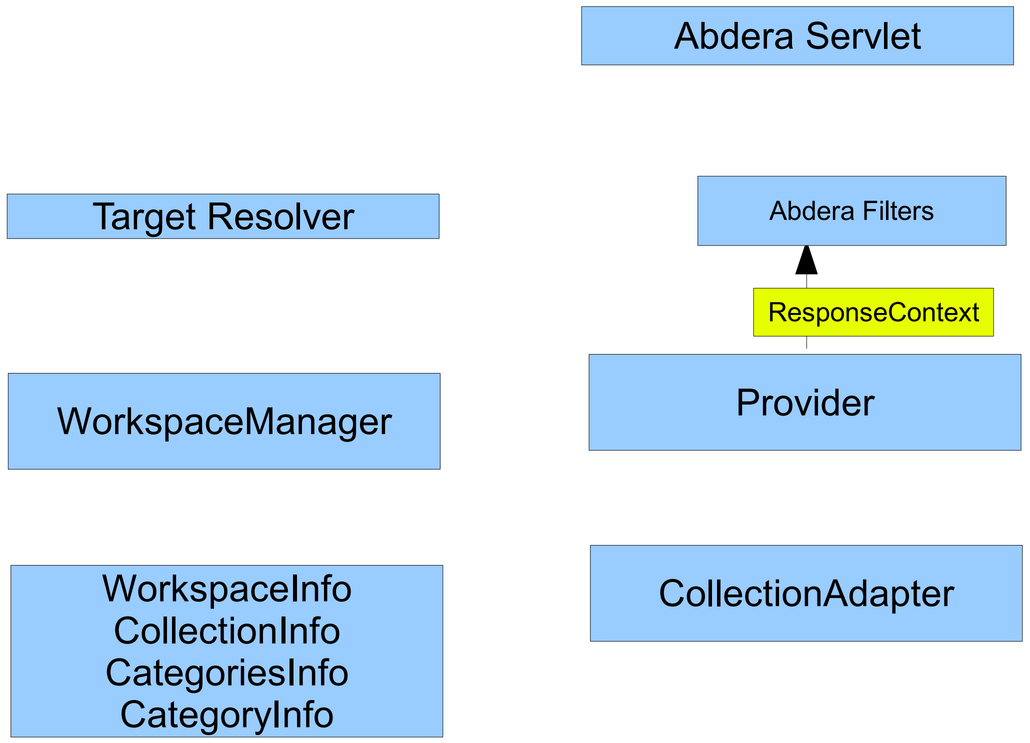
Once a Collection Adapter is selected, the provider will forward the request context on to the appropriate method as determined by the request method, target and target type. For instance, a GET method on a Collection target will be dispatched to the CollectionAdapter.getFeed() method; a PUT method on an Entry target will be dispatched to the CollectionAdapter.putEntry() method.

How requests are processed...



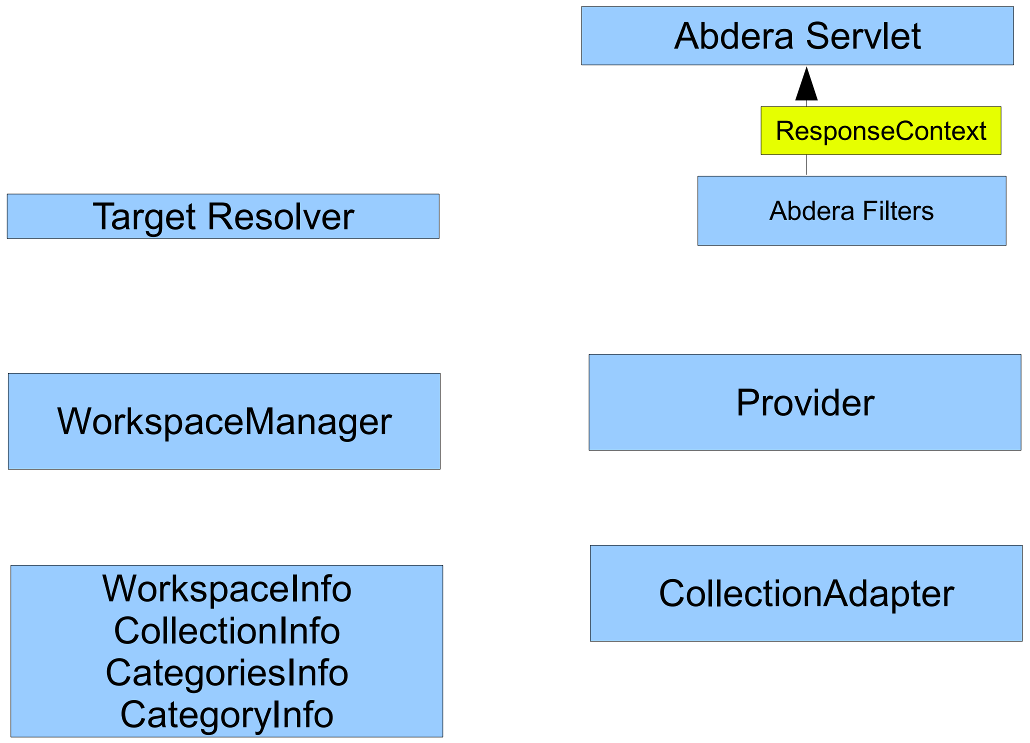
The Collection Adapter processes the request and prepares a ResponseContext object, handing that back to the Provider.

How requests are processed...



The ResponseContext is passed back through the chain of filters

How requests are processed...

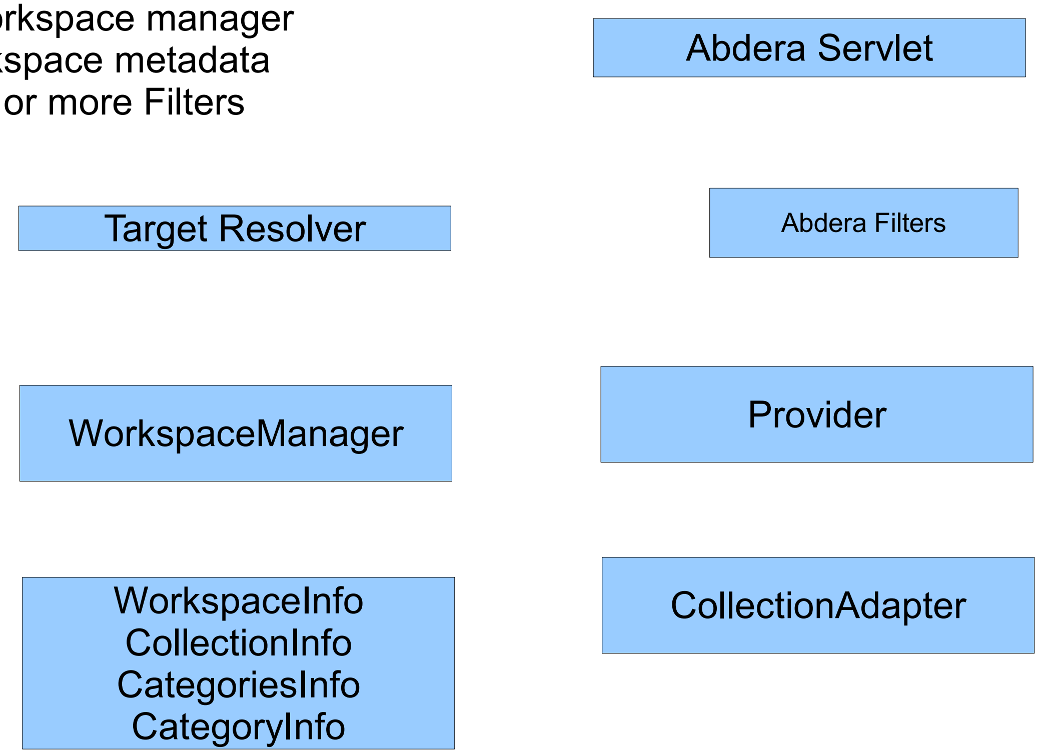


Then back out to the Abdera Servlet which writes the response out to the HttpServletResponse, completing the request processing

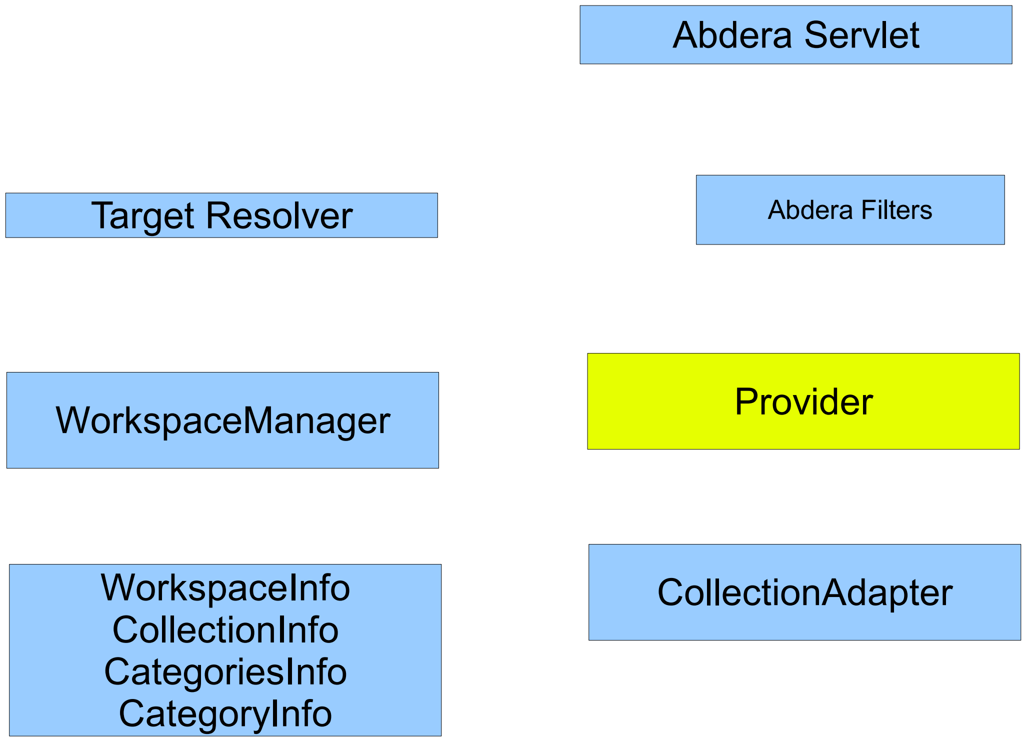
How to implement a server

Anyone wishing to implement an Atompub server using Abdera needs to provide:

- A Provider
- One or more Collection Adapters
- A Target Resolver
- A Workspace manager
- Workspace metadata
- Zero or more Filters



How to implement a server



The Provider is the single most important component. It determines how requests are processed and provides the Target Resolver and the Workspace Manager

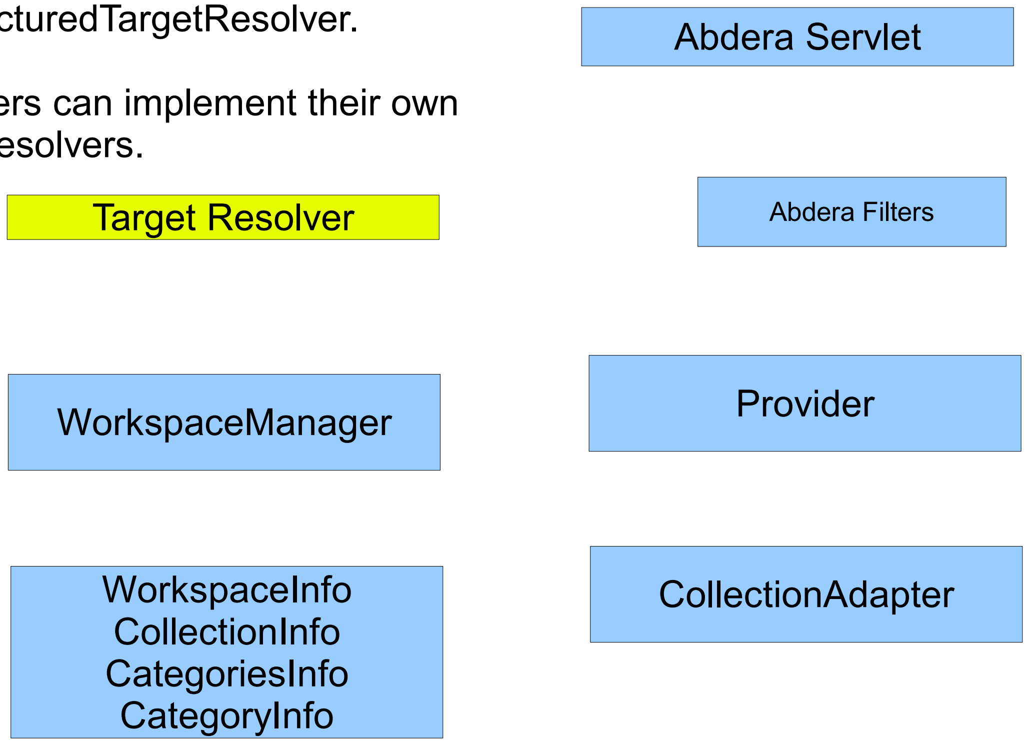
Abdera ships with two complete Provider implementations (DefaultProvider and BasicProvider). A developer can choose to use these or can choose to implement their own Provider.

How to implement a server

The target resolver uses information such as the request URI to determine which resource is being requested.

Abdera ships with two Target Resolver implementations: RegexTargetResolver and StructuredTargetResolver.

Developers can implement their own Target Resolvers.

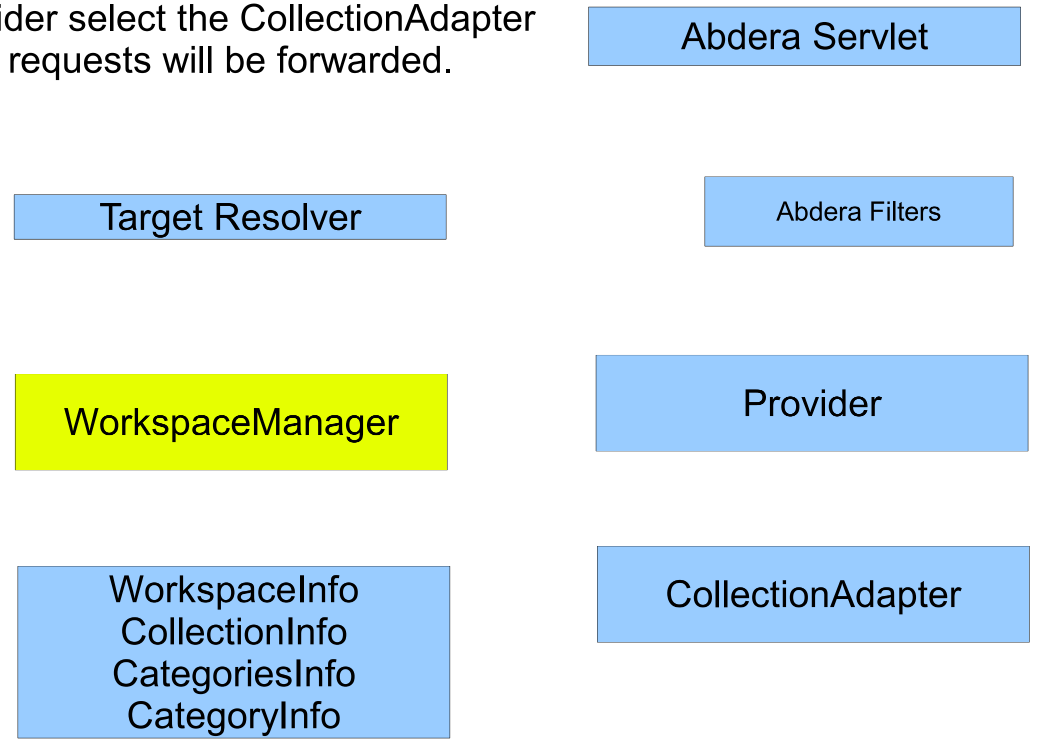


The RegexTargetResolver uses regular expressions to analyze the request URI and determine the identity of the resource being requested.

The StructuredTargetResolver uses a predetermined and static URL structure to determine the identity of the resource being requested.

How to implement a server

The Workspace Manager keeps track of the individual Workspaces and Collections. It tells the provider which collections are available, provides the metadata necessary for to build the Atompub service document, and helps the provider select the CollectionAdapter to which requests will be forwarded.



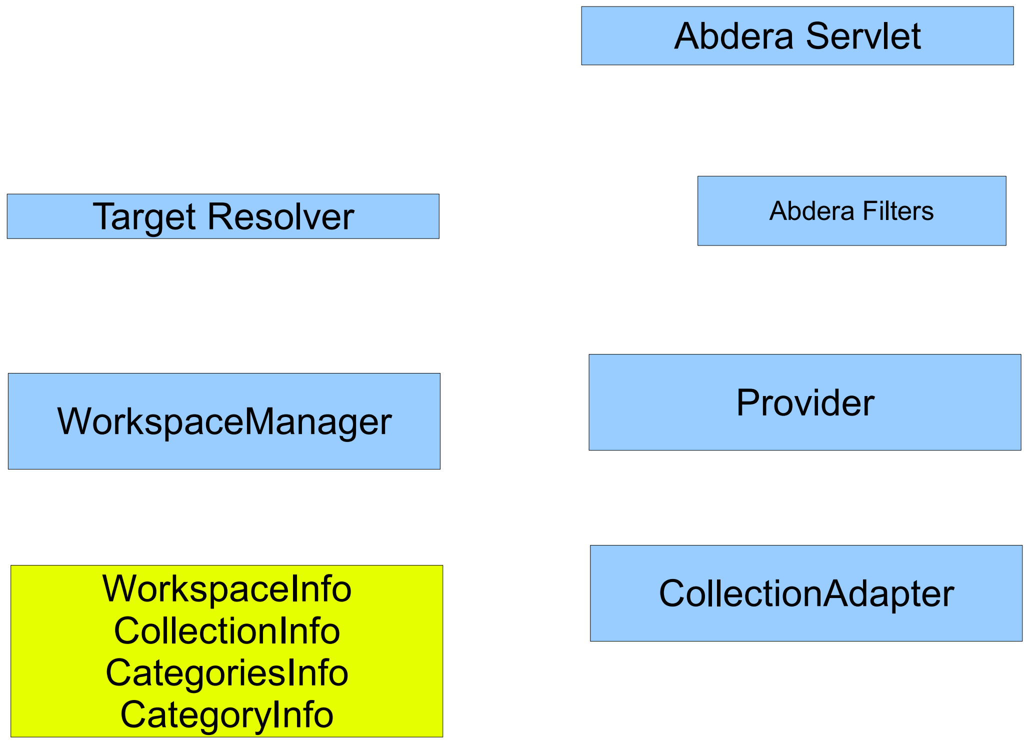
Workspace managers are fairly specific to the Provider implementation.

The DefaultProvider uses the DefaultWorkspaceManager implementation.

The BasicProvider uses the BasicWorkspace workspace manager.

Custom Provider implementations can choose to use these workspace managers or can provide their own WorkspaceManager implementation. The provider could even choose to implement the Workspace manager interface itself.

How to implement a server

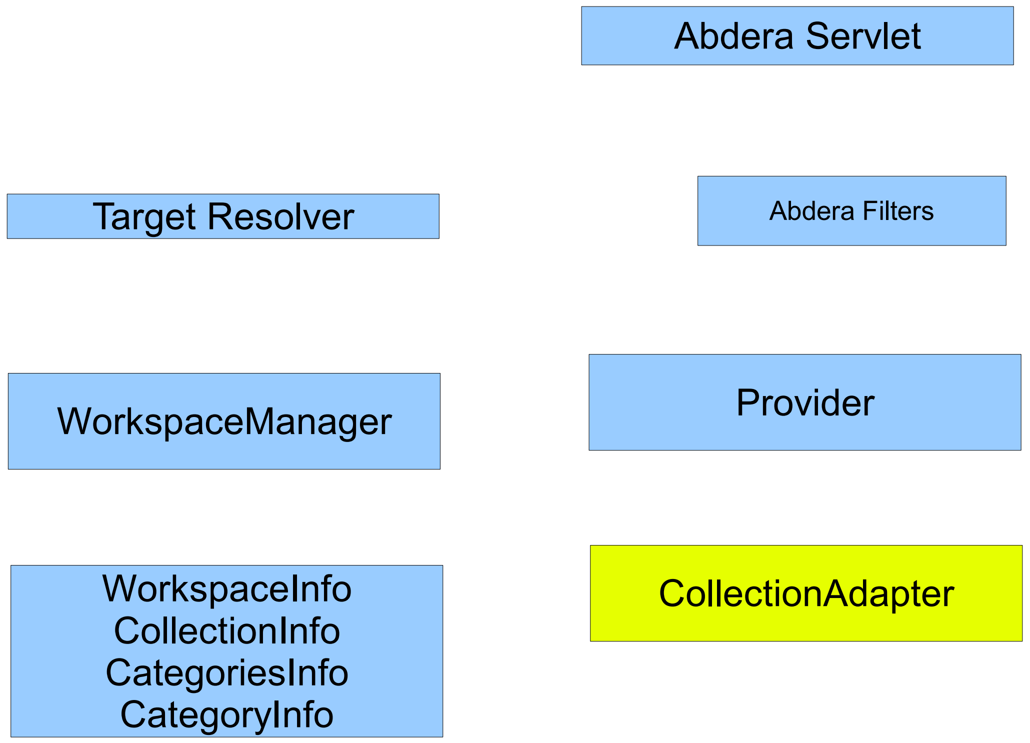


The various *Info interfaces provide the information necessary for building the Atom service document.

This information is provided by the Workspace Manager and is used by the Provider.

How this information is provided to the Workspace Manager depends on the specific implementation. The BasicWorkspace, for instance, uses *.properties files discovered on the classpath to build the metadata. The DefaultWorkspaceManager, however, requires the developer to provide the information.

How to implement a server



The Collection Adapter is the piece that actually implements the business logic of the Atompub server. It bridges the protocol with the backend persistence.

Implementations can vary broadly. Developers can create their own CA's, or use the JCR adapter that ships with Abdera, or use Adapters provided by other packages such as google feed-server, etc

An Example

- In this first example, we will implement an Atompub server using the DefaultProvider.
- The first thing we need to do is prepare the Provider.
- The Provider extends the DefaultProvider and provides some basic configuration metadata

```
public MyProvider extends DefaultProvider {
  public MyProvider() {
    super("^/([^\\/])+/");
    MyAdapter adapter = new MyAdapter();
    adapter.setHref("foo/acme/customers");
    SimpleWorkspaceInfo wi = new SimpleWorkspaceInfo();
    wi.setTitle("Customer Workspace");
    wi.addCollection(ca);
    customerProvider.addWorkspace(wi);
  }
}
```

An Example, continued

- Once the Provider is implemented, we need to implement the Adapter.
- There are several ways to implement Adapters, you could:
  - Implement the CollectionAdapter interface directly
  - Extend the AbstractCollectionAdapter
  - Extend the AbstractEntityCollectionAdapter
  - Extend the BasicAdapter

An Example, continued

- The AbstractCollectionAdapter and AbstractEntityCollectionAdapter utilities classes provide a number of helper functions and handle various default implementation details that make it easier to implement an Adapter.
- The BasicAdapter provides a greatly simplified interface for creating Adapters but is fairly limited in it's abilities. For instance, a BasicAdapter would be incapable of supporting features like feed paging.

An Example, continued

- For this example, we will extend AbstractEntityCollectionAdapter...

```
public class MyAdapter extends AbstractEntityCollectionAdapter<Customer> {
  ...
  public Customer postEntry(
    String title,
    IRI id,
    String summary,
    Date updated,
    List<Person> authors,
    Content content,
    RequestContext request)
      throws ResponseContextException {
    ...
  }
  ...
  public void deleteEntry(
    String resourceName,
    RequestContext request)
      throws ResponseContextException {
    ...
  }
  ...
  // other methods
}
```

An Example, continued

- The DefaultProvider uses the StructuredTargetResolver and the DefaultWorkspaceManager implementations, so we do not have to provide those for our example.
- The final step is to deploy the servlet

```
<web-app ... >
  <servlet>
    <servlet-name>AbderaServlet</servlet-name>
    <servlet-class>org.apache.abdera.protocol.server.servlet.AbderaServlet</servlet-class>
    <init-param>
      <param-name>org.apache.abdera.protocol.server.Provider</param-name>
      <param-value>com.foo.example.MyProvider</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>AbderaServlet</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```

An Example, continued

- That's basically it.
- Of course, I did gloss over a bunch of the details in the Collection Adapter implementation.

Another Example

- The BasicProvider makes things even easier. For the BasicProvider, all you need to do is provide a Collection Adapter implementation and a *.properties file.
- Properties files need to be located in the classpath under abdera.adapters.*
- The name of the properties file is the ID of the adapter (e.g. foo.properties, “foo” is the ID)

```
feedUri=http://localhost:9002/foo
adapterClassName=com.foo.example.MyAdapter
title=Feed Title
author=John Doe
```

Another Example, continued

- Then we just deploy the servlet.

```
<web-app ... >
  <servlet>
    <servlet-name>AbderaServlet</servlet-name>
    <servlet-class>org.apache.abdera.protocol.server.servlet.AbderaServlet</servlet-class>
    <init-param>
      <param-name>org.apache.abdera.protocol.server.Provider</param-name>
      <param-value>org.apache.abdera.protocol.server.provider.basic.BasicProvider</param-
value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>AbderaServlet</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Yes, that's it.

And yes, I did gloss over the details of the Collection Adapter implementation... again. Patience, my friends.

Yet another example

- To achieve the maximum flexibility in your Atompub server implementation, you can implement your own Provider, WorkspaceManager, TargetResolver and Collection Adapter.
- A couple of helper classes are provided to make it easier: AbstractProvider, AbstractWorkspaceProvider, AbstractWorkspaceManager, RegexTargetResolver, etc

Yet another example, continued

- AbstractProvider is the root of all Providers discussed thus far and handles the default request dispatching implementation. Unless you have a very good reason not to, every Provider implementation should extend this class.
- AbstractWorkspaceProvider extends AbstractProvider and implements the WorkspaceManager interface. This allows a Provider to act as it's own WorkspaceManager.

Yet another example, continued

- AbstractWorkspaceManager provides the basic functionality for WorkspaceManager implementations.
- RegexTargetResolver is a Target Resolver implementation that uses regular expressions to analyze the request URI's and select the Target Resource and identify it's Type.
- Adventurous developers can implement their own Target Resolver's if they wish.

Yet another example, continued

```
public class CustomProvider
  extends AbstractWorkspaceProvider {

  private final CollectionAdapter adapter;

  public CustomProvider() {

    this.adapter = new SimpleAdapter();

    super.setTargetResolver(
      new RegexTargetResolver()
        .setPattern("/atom(\\?[^#]*)?", TargetType.TYPE_SERVICE)
        .setPattern("/atom/([^/#?]+);categories", TargetType.TYPE_CATEGORIES, "collection")
        .setPattern("/atom/([^/#?;]+)(\\?[^#]*)?", TargetType.TYPE_COLLECTION, "collection")
        .setPattern("/atom/([^/#?]+)/([^/#?]+)(\\?[^#]*)?", TargetType.TYPE_ENTRY,
"collection","entry")
        .setPattern("/search", OpenSearchFilter.TYPE_OPENSEARCH_DESCRIPTION)
    );


    SimpleWorkspaceInfo workspace = new SimpleWorkspaceInfo();
    workspace.setTitle("A Simple Workspace");
    workspace.addCollection(
      new SimpleCollectionInfo(
        "feed",
        "A simple feed",
        "/atom/feed",
        "application/atom+xml;type=entry"
      ));
    addWorkspace(workspace);
  }

  ...
```

Yet another example, continued

- As you can see, this example is a little more involved.
- The Custom Provider needs to initialize the adapter, the target resolver, set the workspace metadata, etc
- The example uses the stock implementations of the metadata interfaces (e.g. SimpleWorkspaceInfo, SimpleCollectionInfo, etc). Developers can use their own impl's.
- And yes, I'm going to gloss over the Collection Adapter implementation details again.

Yet another example, continued

- Once the Provider is implemented, just deploy the servlet...

```
<web-app ... >
  <servlet>
    <servlet-name>AbderaServlet</servlet-name>
    <servlet-class>org.apache.abdera.protocol.server.servlet.AbderaServlet</servlet-class>
    <init-param>
      <param-name>org.apache.abdera.protocol.server.Provider</param-name>
      <param-value>com.foo.example.CustomProvider</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>AbderaServlet</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Again, that's pretty much it.

Other details

- DefaultProvider and BasicProvider each generate Atompub Service documents that are specific to each provider implementation.
- Custom Providers will generate service documents using the default code provided in AbstractProvider unless the developer chooses to override the getServiceDocument method.
- For now, if you want to add extension elements to the service document, you'll need to generate the document yourself.

Other details, continued

- All providers can have filters.

```
public MyProvider extends DefaultProvider {
  public MyProvider() {
    super("^/([^\\/])+/");
    MyAdapter adapter = new MyAdapter();
    adapter.setHref("foo/acme/customers");
    SimpleWorkspaceInfo wi = new SimpleWorkspaceInfo();
    wi.setTitle("Customer Workspace");
    wi.addCollection(ca);
    customerProvider.addWorkspace(wi);

    addFilter(
      new MyFilter(),
      new MyOtherFilter());

  }
}

public MyFilter implements Filter {
  public ResponseContext filter(
    RequestContext request,
    FilterChain chain) {
      // do filter like things, then call the next filter
      return chain.next(request);
  }
}
```

Filters are invoked in the order they are registered

Collection Adapters

- The Collection Adapter interface exposes methods that model the definition of the Atompub protocol...

```
public interface CollectionAdapter {

  ResponseContext postEntry(RequestContext request);

  ResponseContext deleteEntry(RequestContext request);

  ResponseContext getEntry(RequestContext request);

  ResponseContext putEntry(RequestContext request);

  ResponseContext getFeed(RequestContext request);

  ResponseContext getCategories(RequestContext request);

  ResponseContext extensionRequest(RequestContext request);

}
```

Media Collection Adapters

- If the Collection Adapter needs to support Atompub Media Link Entries, it needs to also implement the MediaCollectionAdapter interface

```
public interface MediaCollectionAdapter
  extends CollectionAdapter {

  ResponseContext postMedia(RequestContext request);

  ResponseContext deleteMedia(RequestContext request);

  ResponseContext getMedia(RequestContext request);

  ResponseContext putMedia(RequestContext request);

}
```

Transactional

- Collection Adapters can also implement the Transactional interface.
- The Provider will call the start/end methods on the Transactional interface before and after invoking the Collection Adapter; and will call the compensate method when an error occurs.

```
public interface Transactional {

  void start(RequestContext request) throws ResponseContextException;

  void end(RequestContext request, ResponseContext response);

  void compensate(RequestContext request, Throwable t);

}
```

Abstract Collection Adapters

- The AbstractCollectionAdapter and AbstractEntityCollectionAdapter base classes provide helpful default impl for many functions that will need to be implemented by most CA's.
- AbstractCollectionAdapter implements the CollectionAdapter, MediaCollectionAdapter, Transactional and CollectionInfo interfaces.
- AbstractEntityCollectionAdapter extends AbstractCollectionAdapter makes it easy to build Collections backed by a set of entities - such as a database row, domain objects, or files.

BasicAdapter

- The BasicAdapter offers a simplified CollectionAdapter interface with constrained functionality.
- BasicAdapter can be used to create very simple Atompub servers

```
public abstract class BasicAdapter
  implements CollectionAdapter {

  ...

  public abstract Feed getFeed() throws Exception;

  public abstract Entry getEntry(Object entryId) throws Exception;

  public abstract Entry createEntry(Entry entry) throws Exception;

  public abstract Entry updateEntry(Object entryId, Entry entry) throws Exception;

  public abstract boolean deleteEntry(Object entryId) throws Exception;

}
```

That's it

- For more information, see the API documentation and the sample code.
- Feedback and questions should be directed to the abdera-dev and abdera-user mailing lists.
- http://incubator.apache.org/abdera