Chapter 10
Servlets

- Introduction to Servlets
- Servlet Life Cycle
- Request Handling
- Response Generation
- HTTP Request Handlers
- The RequestDispatcher Interface
- Modeling Servlets in UML
- Modeling Other Servlet Aspects
- Servlet Deployment and Web Archives
- Identifying Servlets in Enterprise Applications
- Summary
Recall the control object TransferFunds from the discussion in Chapter 6. If you look closely at the final sequence diagram presented in Chapter 6, you'll notice two very distinct types of interactions performed by this class:

- Interactions with boundary objects to obtain information and perform some basic work
- Interactions with entity objects

Implementing a control class with a dual set of responsibilities and a large scope would make the control class less maintainable and less scalable. To make the control class more maintainable and scalable, it is preferable to partition the control class into two classes, one focused on the external interaction and the other responsible for carrying out the internal coordination and logic.

As it turns out, the externally focused part of TransferFunds evolves to a Java servlet. We introduce the servlet in the next section, and then discuss how you actually determine the responsibilities of the servlet in the context of the HomeDirect case study.

**Introduction to Servlets**

Historically speaking, servlets have been around longer and have seen much wider use than other Java 2 Platform, Enterprise Edition (J2EE) technologies. In the past, they tended to be large in size and complicated to maintain in comparison to the level of Web functionality they actually provided. Going forward, servlets will likely continue to see wide use for some time. However, their typical size is shrinking, and the level of complexity they tend to deal with is consistently becoming less.

The biggest benefit servlets offer developers is that they are designed specifically to process Hypertext Transfer Protocol (HTTP) requests coming from the Web client and pass back a suitable response. They perform this function well and require few resources to deliver this functionality.
In terms of structure, servlets are specialized Java classes that closely resemble the structure of Java applets, but they run on a Web server instead of a client.

An interesting point to note is that servlets can never have their own graphical user interface. Web servers host these components through the use of a Web container that manages all aspects of their life cycle.

**Common Usage**

Servlets have the distinction of being the most frequently used J2EE components currently found on the World Wide Web. As stated earlier, they typically involve a compact, lightweight architecture and design. They also tend to work well in cases where the requirements placed on this type of Web component are relatively small.

Most Web developers use servlets as the main point of entry to their server application from the Web client, and in this way, they are simply used as a conduit to pass information back and forth between the client and the server. Allowing client control to add or remove Web pages or files from the server can also be a good use for servlets, as long as the client has sufficient security clearance. Understandably, this usage is less frequently seen in practice.

**Best Served Small**

In theory, servlets are capable of doing just about anything possible that can be done with Java. The question arises as to why Web developers don’t just build everything they need using these components. The problem is that building large servlets to handle complex Web interactions, transactions, database synchronization, and other internal logic is not a very scalable approach. Developers would spend most of their time working out the intricacies of low-level transactions, state management, connection pooling, and so on.

In the past, servlets were often built to perform most or all of the following tasks:

- Check and process user input
- Handle significant business logic
- Perform database queries, updates, and synchronization
- Handle complex Web transactions
- Generate dynamic Web page content as output
- Handle Web page forwarding

More advanced J2EE solutions make use of JavaServer Pages (JSP), Enterprise JavaBeans (EJB), and JavaBeans to split up and offload much of this work, often
using new mechanisms built into J2EE to simplify the more difficult tasks for the developer. Servlets are then responsible for a more manageable set of tasks:

- Gathering and validating user input, but little or no actual processing
- Coordination of output, but with little or no direct generation of dynamic Web page content
- Minimal business logic

As you can see, servlets are best served small.

If constant demand for new Web site functionality did not exist, huge servlets could be built with all the accompanying aches and pains, and they might even stand a reasonable chance of being adequately maintained. However, the fact is that demands on Web sites keep increasing. Every service provider on the Web must continually update and upgrade to give their customers that new bit of data, that new cool feature, or that prized extra that differentiates their service from everyone else’s service.

Unfortunately, the bigger servlets come at the cost of an increased challenge of providing adequate code maintenance, not to mention the increased risk of breaking some of the existing functionality. The blessing of a lightweight architecture at the outset can easily turn into a wretched curse later on if you are not careful.

**J2EE Versions**

The information in this chapter applies equally well to servlets using J2EE 1.3 or J2EE 1.2. The differences between these two specifications are insignificant with respect to the basic Unified Modeling Language (UML) modeling of these particular Web components.

**Servlet Life Cycle**

As stated earlier, servlets are deployed within a servlet container, which in turn is hosted by a Web server. The particular capabilities and level of compliance of the Web server determines which version of the servlet specification you need to be working with.

The basic behavior of a servlet involves a request-response type model derived from the way the HTTP works; thus, the inherent applicability as a Web component. This behavior is illustrated via a statechart diagram in Figure 10-1.

Servlets are built as Java classes that extend one of two basic servlet implementation classes: `HttpServlet` and `GenericServlet`. The former is the most often
used, yet slightly more complex of the two. Both servlet types employ the same basic life cycle.

Life Cycle Methods

The servlet life cycle makes use of three basic request handler methods, of which any or all can be implemented within the extended servlet class:

- **init**: Initializes the servlet
- **service**: Services the client request
- **destroy**: Destroys the servlet

Of these three methods, the **service** method is the most interesting because it actually does the majority of the necessary processing. It typically does the following:

- Receives the request from the client
- Reads the request data
- Writes the response headers
- Gets the writer or output stream object for the response
- Writes the response data

The **service** method is at the heart of the **GenericServlet** type. However, it is almost never overridden and instead is split into lower level HTTP request handlers when used with the **HttpServlet** type.

The **init** and **destroy** life cycle methods are always available to be overridden, but in several cases might not be used if the servlet has no specific objects or connections it needs to initialize or terminate.
A sequence diagram in Figure 10-2 shows a simple example of a servlet. This diagram applies to both the *GenericServlet* and *HttpServlet*. It highlights a simple example where a database query is made to formulate the response to the client. Note that the *service* method is further refined into a specific HTTP request in the case of *HttpServlet*.

**Convenience Method**

Besides the life cycle methods, servlets commonly make use of what are referred to as convenience methods. One such convenience method that applies for all servlets is *getServletInfo*, which returns a general info string about the particular servlet—normally author, version, usage, and so on.

**Required Methods and Tagged Values**

When building a servlet that extends the *GenericServlet* class, the *service* life cycle method must be implemented; otherwise, the servlet is invalid. All other methods are optional.

![Sequence diagram showing servlet life cycle](image)

**Figure 10-2** Sequence diagram showing servlet life cycle
Multiple threads may call a generic servlet instance’s `service` method concurrently. To avoid this, the servlet can implement the `SingleThreadModel` interface, which is really a method of typing the servlet and indicating to the Web container that only a single thread should be allowed to call the method at any given time.

Implementing the `SingleThreadModel` can have a very significant effect on how the container decides to allocate resources when the servlet is deployed on the Web server, which can greatly impact the total number of concurrent servlet instances allowed.

Using this approach may be appropriate if you are dealing with a situation in which the servlet may need to alter information that is not thread safe or access resources that are not thread safe.

It is not recommended that you attempt to serialize any of the servlet methods other than by implementing this interface. The interface itself introduces no new methods.

### Request Handling

Servlets are request-driven and have specific capabilities available to them that simplify handling of incoming requests.

Recall that a request to a servlet may consist of several pieces of data (for example, when a form consisting of several fields is filled in and submitted).

When the Web container receives a request intended for a servlet, it encapsulates the incoming data into a `ServletRequest` object (commonly referred to as the request object) and passes it on as a parameter to the servlet’s `service` method. The servlet can then use the methods available in the `ServletRequest` interface to query the request object. Some of the queries are contained in the following list:

- `getCharacterEncoding` obtains information about the encoding format used for the request.
- `isSecure` finds out if the request was made over a secure channel.
- `getParameterNames` obtains a list of all parameter names in the request.
- `getRemoteAddr` determines the IP address of the client that sent the request.
- `getParameter` is used to retrieve the first parameter value associated with a named parameter type.
- `getParameterValues` is used to retrieve multiple parameter values associated with a named parameter type.
Several other methods are provided for querying different aspects of the request object. See `javax.servlet.ServletRequest` for more information. A specialized version, `HttpServletRequest`, for HTTP based servlet requests is also available. See `javax.servlet.http.HttpServletRequest` for more information.

Figure 10-3 shows a simple usage scenario involving a request object.

```java
HttpSession session = request.getSession(true);
:
:
// obtain the values for UserID and password
String loginID = request.getParameter("USERID");
String loginPassword = request.getParameter("PASSWORD");
:
```

Figure 10-3 Using the request object

Several other methods are provided for querying different aspects of the request object. See `javax.servlet.ServletRequest` for more information. A specialized version, `HttpServletRequest`, for HTTP based servlet requests is also available. See `javax.servlet.http.HttpServletRequest` for more information.

Figure 10-3 shows a simple usage scenario involving a request object.

### Response Generation

A request generally warrants a response, and servlets are no exception in this regard.

Servlets make use of `ServletResponse` to simplify this common task. The `ServletResponse` object, commonly referred to as the response object, is in fact provided to a servlet alongside the request object as a parameter to the `service` method.

Output can be written in either binary or character format by obtaining a handle to either a `ServletOutputStream` object or a `PrintWriter` object, respectively. Some of the other methods provided by the `ServletResponse` interface are contained in the following list:

- `getOutputStream` obtains the handle to a `ServletOutputStream` object for binary data.
- `getWriter` obtains the handle to a `PrintWriter` object for character data.
- `setBufferSize` can be used to establish the buffer size for the response to enable better performance tuning.
- `flushBuffer` flushes the current contents of the buffer.

---

1. If you are new to Java or unsure about this reference, see the “Conventions” section in the Preface of this book.
For more information, see \texttt{javax.servlet.ResponseObject} and \texttt{javax.servlet.ServletOutputStream}.

An HTTP specific response object is also available and provides additional capabilities related to HTTP response header formulation. See \texttt{javax.servlet.http.HttpServletResponse} for more information.

Figure 10-4 shows a simple usage scenario involving a response object.

**Alternatives for Response Generation**

If you take a good look at Figure 10-4, you will see several HTML tags involved in the generation of output from the servlet. This represents only one approach for generation of dynamic output.

Another similar but more structured approach is to use libraries of HTML files to generate common headers and footers for the necessary response Web pages, with the dynamic portion of the page still generated much like what was shown in Figure 10-4.

A third and cleaner approach is to use the power of JSP and JavaBeans whenever possible. In this approach, the servlet simply needs to forward to a JSP page that contains all of the necessary presentation information and use JSP technology and JavaBeans to fill in the dynamic content portions of the page. Other than the forward, the servlet has little else to do with presentation except perhaps coordinating the necessary items for the JSP page to successfully do its work.

We discuss this approach further in Chapter 11.

```java
PrintWriter out;
out = response.getWriter();
out.println("<HTML><HEAD><TITLE>");
out.println("Login Unsuccessful");
out.flush();
out.close();
```

Figure 10-4  Generating the response
HTTP Request Handlers

The HttpServlet class extends the GenericServlet class and therefore inherits all of the standard servlet capabilities. In addition to the basic servlet life cycle methods and convenience method, the more complex HttpServlet class adds methods to aid in the processing of HTTP requests. These commonly used handler methods are

- **doGet**: Handles HTTP GET requests
- **doPost**: Handles HTTP POST requests

In the case of `doGet`, there is an additional method used for conditional HTTP GET support (the different HTTP request types are explained later in this section). The `getLastModified` method is like HTTP GET, but only returns content if it has changed since a specified time. This method can only be used if `doGet` has also been overridden and is intended to be used in cases where you are dealing with content that does not change much from request to request.

### Advanced Handler Methods

There are several advanced handler methods that are defined as well:

- **doPut**: Handles HTTP PUT requests
- **doDelete**: Handles HTTP DELETE requests
- **doOptions**: Handles HTTP OPTIONS requests
- **doTrace**: Handles HTTP TRACE requests

Unlike the GenericServlet class, servlets based on HttpServlet have almost no valid reason to override the `service` method. Instead, you typically override these request handlers, which the base `service` method implementation calls when appropriate. The `doOptions` and `doTrace` methods also have virtually no valid reason to be overridden and are present only for full HTTP support. An HttpServlet must override at least one method, which usually means one of the remaining life cycle methods or request handlers.
Quick Guide to HTTP Requests

For the most commonly used request handler methods, the following list provides a quick guide of what the HTTP requests are for:

- **GET**: A call to get information from the server and return it in a response to the client. The method processing this call must not have any side effects, so it can be repeated safely again and again. A GET call is typically used when a servlet URL is accessed directly from a Web browser or via a forward from a form on an HTML or JSP page. A GET call shows the data being passed to the servlet as part of the displayed URL on most Web browsers. In certain cases, this might not be very desirable from a security perspective.

- **POST**: A call to allow the client to send data to the server. The method processing this call is allowed to cause side effects, such as updating of data stored on the server. A POST call can be used instead of a GET when forwarding from a form on an HTML or JSP page. Unlike GET, the use of POST hides from view any data being passed to the servlet. Some developers choose to process GET and POST exactly the same, or simply ignore one or the other if they do not want that particular call to be supported.

- **PUT**: This call is similar to POST, but allows the client to place an actual file on a server instead of just sending data. It is also allowed to cause side effects, just like POST. Although available, the use of a PUT call is not very common.

- **DELETE**: This call is similar to PUT, but allows the client to remove a file or Web page from the server. It is also allowed to cause side effects in the same way as PUT. Although available, the use of a DELETE call is not very common.

There is another request not specifically mentioned in the preceding list called HTTP HEAD. This request, although valid in the context of the HttpServletResponse class itself, is actually handled internally by making a call to the doPost method, which you might have overridden. It differs in that it only returns the response headers that result from processing doGet and none of the actual response data.

The RequestDispatcher Interface

Given the simplicity of servlets, it makes sense to keep each servlet focused on a specific task, and then set up multiple servlets to collaboratively achieve a more complex task. Servlets can take care of the mechanical aspects of such collaborative efforts easily by implementing the RequestDispatcher interface.
The RequestDispatcher interface provides two key capabilities:

- **forward**: This method allows a servlet to forward a request to another Web component. The servlet forwarding the request may process the request in some way prior to the forwarding. Forward can effectively be used to achieve servlet chaining where each link in the chain produces some output that can be merged with the original request data, and then be used as the input to the next servlet in the chain. This is essentially similar to the concept of pipes in the UNIX world.

  Note that the term “redirect” is sometimes used interchangeably with “forward,” intending the same meaning. However, this should not be confused with the `sendRedirect` method found on the servlet response. A `sendRedirect` call does not guarantee preservation of the request data when it forwards to a new page, so it does not allow for the same servlet chaining capabilities.

- **include**: This method permits the contents of another Web component to be included in the response from the calling servlet. The first servlet simply includes the other servlet at the appropriate point in the output, and the output from the servlet being included is added to the output stream. This is similar in concept to Server Side Includes (SSI).

### Modeling Servlets in UML

The `GenericServlet` class is usually modeled as a standard Java class with the `<<Generic_Servlet>>` stereotype applied. The presence of the stereotype allows for the servlet to be represented in a compact form and still be easily distinguished as a generic servlet without the need to show the inheritance tree on the same diagram. A generic servlet can include any of the life cycle methods or the convenience method discussed earlier.

A more expanded view of the servlet class showing the inheritance from the `GenericServlet` class can also be used. In most cases, though, the more compact stereotyped class view is sufficient. The compact and expanded representations of the servlet are shown in Figure 10-5.

If the servlet implements the `SingleThreadModel` interface, which controls serialization of the `service` method, the servlet can be shown with the interface

---

2. SSI allows embedding of special tags into an HTML document. The tags are understood by the Web server and are translated dynamically as the HTML document is served to the browser. JSPs build on this idea.
to highlight this aspect. Optionally, the servlet can be tagged with \texttt{(SingleThreadServlet=True)} instead to clearly identify this on the diagram in a somewhat more compact format.

An example of a servlet that implements the \texttt{SingleThreadModel} is shown in Figure 10-6.

The \texttt{HttpServlet} class is modeled similarly to \texttt{GenericServlet}, but with the \texttt{<<Http_Servlet>>} stereotype applied. It can also include the life cycle methods, the convenience method, and any of the HTTP request handlers previously discussed.

---

**Figure 10-5** Compact and full representation of a generic servlet

**Figure 10-6** Servlet supporting the SingleThreadModel
The SingleThreadModel details as well as the tagged value for SingleThreadServlet apply in the HttpServlet class exactly the same way as they did for GenericServlet. As stated earlier, you should not attempt to serialize any of the servlet methods other than by implementing this interface. This interface does not introduce any new methods.

**Modeling Other Servlet Aspects**

Other aspects of servlets that warrant modeling are servlet forward, servlet include, ServletContext, and Servlet Session Management. The following sections discuss these aspects in more detail.

**Servlet Forward**

Servlet forward is a special kind of relationship, and modeling it explicitly can help clarify the overall application logic. For example, it can shed light on the flow of the processing logic. In complicated forward chains, the relationship may be indicative of some algorithm being implemented. Two specific approaches help to identify the overall application logic in this regard.

First, on the class diagram, label the relationships between the servlets that invoke forward on other Web components with the <<forward>> relationship. An example is shown in Figure 10-7.

For more complicated servlet chaining, an activity diagram can be used to show the overall interaction. If desired, request and response objects with attributes appropriately updated at specific points can be shown to demonstrate the overall algorithm. See Figure 10-8.

![Figure 10-7](image-url)  Modeling servlet forwarding on a class diagram
In this case, we have labeled the transition with the <<forward>> stereotype to emphasize that it represents a forward relationship between the elements involved. The comments shown for each occurrence of the response object identify what happens as the request and response objects pass through the chain.

**Servlet Include**

*Include* is another significant and special relationship as it affects the results produced by a servlet. In fact, *include* may be used as a means to structure and organize the overall output in a modular fashion. Servlet *include* relationships are modeled in the same fashion as the *forward* relationship, that is, as a unidirectional association stereotyped <<include>>. The direction of the association is
from the including servlet to the resource being included. An example is shown in Figure 10-9. In the example, a servlet responsible for creating a mortgage amortization table includes header and footer servlets whose sole purpose is to generate the page header and footer, respectively.

**ServletContext**

Each servlet runs in some environment. The **ServletContext** provides information about the environment the servlet is running in. A servlet can belong to only one **ServletContext** as determined by the administrator. Typically, one **ServletContext** is associated with each Web application deployed in a container. In the case of distributed containers, one **ServletContext** is associated with one Web application per virtual machine.

The **ServletContext** interface can be used by servlets to store and retrieve information and share information among servlets. A servlet obtains the **ServletContext** it is running in by using the `getServletContext` method.

Some of the basic services provided by the **ServletContext** interface are

- **setAttribute**: Stores information in the context
- **getAttribute**: Retrieves information stored in the **ServletContext**

---

**Figure 10-9**  Servlet include relationship
getAttributeNames: Obtains the names of attributes in the context
removeAttribute: Removes an attribute in the context

An approach similar to the one discussed for servlet forwarding and shown in Figure 10-8 can be employed to model servlet interactions with the Servlet-Context.

Servlet Session Management

Given the stateless nature of the HTTP protocol, managing repeat interaction and dialog with the same client (such as that required for an ongoing shopping session) poses some serious challenges. There are various means of overcoming these challenges:

- **Hidden fields**: Hidden fields are embedded within the page displayed to the client. These fields are sent back to the client each time a new request is made, thereby permitting client identification each time a client makes a request.

- **Dynamic URL rewriting**: Extra information is added to each URL the client clicks on. This extra information is used to uniquely identify each client for the duration of the client session, for example, adding a “?sessionid=97859” to the end of each URL the client clicks to identify that the request is associated with session id 97859.

- **Cookies**: Stored information can later be passed back to the client repeatedly. The Web server provides the cookie to the browser. Cookies are one of the more popular means of setting up a servlet session.

- **Server-side session object**: Cookies and URL encoding suffer from limitations on how much information can be sent back with each request. In server-side session management, the session information is maintained on the server in a session object and can be accessed as required. Server-side session objects are expensive to use, so it is best to use them sparingly.

The Java Servlet Application Programming Interface (API) provides abstractions that directly support some of the session management techniques discussed in the preceding list.

The core abstraction provided by the servlet API is the **HTTP session**, which facilitates handling of multiple requests from the same user.

Figure 10-10 gives an example of servlet session management.

Activity diagrams can be used to model the servlet and session interaction. This is similar to the approach discussed for servlet forwarding and shown in Figure 10-8.
A descriptor based on XML is used in the deployment of servlets on a Web server. The compiled servlet class, additional supporting Java classes, and the deployment descriptor are packaged together into a Web archive file, also known as a "war" file.

The deployment descriptor is an XML-based file that contains specific configuration and deployment information for use by the servlet container.

Figure 10-11 shows an example of a vanilla XML deployment descriptor for an HttpServlet. Additional required fields in the descriptor are filled in during configuration and deployment on the Web server.

We discuss servlet deployment descriptors and Web archive files and their role in the context of modeling in Chapter 15.
Identifying Servlets in Enterprise Applications

Now that you have become intimately familiar with servlets, it is time to return to building the HomeDirect online banking example.

At the beginning of this chapter, we identified the need to evolve the control object in the Transfer funds use case by splitting it into two, one focused on the external interaction and the other focused on the internal interaction.

Of course, the question remains: How do you actually arrive at this division of responsibilities? The answer is partly based on understanding what a servlet is capable of doing and the rest on judgment and experience. In general, the role of the servlet is that of a coordinator between the boundary objects and the rest of the system. All the interaction between the boundary object and the composite control class belongs in the new servlet. How you split the interaction that is shown between the control object and the entity objects is somewhat less clear.

The key factor to remember is that the servlet is primarily a coordinator; and hence, it should only take on lightweight responsibilities, which could include initiating some business logic. However, actual business logic, computations, interaction with entity objects, and so on would all fall outside of these responsibilities.

With this in mind, let’s take another look at the interactions involving the control object as shown in Figure 10-12.

If we look at all of the control object responsibilities, we see that the lower half is comprised of several actions that together form a complete transaction. We decide to separate this part and have it be handled by an internally focused control object, leaving the rest to be taken care of by a servlet. Figure 10-13 shows the result of this division of duties.

In this scenario, the servlet is an example of what the RUP calls a front component. A front component is typically a servlet or a JSP that is primarily responsible for processing user input but is not itself responsible for presentation. Rather, it acts simply as an entry point to the application and as a coordinator with other components. Note that the term “TransferPage” is used to generically represent a user interface. We might decide to make this a static HTML page or something more dynamic.

We discuss what to do with the other, internal focused control object in the next chapter.

Of the two types of servlets discussed, an HttpServlet appears ideally suited to take on the external interaction role due to the Web-based nature of the HomeDirect interface.

Figure 10-14 expands further on this scenario. There are really two customer actions involved in this use case. The first is where the customer decides
to do a transfer action. This invokes MainServlet, which coordinates the retrieval of the pertinent accounts data and displays this via the TransferPage boundary object. The customer then selects the desired accounts and enters the amount to transfer. Control at this point is forwarded to a secondary TransferServlet, which coordinates the actual transfer action via the internally focused control object.

Figure 10-15 shows the details of the servlets for this example. We purposely have the servlets handling as little processing as possible, offloading most of the
work to the other J2EE components, which we discuss in more detail in later chapters covering JSP and EJB technology.

The decision to split up the servlet responsibilities will vary depending on specific needs. In this case, our preference was to minimize the responsibilities of the MainServlet to being a coordinator only. A secondary level of servlets was therefore developed to handle the details of individual use cases.
Identifying Servlets in Enterprise Applications

Figure 10-14  MainServlet and TransferServlet division of responsibilities

Figure 10-15  MainServlet and TransferServlet details
Servlets have a lightweight architecture and are ideally suited for request-response paradigms. Servlets are like ordinary Java classes with the exception that specific life cycle methods must exist in the servlet. Specific HTTP request handler methods are used for `HttpServlet`. Two types of servlets, `GenericServlet` and `HttpServlet`, are defined in the J2EE.

Servlets are modeled as stereotyped Java classes. UML modeling techniques can bring special focus on some aspects of servlets, such as forwarding, including, and session management by servlets.

An XML deployment descriptor is required for deploying a servlet.