SERVER-SIDE

WEB SERVICES II

THE TOMCAT SERVER

RESTFUL APIS

OTHER WEB APIS

VERSION

v1.0 09 October 2021

ACKNOWLEDGMENTS

THANKS TO:

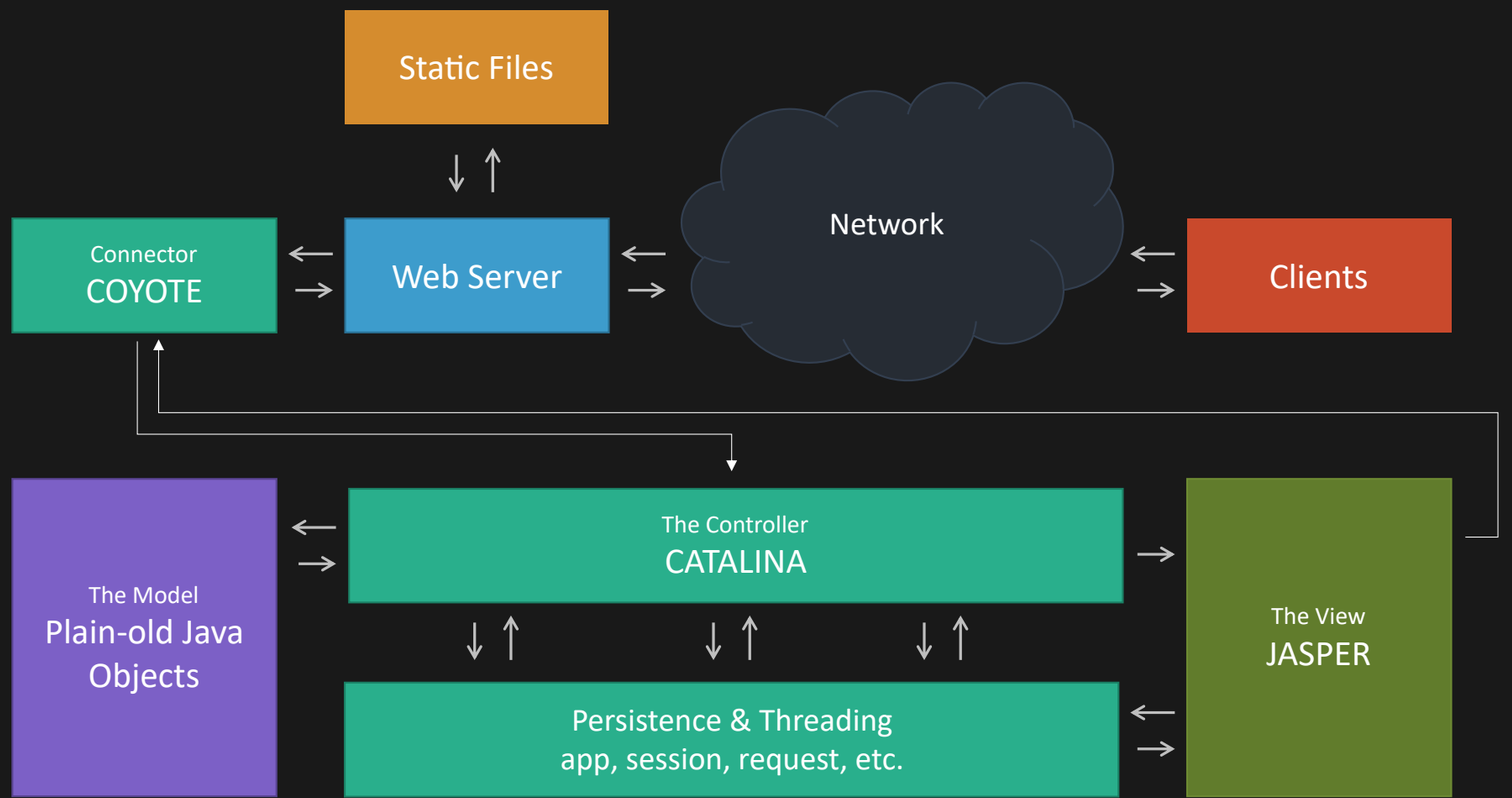
- Hamzeh Roumani, who has shaped EECS-4413 into a leading hands-on CS course at EECS and who generously shared all of his course materials and, more importantly, his teaching philosophy with me;
- Parke Godfrey, my long-suffering Master's supervisor and mentor; and
- Suprakash Datta for giving me this opportunity to teach this course.

PRINTABLE VERSION OF THE TALK

Download PDF

THE TOMCAT SERVER

TOMCAT



FIRST SERVLET

- `MyFirstServlet extends HttpServlet`. This is mandatory because all servlets must be either a generic servlet that extends `javax.servlet.GenericServlet` or an HTTP servlet that extends `javax.servlet.http.HttpServlet`.
- Overriding `doGet()` and `doPost()` methods. These methods are defined in `HttpServlet` class. Whenever a `GET` or `POST` request come, it is mapped to it's respective method.
- `HTTP GET` request to this servlet, then `doGet()` method is called. There are some other useful methods as well which you can override to control the application in runtime (e.g.: `getServletInfo()`).
- `HttpServletRequest` and `HttpServletResponse` are default parameters to all `doXXX()` methods. We will learn more about these objects in later section.
- You can use `@WebServlet` annotation to automatically register your servlet into the runtime, specifying the servlet's name and URL patterns that the runtime will match for it.

```java
package services;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

@WebServlet(name = "MyFirstServlet", urlPatterns = {"/first"})
public class MyFirstServlet extends HttpServlet {
  @Override
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    try (PrintStream out = new PrintStream(response.getOutputStream(), true)) {
      // Write some content
      out.println("<html>");
      out.println("<head>");
      out.println("<title>MyFirstServlet</title>");
      out.println("</head>");
      out.println("<body>");
      out.println("<h2>Servlet MyFirstServlet at " + request.getContextPath() + "</h2>'
      out.println("</body>");
      out.println("</html>");
    }
  }

  @Override
  protected void doPost(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    // handle POST request
  }
}
```

Source: Complete Java Servlets Tutorial (December 2020).

SERVLET LIFE CYCLE METHODS

Whenever in your application, a servlet is loaded and used; there occurs a series of events, during the initialization and destruction of that servlet. These are called life cycle events (or methods) of the servlet. Three methods are central to the life cycle of a servlet. These are `init()`, `service()`, and `destroy()`. They are implemented by every servlet and are invoked at specific times by the runtime.

```java
@WebServlet()
public class LifecycleServlet extends HttpServlet {
  @Override
  protected void init(ServletConfig config) throws ServletException {
    // custom initialization code
  }

  @Override
  protected void service(HttpServletRequest  request,
                         HttpServletResponse response)
      throws ServletException, IOException {
    switch (request.getMethod()) {
      case "GET":     doGet(request, response);     break;
      case "HEAD":    doHead(request, response);    break;
      case "POST":    doPost(request, response);    break;
      case "PUT":     doPut(request, response);     break;
      case "DELETE":  doDelete(request, response);  break;
      case "OPTIONS": doOptions(request, response); break;
      case "TRACE":   doTrace(request, response);   break;
      default:
        response.sendError(HttpServletResponse.SC_NOT_IMPLEMENTED);
    }
  }

  @Override
  protected void destroy() {
    // custom destruction code
  }
}
```

Source: Complete Java Servlets Tutorial (December 2020).

SERVLET LIFE CYCLE METHODS

Whenever in your application, a servlet is loaded and used; there occurs a series of events, during the initialization and destruction of that servlet. These are called life cycle events (or methods) of the servlet. Three methods are central to the life cycle of a servlet. These are `init()`, `service()`, and `destroy()`. They are implemented by every servlet and are invoked at specific times by the runtime.

1. During initialization stage of the servlet life cycle, the web container initializes the servlet instance by calling the `init()` method, passing an object implementing the `javax.servlet.ServletConfig` interface. This configuration object allows the servlet to access name-value initialization parameters. This is called only once in lifetime of that servlet instance.

```java
@WebServlet()
public class LifecycleServlet extends HttpServlet {
  @Override
  protected void init(ServletConfig config) throws ServletException {
    // custom initialization code
  }

  @Override
  protected void service(HttpServletRequest  request,
                         HttpServletResponse response)
      throws ServletException, IOException {
    switch (request.getMethod()) {
      case "GET":     doGet(request, response);     break;
      case "HEAD":    doHead(request, response);    break;
      case "POST":    doPost(request, response);    break;
      case "PUT":     doPut(request, response);     break;
      case "DELETE":  doDelete(request, response);  break;
      case "OPTIONS": doOptions(request, response); break;
      case "TRACE":   doTrace(request, response);   break;
      default:
        response.sendError(HttpServletResponse.SC_NOT_IMPLEMENTED);
    }
  }

  @Override
  protected void destroy() {
    // custom destruction code
  }
}
```

Source: Complete Java Servlets Tutorial (December 2020).

SERVLET LIFE CYCLE METHODS

Whenever in your application, a servlet is loaded and used; there occurs a series of events, during the initialization and destruction of that servlet. These are called life cycle events (or methods) of the servlet. Three methods are central to the life cycle of a servlet. These are `init()`, `service()`, and `destroy()`. They are implemented by every servlet and are invoked at specific times by the runtime.

1. During initialization stage of the servlet life cycle, the web container initializes the servlet instance by calling the `init()` method, passing an object implementing the `javax.servlet.ServletConfig` interface. This configuration object allows the servlet to access name-value initialization parameters. This is called only once in lifetime of that servlet instance.
2. After initialization, the servlet instance can service client requests. The web container calls the `service()` method of the servlet for every request. The `service()` method determines the kind of request being made and dispatches it to an appropriate method to handle the request.

```java
@WebServlet()
public class LifecycleServlet extends HttpServlet {
  @Override
  protected void init(ServletConfig config) throws ServletException {
    // custom initialization code
  }

  @Override
  protected void service(HttpServletRequest  request,
                         HttpServletResponse response)
      throws ServletException, IOException {
    switch (request.getMethod()) {
      case "GET":     doGet(request, response);     break;
      case "HEAD":    doHead(request, response);    break;
      case "POST":    doPost(request, response);    break;
      case "PUT":     doPut(request, response);     break;
      case "DELETE":  doDelete(request, response);  break;
      case "OPTIONS": doOptions(request, response); break;
      case "TRACE":   doTrace(request, response);   break;
      default:
        response.sendError(HttpServletResponse.SC_NOT_IMPLEMENTED);
    }
  }

  @Override
  protected void destroy() {
    // custom destruction code
  }
}
```

Source: Complete Java Servlets Tutorial (December 2020).

SERVLET LIFE CYCLE METHODS

Whenever in your application, a servlet is loaded and used; there occurs a series of events, during the initialization and destruction of that servlet. These are called life cycle events (or methods) of the servlet. Three methods are central to the life cycle of a servlet. These are `init()`, `service()`, and `destroy()`. They are implemented by every servlet and are invoked at specific times by the runtime.

1. During initialization stage of the servlet life cycle, the web container initializes the servlet instance by calling the `init()` method, passing an object implementing the `javax.servlet.ServletConfig` interface. This configuration object allows the servlet to access name-value initialization parameters. This is called only once in lifetime of that servlet instance.
2. After initialization, the servlet instance can service client requests. The web container calls the `service()` method of the servlet for every request. The `service()` method determines the kind of request being made and dispatches it to an appropriate method to handle the request.
3. Finally, the web container calls the `destroy()` method that takes the servlet out of service. You should call this method if you want to close or destroy some filesystem or network resources before the servlet goes out of scope. The `destroy()` method, like `init()`, is called only once in the lifecycle of a servlet.

```java
@WebServlet()
public class LifecycleServlet extends HttpServlet {
  @Override
  protected void init(ServletConfig config) throws ServletException {
    // custom initialization code
  }

  @Override
  protected void service(HttpServletRequest  request,
                         HttpServletResponse response)
      throws ServletException, IOException {
    switch (request.getMethod()) {
      case "GET":     doGet(request, response);     break;
      case "HEAD":    doHead(request, response);    break;
      case "POST":    doPost(request, response);    break;
      case "PUT":     doPut(request, response);     break;
      case "DELETE":  doDelete(request, response);  break;
      case "OPTIONS": doOptions(request, response); break;
      case "TRACE":   doTrace(request, response);   break;
      default:
        response.sendError(HttpServletResponse.SC_NOT_IMPLEMENTED);
    }
  }

  @Override
  protected void destroy() {
    // custom destruction code
  }
}
```

Source: Complete Java Servlets Tutorial (December 2020).

HANDLING SERVLET REQUESTS AND RESPONSES

Servlets make it easy to create web applications that adhere to a request and response life cycle. They have the ability to provide HTTP responses and also process business logic within the same body of code. The ability to process business logic makes servlets much more powerful than standard HTML code.

Source: Complete Java Servlets Tutorial (December 2020).

```java
@WebServlet()
public class ProductServlet extends HttpServlet {
  @Override
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/plain");
    // ...
    String query = request.getParameter("query");
    String name  = request.getParameter("name");
    // ...
    String agent = request.getHeader("User-Agent");
    Date since   = new Date(request.getDateHeader("If-Modified-Since"));
    // ...
    if (...) {
      response.sendRedirect("/anotherURL");
      return;
    }
    // ...
    try (PrintStream out = new PrintStream(response.getOutputStream(), true)) {
      if (...) {
        // ...
        response.addHeader("Custom-Header", ...);
        response.addDateHeader("Date", date);
        out.println(output);
      } else {
        response.setStatus(HttpServletResponse.SC_BAD_REQUEST);
      }
    }
  }

  @Override
  protected void doPut(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("application/json");
    try (Scanner in = new Scanner(request.getInputStream())) {
      StringBuffer sb = new StringBuffer();
      while (in.hasNextLine()) {
        sb.append(in.nextLine());
      }
      // ...
    }
  }
}
```

HANDLING SERVLET REQUESTS AND RESPONSES

Servlets make it easy to create web applications that adhere to a request and response life cycle. They have the ability to provide HTTP responses and also process business logic within the same body of code. The ability to process business logic makes servlets much more powerful than standard HTML code.

- Obtaining the request parameters.

Source: Complete Java Servlets Tutorial (December 2020).

```java
@WebServlet()
public class ProductServlet extends HttpServlet {
  @Override
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/plain");
    // ...
    String query = request.getParameter("query");
    String name  = request.getParameter("name");
    // ...
    String agent = request.getHeader("User-Agent");
    Date since   = new Date(request.getDateHeader("If-Modified-Since"));
    // ...
    if (...) {
      response.sendRedirect("/anotherURL");
      return;
    }
    // ...
    try (PrintStream out = new PrintStream(response.getOutputStream(), true)) {
      if (...) {
        // ...
        response.addHeader("Custom-Header", ...);
        response.addDateHeader("Date", date);
        out.println(output);
      } else {
        response.setStatus(HttpServletResponse.SC_BAD_REQUEST);
      }
    }
  }

  @Override
  protected void doPut(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("application/json");
    try (Scanner in = new Scanner(request.getInputStream())) {
      StringBuffer sb = new StringBuffer();
      while (in.hasNextLine()) {
        sb.append(in.nextLine());
      }
      // ...
    }
  }
}
```

HANDLING SERVLET REQUESTS AND RESPONSES

Servlets make it easy to create web applications that adhere to a request and response life cycle. They have the ability to provide HTTP responses and also process business logic within the same body of code. The ability to process business logic makes servlets much more powerful than standard HTML code.

- Obtaining the request parameters.
- Reading in the HTTP request body.

Source: Complete Java Servlets Tutorial (December 2020).

```java
@WebServlet()
public class ProductServlet extends HttpServlet {
  @Override
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/plain");
    // ...
    String query = request.getParameter("query");
    String name  = request.getParameter("name");
    // ...
    String agent = request.getHeader("User-Agent");
    Date since   = new Date(request.getDateHeader("If-Modified-Since"));
    // ...
    if (...) {
      response.sendRedirect("/anotherURL");
      return;
    }
    // ...
    try (PrintStream out = new PrintStream(response.getOutputStream(), true)) {
      if (...) {
        // ...
        response.addHeader("Custom-Header", ...);
        response.addDateHeader("Date", date);
        out.println(output);
      } else {
        response.setStatus(HttpServletResponse.SC_BAD_REQUEST);
      }
    }
  }

  @Override
  protected void doPut(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("application/json");
    try (Scanner in = new Scanner(request.getInputStream())) {
      StringBuffer sb = new StringBuffer();
      while (in.hasNextLine()) {
        sb.append(in.nextLine());
      }
      // ...
    }
  }
}
```

HANDLING SERVLET REQUESTS AND RESPONSES

Servlets make it easy to create web applications that adhere to a request and response life cycle. They have the ability to provide HTTP responses and also process business logic within the same body of code. The ability to process business logic makes servlets much more powerful than standard HTML code.

- Obtaining the request parameters.
- Reading in the HTTP request body.
- Obtaining the request headers.

Source: Complete Java Servlets Tutorial (December 2020).

```java
@WebServlet()
public class ProductServlet extends HttpServlet {
  @Override
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/plain");
    // ...
    String query = request.getParameter("query");
    String name  = request.getParameter("name");
    // ...
    String agent = request.getHeader("User-Agent");
    Date since   = new Date(request.getDateHeader("If-Modified-Since"));
    // ...
    if (...) {
      response.sendRedirect("/anotherURL");
      return;
    }
    // ...
    try (PrintStream out = new PrintStream(response.getOutputStream(), true)) {
      if (...) {
        // ...
        response.addHeader("Custom-Header", ...);
        response.addDateHeader("Date", date);
        out.println(output);
      } else {
        response.setStatus(HttpServletResponse.SC_BAD_REQUEST);
      }
    }
  }

  @Override
  protected void doPut(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("application/json");
    try (Scanner in = new Scanner(request.getInputStream())) {
      StringBuffer sb = new StringBuffer();
      while (in.hasNextLine()) {
        sb.append(in.nextLine());
      }
      // ...
    }
  }
}
```

HANDLING SERVLET REQUESTS AND RESPONSES

Servlets make it easy to create web applications that adhere to a request and response life cycle. They have the ability to provide HTTP responses and also process business logic within the same body of code. The ability to process business logic makes servlets much more powerful than standard HTML code.

- Obtaining the request parameters.
- Reading in the HTTP request body.
- Obtaining the request headers.
- Sending the body of the HTTP response.

Source: Complete Java Servlets Tutorial (December 2020).

```java
@WebServlet()
public class ProductServlet extends HttpServlet {
  @Override
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/plain");
    // ...
    String query = request.getParameter("query");
    String name  = request.getParameter("name");
    // ...
    String agent = request.getHeader("User-Agent");
    Date since   = new Date(request.getDateHeader("If-Modified-Since"));
    // ...
    if (...) {
      response.sendRedirect("/anotherURL");
      return;
    }
    // ...
    try (PrintStream out = new PrintStream(response.getOutputStream(), true)) {
      if (...) {
        // ...
        response.addHeader("Custom-Header", ...);
        response.addDateHeader("Date", date);
        out.println(output);
      } else {
        response.setStatus(HttpServletResponse.SC_BAD_REQUEST);
      }
    }
  }

  @Override
  protected void doPut(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("application/json");
    try (Scanner in = new Scanner(request.getInputStream())) {
      StringBuffer sb = new StringBuffer();
      while (in.hasNextLine()) {
        sb.append(in.nextLine());
      }
      // ...
    }
  }
}
```

HANDLING SERVLET REQUESTS AND RESPONSES

Servlets make it easy to create web applications that adhere to a request and response life cycle. They have the ability to provide HTTP responses and also process business logic within the same body of code. The ability to process business logic makes servlets much more powerful than standard HTML code.

- Obtaining the request parameters.
- Reading in the HTTP request body.
- Obtaining the request headers.
- Sending the body of the HTTP response.
- Setting the HTTP response content type.

Source: Complete Java Servlets Tutorial (December 2020).

```java
@WebServlet()
public class ProductServlet extends HttpServlet {
  @Override
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/plain");
    // ...
    String query = request.getParameter("query");
    String name  = request.getParameter("name");
    // ...
    String agent = request.getHeader("User-Agent");
    Date since   = new Date(request.getDateHeader("If-Modified-Since"));
    // ...
    if (...) {
      response.sendRedirect("/anotherURL");
      return;
    }
    // ...
    try (PrintStream out = new PrintStream(response.getOutputStream(), true)) {
      if (...) {
        // ...
        response.addHeader("Custom-Header", ...);
        response.addDateHeader("Date", date);
        out.println(output);
      } else {
        response.setStatus(HttpServletResponse.SC_BAD_REQUEST);
      }
    }
  }

  @Override
  protected void doPut(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("application/json");
    try (Scanner in = new Scanner(request.getInputStream())) {
      StringBuffer sb = new StringBuffer();
      while (in.hasNextLine()) {
        sb.append(in.nextLine());
      }
      // ...
    }
  }
}
```

HANDLING SERVLET REQUESTS AND RESPONSES

Servlets make it easy to create web applications that adhere to a request and response life cycle. They have the ability to provide HTTP responses and also process business logic within the same body of code. The ability to process business logic makes servlets much more powerful than standard HTML code.

- Obtaining the request parameters.
- Reading in the HTTP request body.
- Obtaining the request headers.
- Sending the body of the HTTP response.
- Setting the HTTP response content type.
- Setting the HTTP response status code.

Source: Complete Java Servlets Tutorial (December 2020).

```java
@WebServlet()
public class ProductServlet extends HttpServlet {
  @Override
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/plain");
    // ...
    String query = request.getParameter("query");
    String name  = request.getParameter("name");
    // ...
    String agent = request.getHeader("User-Agent");
    Date since   = new Date(request.getDateHeader("If-Modified-Since"));
    // ...
    if (...) {
      response.sendRedirect("/anotherURL");
      return;
    }
    // ...
    try (PrintStream out = new PrintStream(response.getOutputStream(), true)) {
      if (...) {
        // ...
        response.addHeader("Custom-Header", ...);
        response.addDateHeader("Date", date);
        out.println(output);
      } else {
        response.setStatus(HttpServletResponse.SC_BAD_REQUEST);
      }
    }
  }

  @Override
  protected void doPut(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("application/json");
    try (Scanner in = new Scanner(request.getInputStream())) {
      StringBuffer sb = new StringBuffer();
      while (in.hasNextLine()) {
        sb.append(in.nextLine());
      }
      // ...
    }
  }
}
```

HANDLING SERVLET REQUESTS AND RESPONSES

Servlets make it easy to create web applications that adhere to a request and response life cycle. They have the ability to provide HTTP responses and also process business logic within the same body of code. The ability to process business logic makes servlets much more powerful than standard HTML code.

- Obtaining the request parameters.
- Reading in the HTTP request body.
- Obtaining the request headers.
- Sending the body of the HTTP response.
- Setting the HTTP response content type.
- Setting the HTTP response status code.
- Addding HTTP response headers.

Source: Complete Java Servlets Tutorial (December 2020).

```java
@WebServlet()
public class ProductServlet extends HttpServlet {
  @Override
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/plain");
    // ...
    String query = request.getParameter("query");
    String name  = request.getParameter("name");
    // ...
    String agent = request.getHeader("User-Agent");
    Date since   = new Date(request.getDateHeader("If-Modified-Since"));
    // ...
    if (...) {
      response.sendRedirect("/anotherURL");
      return;
    }
    // ...
    try (PrintStream out = new PrintStream(response.getOutputStream(), true)) {
      if (...) {
        // ...
        response.addHeader("Custom-Header", ...);
        response.addDateHeader("Date", date);
        out.println(output);
      } else {
        response.setStatus(HttpServletResponse.SC_BAD_REQUEST);
      }
    }
  }

  @Override
  protected void doPut(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("application/json");
    try (Scanner in = new Scanner(request.getInputStream())) {
      StringBuffer sb = new StringBuffer();
      while (in.hasNextLine()) {
        sb.append(in.nextLine());
      }
      // ...
    }
  }
}
```

HANDLING SERVLET REQUESTS AND RESPONSES

Servlets make it easy to create web applications that adhere to a request and response life cycle. They have the ability to provide HTTP responses and also process business logic within the same body of code. The ability to process business logic makes servlets much more powerful than standard HTML code.

- Obtaining the request parameters.
- Reading in the HTTP request body.
- Obtaining the request headers.
- Sending the body of the HTTP response.
- Setting the HTTP response content type.
- Setting the HTTP response status code.
- Addding HTTP response headers.
- Redirecting the client to a different URL.

Source: Complete Java Servlets Tutorial (December 2020).

```java
@WebServlet()
public class ProductServlet extends HttpServlet {
  @Override
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/plain");
    // ...
    String query = request.getParameter("query");
    String name  = request.getParameter("name");
    // ...
    String agent = request.getHeader("User-Agent");
    Date since   = new Date(request.getDateHeader("If-Modified-Since"));
    // ...
    if (...) {
      response.sendRedirect("/anotherURL");
      return;
    }
    // ...
    try (PrintStream out = new PrintStream(response.getOutputStream(), true)) {
      if (...) {
        // ...
        response.addHeader("Custom-Header", ...);
        response.addDateHeader("Date", date);
        out.println(output);
      } else {
        response.setStatus(HttpServletResponse.SC_BAD_REQUEST);
      }
    }
  }

  @Override
  protected void doPut(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("application/json");
    try (Scanner in = new Scanner(request.getInputStream())) {
      StringBuffer sb = new StringBuffer();
      while (in.hasNextLine()) {
        sb.append(in.nextLine());
      }
      // ...
    }
  }
}
```

WRITING AND READING COOKIES USING SERVLETS

```
Cookie cookie = new Cookie("sessionId","123456789");
cookie.setHttpOnly(true);
cookie.setMaxAge(-30);
response.addCookie(cookie);
// ...
Cookie[] cookies = request.getCookies();
for(Cookie cookie : cookies) {
  // cookie.getName();
  // cookie.getValue();
}
```

To create a cookie, simply instantiate a new `javax.servlet.http.Cookie` object and assign a name and value to it. Once the cookie has been instantiated, properties can be set that will help to configure the cookie. IThe cookie's `setMaxAge()` and `setHttpOnly()` methods can be called to set the time of life for the cookie and ensure that it will be guarded against client-side scripting.

ADDING AND SETTING SESSIONS USING SERVLETS

```
HttpSession session = request.getSession(true);
String username = (String)session.getAttribute("username");
session.setAttribute("username", username);
session.removeAttribute('old_value');
session.invalidate();
```

To create and retrieve a session, call the `request.getSession(true)` method to obtain the `javax.servlet.http.HttpSession` object. If it isn't created yet, create it otherwise return the existing session. You can get and set as many as session attributes as you like. Each attribute has a unique name associated with its assigned value. If unset, the `getAttribute()` method returns `null`. The server can invalidate the session at any time.

THE `web.xml` DEPLOYMENT DESCRIPTORS FILE

The `web.xml` file is a standard configuration file used to specify details and configuration regarding your server and its servlets. It allows your webapp to be portable, i.e. moved to a different server with different configurations or a different file path and still work with only minor changes to the configuration file. The webapp shouldn't need to be re-compiled or edited to run in a different environment.

Java web applications use a deployment descriptor file to determine how URLs map to servlets, which URLs require authentication, and other information. This file is named `web.xml`, and resides in the app's `WEB-INF/` directory. `web.xml` is part of the servlet standard for web applications.

A web application's deployment descriptor describes the classes, resources and configuration of the application and how the web server uses them to serve web requests. When the web server receives a request for the application, it uses the deployment descriptor to map the URL of the request to the code that ought to handle the request.

Source: Configuring the web.xml Deployment Descriptor and A web.xml Deployment Descriptor Elements.

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                             http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0
         version="4.0">
  <context-param>
    <param-name>teamFont</param-name>
    <param-value>Helvetica</param-value>
  </context-param>
  ...
  <servlet>
    <servlet-name>redteam</servlet-name>
    <servlet-class>mysite.server.TeamServlet</servlet-class>
    <init-param>
      <param-name>teamColor</param-name>
      <param-value>red</param-value>
    </init-param>
    ...
  </servlet>
  ...
  <servlet-mapping>
    <servlet-name>redteam</servlet-name>
    <url-pattern>/red/*</url-pattern>
  </servlet-mapping>
  ...
</web-app>
```

THE `web.xml` DEPLOYMENT DESCRIPTORS FILE

The `web.xml` file is a standard configuration file used to specify details and configuration regarding your server and its servlets. It allows your webapp to be portable, i.e. moved to a different server with different configurations or a different file path and still work with only minor changes to the configuration file. The webapp shouldn't need to be re-compiled or edited to run in a different environment.

Java web applications use a deployment descriptor file to determine how URLs map to servlets, which URLs require authentication, and other information. This file is named `web.xml`, and resides in the app's `WEB-INF/` directory. `web.xml` is part of the servlet standard for web applications.

A web application's deployment descriptor describes the classes, resources and configuration of the application and how the web server uses them to serve web requests. When the web server receives a request for the application, it uses the deployment descriptor to map the URL of the request to the code that ought to handle the request.

Source: Configuring the web.xml Deployment Descriptor and A web.xml Deployment Descriptor Elements.

```java
@WebServlet()
public class TeamServlet extends HttpServlet {
  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response)
      throws ServletException, IOException {
    // ...
    getServletContext().getInitParameter("teamFont"); // "Helvetica
    getServletConfig().getInitParameter("teamColor"); // "red"
    getServletConfig().getInitParameter("bgColor");   // "#CC0000"
    // ...
  }
}
```

- The servlet can access its initialization parameters by getting its servlet configuration using its own `getServletConfig()` method, then calling the `getInitParameter()` method on the configuration object using the name of the parameter as an argument.
- For shared initialization parameters, they can be retrieved by calling the `getServletContext()` method, then calling the `getInitParameter()` method on the context object using the name of the parameter as an argument.

WEB APIS

RESTful APIs and other Web APIs

WHAT IS REST?

REPRESENTATIONAL STATE TRANSFER

Representational State Transfer (REST) is an architectural style that abstracts elements within a distributed hypermedia system. REST ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements. It encompasses the fundamental constraints upon components, connectors, and data that define the basis of the Web architecture, and thus the essence of its behavior as a network-based application.

The key abstraction of information in REST is a **resource**. Any information that can be named can be a resource: a document or image, a temporal service (e.g. “today’s weather in Los Angeles”), a collection of other resources, a non-virtual object (e.g. a person), and so on. In other words, any concept that might be the target of an author’s hypertext reference must fit within the definition of a resource. A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time.

Source: What is REST; and Architectural Styles and the Design of Network-based Software Architectures by Roy Thomas Fielding (2000).

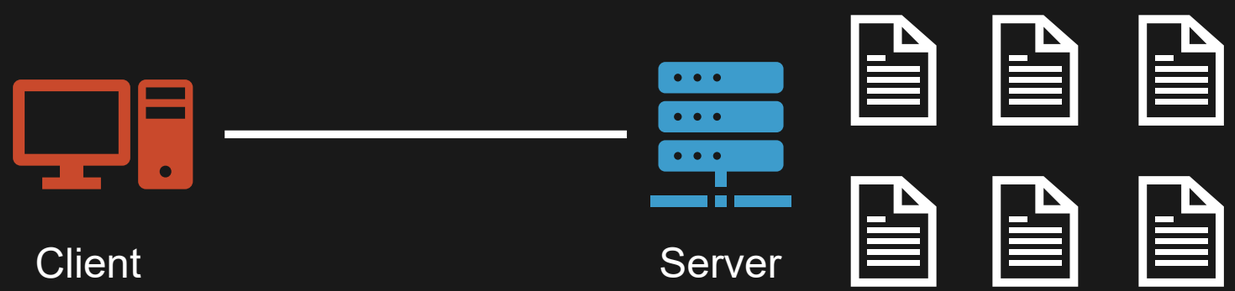
GUIDING PRINCIPLES OF REST

REST has its own guiding principles and constraints. These principles must be satisfied if a service interface needs to be referred to as RESTful. REST does not enforce any rule regarding how it should be implemented at the lower level, it just put high-level design guidelines and leaves us to think of our own implementation. The six guiding principles or constraints of the RESTful architecture are:

1. CLIENT-SERVER

Client applications and server applications **must** be able to evolve separately without any dependency on each other. A client should know only resource URIs, and that's all. Servers and clients may also be replaced and developed independently, as long as the interface between them is not altered.

Separation of concerns is the principle behind the client-server constraints. By separating the user interface concerns from the data storage concerns, we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components. Perhaps most significant to the Web, however, is that the separation allows the components to evolve independently, thus supporting the Internet-scale requirement of multiple organizational domains.



Source: What is REST; and Architectural Styles and the Design of Network-based Software Architectures by Roy Thomas Fielding (2000).

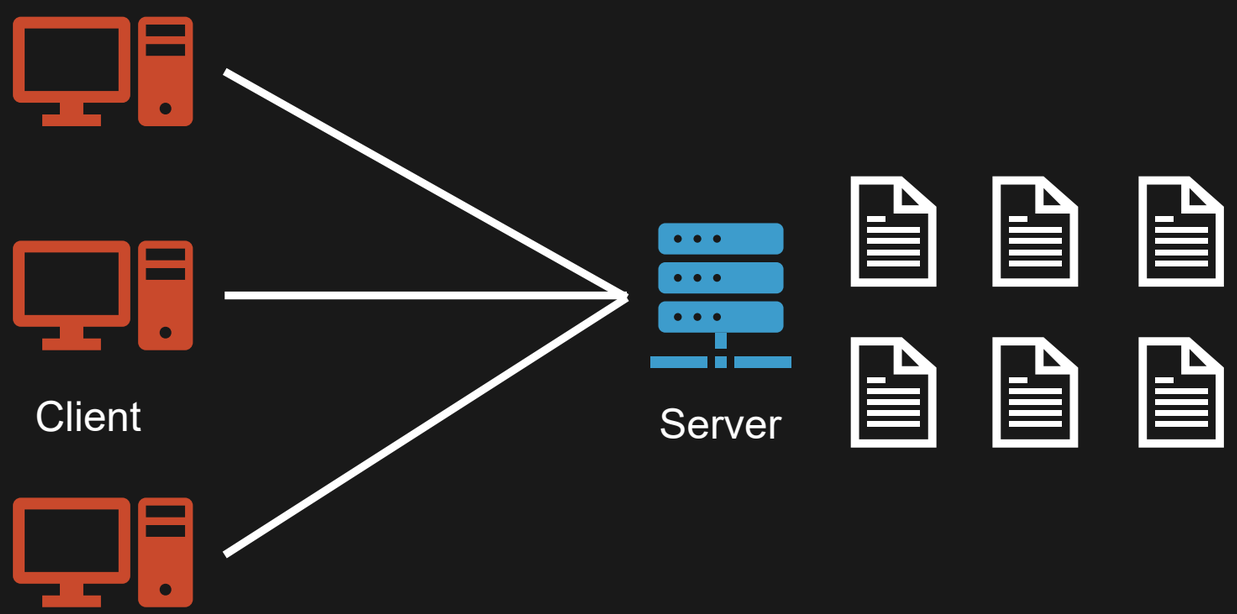
GUIDING PRINCIPLES OF REST

REST has its own guiding principles and constraints. These principles must be satisfied if a service interface needs to be referred to as RESTful. REST does not enforce any rule regarding how it should be implemented at the lower level, it just put high-level design guidelines and leaves us to think of our own implementation. The six guiding principles or constraints of the RESTful architecture are:

2. STATELESS

Communication must be stateless in nature, such that each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client.

This induces the properties of visibility, reliability, and scalability. Visibility: a monitoring system does not have to look beyond a single request in order to determine the full nature of the request. Reliability: it eases the task of recovering from partial failures. Scalability: not having to store state between requests allows the server component to quickly free resources, and further simplifies implementation because the server doesn't have to manage resource usage across requests.



Source: What is REST; and Architectural Styles and the Design of Network-based Software Architectures by Roy Thomas Fielding (2000).

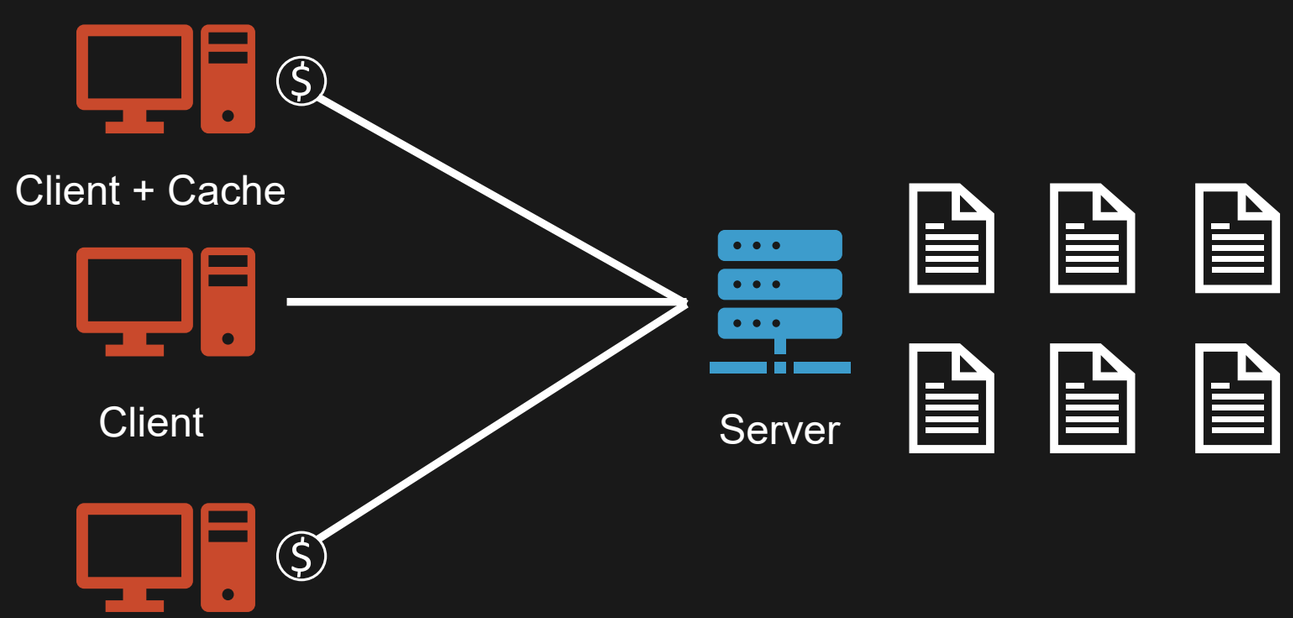
GUIDING PRINCIPLES OF REST

REST has its own guiding principles and constraints. These principles must be satisfied if a service interface needs to be referred to as RESTful. REST does not enforce any rule regarding how it should be implemented at the lower level, it just put high-level design guidelines and leaves us to think of our own implementation. The six guiding principles or constraints of the RESTful architecture are:

3. CACHEABLE

To improve network efficiency, cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.

The advantage of adding cache constraints is that they have the potential to partially or completely eliminate some interactions, improving efficiency, scalability, and user-perceived performance by reducing the average latency of a series of interactions. The trade-off, however, is that a cache can decrease reliability if stale data within the cache differs significantly from the data that would have been obtained had the request been sent directly to the server.



Source: What is REST; and Architectural Styles and the Design of Network-based Software Architectures by Roy Thomas Fielding (2000).

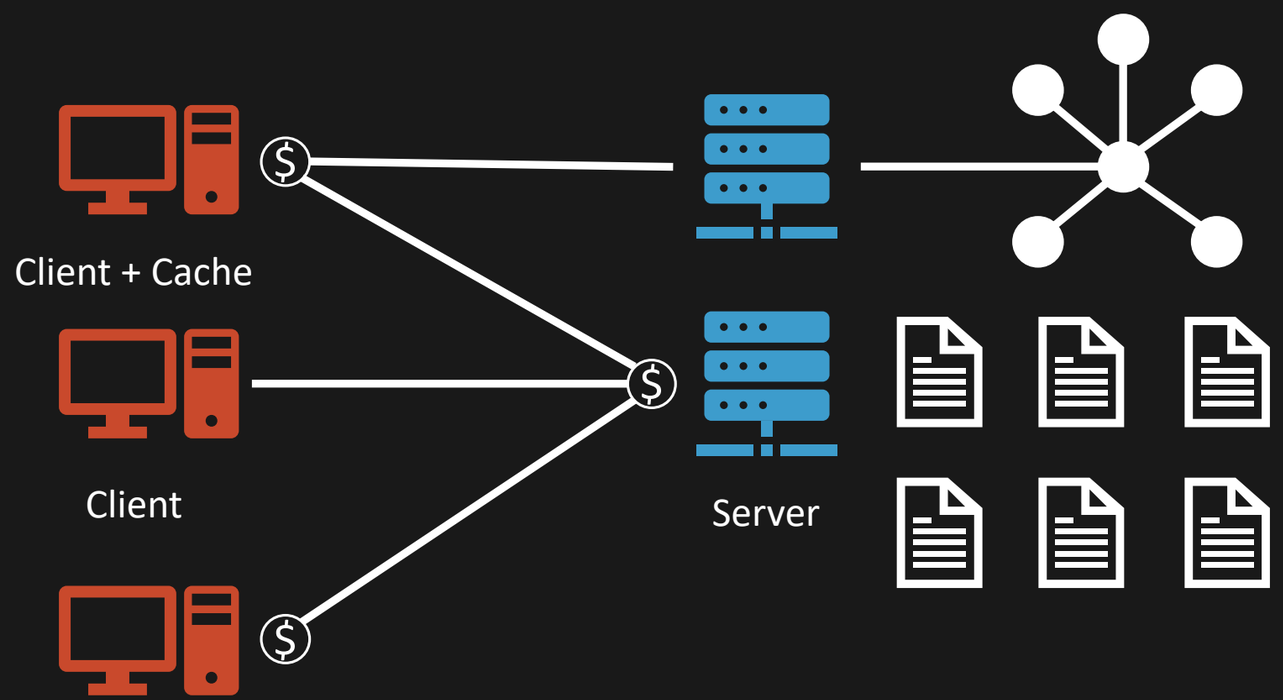
GUIDING PRINCIPLES OF REST

REST has its own guiding principles and constraints. These principles must be satisfied if a service interface needs to be referred to as RESTful. REST does not enforce any rule regarding how it should be implemented at the lower level, it just put high-level design guidelines and leaves us to think of our own implementation. The six guiding principles or constraints of the RESTful architecture are:

4. UNIFORM INTERFACE

The software engineering principle of generality is applied to the component interfaces simplifying the overall system architecture and improving visibility of interactions. Implementations are decoupled from the services they provide, which encourages independent evolvability.

REST is defined by four interface constraints in order to obtain a uniform interface: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state.



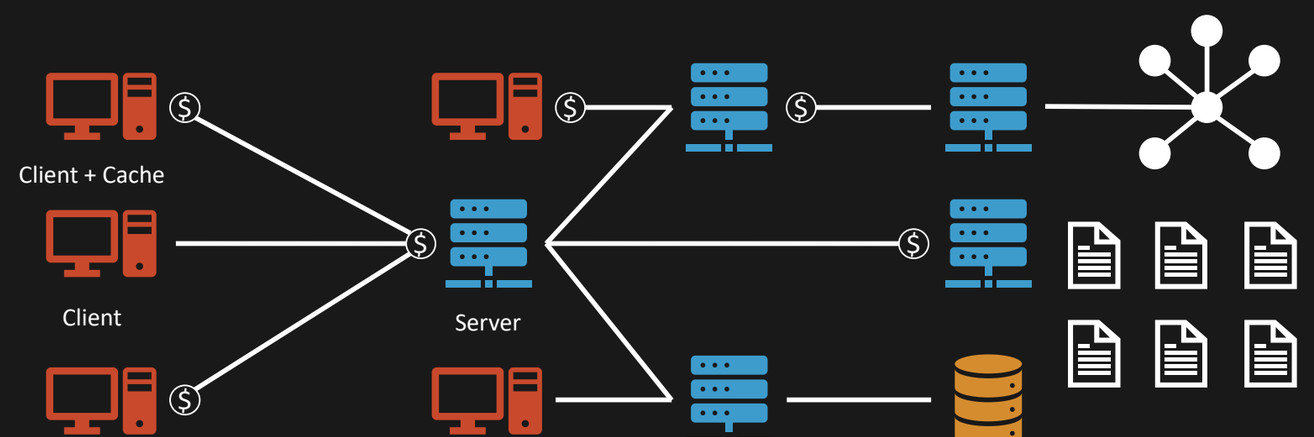
Source: What is REST; and Architectural Styles and the Design of Network-based Software Architectures by Roy Thomas Fielding (2000).

GUIDING PRINCIPLES OF REST

REST has its own guiding principles and constraints. These principles must be satisfied if a service interface needs to be referred to as RESTful. REST does not enforce any rule regarding how it should be implemented at the lower level, it just put high-level design guidelines and leaves us to think of our own implementation. The six guiding principles or constraints of the RESTful architecture are:

5. LAYERED SYSTEM

In order to further improve behavior for Internet-scale requirements, the layered system contraint allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot “see” beyond the immediate layer with which they are interacting. By restricting knowledge of the system to a single layer, we place a bound on the overall system complexity and promote substrate independence. Layers can be used to encapsulate legacy services and to protect new services from legacy clients, simplifying components by moving infrequently used functionality to a shared intermediary. Intermediaries can also be used to improve system scalability by enabling load balancing of services across multiple networks and processors.



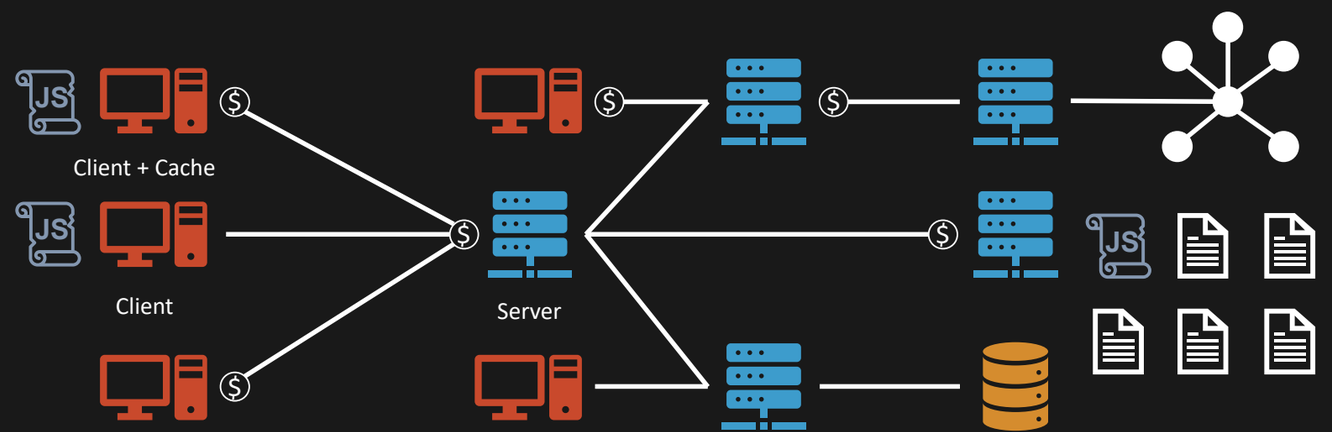
Source: What is REST; and Architectural Styles and the Design of Network-based Software Architectures by Roy Thomas Fielding (2000).

GUIDING PRINCIPLES OF REST

REST has its own guiding principles and constraints. These principles must be satisfied if a service interface needs to be referred to as RESTful. REST does not enforce any rule regarding how it should be implemented at the lower level, it just put high-level design guidelines and leaves us to think of our own implementation. The six guiding principles or constraints of the RESTful architecture are:

6. CODE-ON-DEMAND

REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented. Allowing features to be downloaded after deployment improves system extensibility. However, it also reduces visibility, and thus is only an optional constraint within REST.



Source: What is REST; and Architectural Styles and the Design of Network-based Software Architectures by Roy Thomas Fielding (2000).

CREATE
READ
UPDATE
DELETE

When we are building APIs, we want our models to provide four basic types of functionality. The model must be able to Create, Read, Update, and Delete resources. In a REST environment, CRUD often corresponds to the HTTP methods `PUT` / `POST`, `GET`, `PUT`, and `DELETE`, respectively. These are the fundamental elements of a persistent storage system.

Despite the popular usage and widespread misconception, `POST` is not the "correct method for creating resource". The semantics of other methods are determined by the HTTP protocol, but the semantics of `POST` are determined by the target media type itself. `POST` is the method used for any operation that isn't standardized by HTTP, so it can be used for creation, but also can be used for updates, or anything else that isn't already done by some other method.

`PUT` is not the "correct method for updating resource". `PUT` is the method used to replace a resource completely, ignoring its current state. You can use `PUT` for creation if you have the whole representation the server expects, and you can use `PUT` for update if you provide a full representation, including the parts that you won't change, but it's not correct to use `PUT` for partial updates, because you're asking for the server to consider the current state of the resource. `PATCH` is the method to do that.

Source: What is CRUD?; and S3 REST API and POST method - StackOverflow.

RESTFUL API

CRUD	HTTP
Create	`PUT`
Read	`GET`
Update	`PUT`
Delete	`DELETE`

DATABASES - SQL

CRUD	SQL
Create	`INSERT`
Read	`SELECT`
Update	`UPDATE`
Delete	`DELETE`

REST API EXAMPLE

`PUT /products`	Create a new Product.
`GET /products`	Retrieve a list of all Products.
`GET /product/<id>`	Retrieve the Product with the given `<id>`.
`PUT /product/<id>`	Update the Product with the given `<id>`.
`DELETE /product/<id>`	Delete the Product with the given `<id>`.

OTHER WEB APIS

SIMPLE OBJECT ACCESS PROTOCOL (SOAP)

Simple Object Access Protocol (SOAP) is a lightweight protocol for the exchange of information in a decentralized, distributed environment. It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it; a set of encoding rules for expressing instances of application-defined data types; and a convention for representing remote procedure calls and responses.

- A communication protocol designed to communicate via Internet.
- Extends HTTP for XML messaging.
- Provides data transport for Web services.
- Exchanges complete documents or calls remote procedures.
- Can be used for broadcasting a message.
- Both platform and language independent.
- The XML way of defining what information is sent and how.
- Enables client applications to easily connect to remote services and invoke remote methods.

EXAMPLE REQUEST

```
<?xml version="1.0" ?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
               soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
  <soap:Body>
    <m:GetPrice xmlns:m="https://www.example.com/prices">
      <m:Item>Apples</m:Item>
    </m:GetPrice>
  </soap:Body>
</soap:Envelope>
```

EXAMPLE RESPONSE

```
<?xml version="1.0" ?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
               soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
  <soap:Body>
    <m:GetPriceResponse xmlns:m="https://www.example.com/prices">
      <m:Price>1.90</m:Price>
    </m:GetPriceResponse>
  </soap:Body>
</soap:Envelope>
```

WEB SERVICES DESCRIPTION LANGUAGE (WSDL)

A **Web Services Description Language (WSDL)** document is a standard way of describing a web service. A WSDL file is written in XML, and it defines the location of the web service, its operations (methods), the messages used by each operation, and the XML elements, or data types, within.

WSDL is often used in combination with SOAP and XML Schema to provide web services over the Internet. A client program connecting to a web service can read the WSDL to determine what functions are available on the server. Any special datatypes used are embedded in the WSDL file in the form of XML Schema. The client can then use SOAP to actually call one of the functions listed in the WSDL.

A WSDL file typically consists of the following sections:

`types`	Which defines the data types (XML elements) that are used by the web service.
`message`	Each of which defines a message exchanged with the web service.
`portType`	Which combine multiple messages into a single operation: for synchronous operations, this is usually one input and one output.
`binding`	Which defines exactly how each operation will take place over the network (SOAP, in this example).
`service`	Which says where the service can be accessed from – in other words, its endpoint.

WSDL EXAMPLE


```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:tns="http://www.example.com/BookService/"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  name="BookService" targetNamespace="http://www.example.com/BookService/">

  <!--  The `types` element defines the data types (XML elements)
        that are used by the web service. -->
  <wsdl:types>
    <xsd:schema targetNamespace="http://www.example.com/BookService/">
      <xsd:element name="Book">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="ID" type="xsd:string" minOccurs="0"/>
            <xsd:element name="Title" type="xsd:string"/>
            <xsd:element name="Author" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="GetBook">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="ID" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="GetBookResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element ref="tns:Book" minOccurs="0" maxOccurs="1"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
  </wsdl:types>
  ...
```

```
  <!--  A wsdl `message` element is used to define a message
        exchanged between a web service, consisting of zero
        or more `part`s. -->
  <wsdl:message name="GetBookRequest">
    <wsdl:part element="tns:GetBook" name="parameters"/>
  </wsdl:message>
  <wsdl:message name="GetBookResponse">
    <wsdl:part element="tns:GetBookResponse" name="parameters"/>
  </wsdl:message>

  <!--  A WSDL `portType` is used to combine multiple `message`s
        (e.g. input, output) into a single operation.
        Here we define three synchronous (input/output) operations
        and the `message`s that must be used for each. -->
  <wsdl:portType name="BookService">
    <wsdl:operation name="GetBook">
      <wsdl:input message="tns:GetBookRequest"/>
      <wsdl:output message="tns:GetBookResponse"/>
    </wsdl:operation>
  </wsdl:portType>

  <!--  The `binding` element defines exactly how each
        `operation` will take place over the network.
        In this case, we are using SOAP. -->
  <wsdl:binding name="BookServiceSOAP" type="tns:BookService">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="GetBook">
      <soap:operation soapAction="http://www.example.com/BookService/GetBook"/>
      <wsdl:input><soap:body use="literal"/></wsdl:input>
      <wsdl:output><soap:body use="literal"/></wsdl:output>
    </wsdl:operation>
  </wsdl:binding>

  <!--  The `service` element finally says where the service
        can be accessed from - in other words, its endpoint. -->
  <wsdl:service name="BookService">
    <wsdl:port binding="tns:BookServiceSOAP" name="BookServiceSOAP">
      <soap:address location="http://www.example.org/BookService"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

MORE WEB APIS

- XML-RPC
- JSON-RPC
- GraphQL
- FALCOR
- gRPC
- RabbitMQ

This slide is intentionally left blank.

Return to Course Page