Overview

Overview of basic architecture of the web:

- Simple textual protocol.
- URI's.
- Basic methods `GET`, `POST`, `HEAD`.
- Web services, additional methods.
- REST

Why Was the WWW Successful

- Numerous earlier attempts like CORBA to build distributed systems.
- WWW succeeded because:
  - it used a very simple protocol with general methods, rather than those specialized to a specific domain.
  - it initially was built for humans; it was only later realized that it could also be used by machines.

HTTP/1.x: a Simple Textual Protocol

- HTTP 1.x is a **text** protocol (not binary).
- It is easy for humans to debug the protocol as the protocol data is directly human-readable.
- Often protocol data consists of header lines separated from textual body by an empty line.
- A header consists simply of a header name separated from its value by a single colon `:`.
- Headers describe type of content.
- Body may need to be encoded especially if it is binary.

An Example using telnet

```
$ telnet www.binghamton.edu 80
...
GET / HTTP/1.0

HTTP/1.1 301 Moved Permanently
Date: ...
...
Location: https://www.binghamton.edu/
Content-Length: ...
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>301 Moved Permanently</title>
...
</body></html>
Connection closed by foreign host.
```

Interacting with a Web Server

- Request consists of a request method like `GET`, a URL (relative to the server) and the version like `HTTP/1.0`. This can be following by zero-or-more *name*: *value* header lines. The request headers are terminated by an empty line. This may be followed by an entity body depending on the request.
- The response is similar except that it starts with a line containing a protocol version and status.
- More modern replacements for `telnet` include netcat and *curl*.
- Purely textual protocol makes it easy for humans to use these kinds of general network programs to interact with web sites.

Identifying and Locating Web Resources

- A **Uniform Resource Identifier** (URI) is an identifier for an abstract or physical resource.
- A **Uniform Resource Locator** (URL) is a URI with an access method which allows locating a resource.
- A **Uniform Resource Name** is a URI which uses specific sub-schemes and uniquely identifies a resource.
- Relative URLs relative to some base.
- Original RFC is quite readable.
- There is confusion about the above differences, URI and URL often used interchangeably; see this.

URI Components

Consider the URI *<http://zdu.binghamton.edu/cgi-bin/echo.pl?name=john&name=mary#label>*

Scheme
: All URI's start with an identifier giving the specification it follows. This is followed by a : char. The example uses scheme `http`.

Authority
: Specifies the naming authority for the resource. Preceeded by a //. The example has the authority `zdu.binghamton.edu`, which corresponds to a hostname in the *domain-name system* (DNS). Can contains user-info (preceeded by an `@`), a host-name or IP address and a port number (preceeded by a :).

URI Components Continued

Path Separated from the authority by a / character. The example has the path `cgi-bin/echo.pl`.
It is terminated by a subsequent ? or # character.

Query Indicated by the first ? after the path and is terminated by a # character (or the end of the URI). The example has the query `name=john&name=mary`.

Fragment Identifies a secondary resource (relative to the primary resource). Follows a # character after the query. The example has a fragment `label`. This is not sent to the server.

URI Examples

```
https://zdu.binghamton.edu:8080/cgi-bin/hello.rb
  ?name1=fred&name2=john#label

http://128.226.116.131/

mailto:umrigar@binghamton.edu

file:///home/umrigar/cs580w/    #absolute paths only

urn:isbn:978-0596517748
```

URI Encoding

- Encode characters which may have reserved meanings within a URL.
- RFC 3986 reserves special characters like /, ? and &.
- Special characters need to be escaped using %*hh* where *hh* is the ASCII code for the character.
  - Slash / represented as %2F.
  - Question-mark ? represented as %3F.
  - Ampersand & represented as %26.
- Alphanumerics, hyphen -, underscore _, period . and tilde ~ never need to be escaped.
- Characters need not be URI-escaped if used within a context where they are not special; for example, / does not need to be escaped within a query string.

JavaScript Encode URI Functions

- encodeURI(*string*) Will encode only those special characters which do not have special use within a URI. So it will not escape characters like /, ?, #. Use to encode entire URI which does not contain special characters within contexts where they have special meaning. Decode using `decodeURI()`.
- encodeURIComponent(*string*) Will encode all characters except -, _, ., !, ~, *, ', (and). Hence safe to use only on URI component. Decode using `decodeURIComponent()`.

JavaScript Encode URI Functions Examples

```
> uri = ’http://www.example.com?q=encode url’
’http://www.example.com?q=encode url’
> encodeURI(uri)
’http://www.example.com?q=encode%20url’
> encodeURIComponent(uri)
’http%3A%2F%2Fwww.example.com%3Fq%3Dencode%20url’
> decodeURI(encodeURI(uri))
’http://www.example.com?q=encode url’
>
```

HTTP Overview

- A **client** makes a **request** for a resource on a **server**.
- A **server** returns a **response** which is a **representation** of the requested resource.
- Both request and response are text containing header lines separated from body by a empty line.
- HTTP does not care about headers it does not understand. **Postel's Principle** ensures *robustness*: *Be conservative in what you do, be liberal in what you accept from others*.
- **Uniform Resource Locators** (URLs) are used for identifying resources.

Stateless Protocol

As far as HTTP goes, no state is stored on the server.

- HTTP does not in any way associate requests from the same client.
- State is maintained by sending some identification information with each request. This is then used to access state stored on the server.
- Identifying information is often sent via cookies or URL parameters.
- Statelessness makes it possible for the protocol to scale.

HTTP Method Properties

Two properties which allow building robust applications in the presence of errors:

Safe method Should not change application state on the server.

Idempotent method Multiple identical requests have the same effect as a single request.

The GET Method

- Requests a representation of a resource.
- Safe and idempotent.
- No body in request.
- Has format `GET` *resource* `HTTP/`*version*, where *resource* is the path to the resource on the server and *version* is the version of the HTTP protocol: `1.1` widely used; `2.0` (binary protocol) is being deployed.
- Can be cached.
- Allowed in HTML forms.

The POST Method

- Sends data to server. Usually used for submitting forms or creating subordinate resources (subordinate to the requested URL).
- No safety or idempotency guarantees.
- If the `Content-Type` header is `application/x-www-form-urlencoded`, then the body consists *name=value* pairs separated by & characters. Non-alphanumeric characters are %-encoded.
- `Content-Type` of `multipart/form-data` often used for binary data as when uploading a file.
- Cannot be cached. Often breaks browser back button on poorly implemented web sites.
- Allowed in HTML forms.

The HEAD Method

- Like `GET` but response does not include a body.
- Used to query the status of a resource.
- Helps with caching.
- Idempotent and safe.
- Cacheable.
- No response body.

The PUT Method

- Can be used for creating or updating resource at specified URI.
- When updating, the specified object completely replaces resource.
- Unsafe but idempotent; hence if the same PUT request is repeated multiple times, the effect is the same as a single PUT request.
- Cannot be cached.
- Not allowed in HTML forms.
- No response body.

The PATCH Method

- Can be used for partial modifications of resource at specified URI.
- Unlike PUT, request body only specifies changes to resource.
- Neither safe nor idempotent; however, there is no reason an application cannot set up PATCH operations to be idempotent.
- Cannot be cached.
- Not allowed in HTML forms.
- No response body.

The DELETE Method

- Used to delete resource specified by URL.
- Unsafe but idempotent; hence if the same DELETE request is repeated multiple times, the effect is the same as a single DELETE request.
- Cannot be cached.
- Not allowed in HTML forms.
- No response body.

Put vs Post for Creation

1. Use PUT when client specifies URL for created resource.
2. Use POST when server specifies URL for created resource. So created resource is subordinate to an existing resource.

HTTP Status Codes

1xx **Informational** messages.

2xx Used to indicate **success**.

3xx Used to indicate **redirection** via the `Location` header.

4xx Used to indicate a **client error**.

5xx Used to indicate a **server error**.

Some Notable Status Codes

See *HTTP Status Codes*:

200 Ok.

201 **Created**. A new resource has been created. Most specific URI for new resource given by `Location` header in response.

204 **No content**. Success but no content.

301 **Moved permanently**. Resource moved permanently to URL specified by `Location` header.

302 **Found**. Moved temporarily to URL specified by `Location` header. Became synonymous with 303.

303 **See other**. Resource can be retrieved by doing a `GET` to URL specified by `Location` header.

304 **Conditional get**. Used for caching.

307 **Moved temporarily** to URL specified by `Location` header.

Some Notable Status Codes Continued

400 **Bad request**. Client sent an incorrect request.

401 **Unauthorized**. Requires authentication.

404 **Not found**. No resource at specified URL.

409 **Conflict**. Request conflicts with current state of resource.

500 **Internal server error**.

Web Services

- The web is one of the most successful distributed systems ever built.
- **Web services** allow access to web resources by programs rather than humans.
- Programs can harvest information from the web by **scraping** information from HTML web pages.
- HTML can be authored so that information can be accessed easily by programs (often true of current web pages), but information is often hidden within text.
- HTML is only one representation for information; other representations like JSON and XML are easily read by programs.
- Additional HTTP methods available for web services (human web largely uses only GET, POST and HEAD).

SOAP

Originally stood for *Simple Object Access Protocol*.

- A style of web services.
- Original motivation appeared to be tunneling through corporate firewalls using web ports.
- Largely *remote procedure call* using HTTP and XML. Many implementations did not really use web architecture.
- Huge collection of standards. Lots of tooling.
- Will not cover further in this course even though it is still quite popular (mainly legacy compatibility).

Representational State Transfer

Representational **S**tate **T**transfer.

- An architectural style.
- Post-documentation of web architectural style by Roy Fielding.
- REST web services use URL's to represent resources and HTTP methods as the actions on the resources.

REST Web Services

Features of REST web services:

- HTTP messages.
- URI's.
- Representations.
- Links (HATEOAS).
- Caching.

Already discussed HTTP messages and URI's.

Representations of Resources

A **resource** can be thought of like an object.

- Objects can contain other objects (value objects). Similarly resources can **embed** other resources.
- Objects can reference other objects (via object references). Similarly resources can **link** to other resources.
- Resources are named by URI's.
- Resources can have multiple representations.

JSON Representation

JSON is a popular way of representing resources.

```
{
  "id": "1234",
  "name": "John Smith",
  "email": "jsmith@mail.example.com"
}
```

XML Representation

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<person>
  <id>1234</id>
  <name>John Smith</name>
  <email>jsmith@mail.example.com</email>
</person>
```

- The first line is a XML declaration.
- `<element>...</element>` is an element.

Alternate XML Representation

Can move **atomic** information into element attributes.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<person id="1234">
  <name>John Smith</name>
  <email>jsmith@mail.example.com</email>
</person>
```

Well-Formed vs Valid XML

- If XML nesting structure syntax is correct, then it is said to be **well-formed**.
- No restriction on vocabulary (element names, attribute names) of well-formed XML.
- It is possible to restrict element and attribute names and their permitted containment relationships using an external specification. XML which meets such restrictions is said to be **valid**. Some alternatives for specifying the restrictions:
  - Document Type Definitions (DTDs).
  - XML Schema.
  - RELAX NG.

Content Negotiation

- Client can indicate what kind of representation it wants by using a specific extension like `.xml` or `.json` in the URL as in

  ```
  http://example.com/api/person.json?id=1234
  http://example.com/api/person.xml?id=1234
  ```

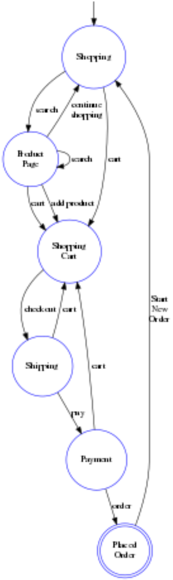
  and the server needs to honor these URLs.
- Client can indicate its preferences using a special `ACCEPT` header in its request:

  ```
  GET /person?id=1234
  ...
  ACCEPT: application/json
  ```

HyperText As The Engine Of Application State

- Acronym HATEOAS.
- The state of an application is maintained in a document (JSON, XML, HTML) returned to a client. This client state is often linked to server-side state using cookies or URLs.
- The document contains links or forms.
- Client transitions to a new state by following a link or filling-in and submitting a form.
- A browser application is a state machine with the browser displaying a window into the current application state and state transitions taken by following links or submitting a form.

HATEOAS for an E-Commerce Site


Shopping
search
continue shopping
Product Page
search
cart
cart
add product
Shopping Cart
checkout
cart
Start New Order
Shipping
cart
pay
Payment
order
Placed Order