# XML Namespaces

The XML-namespace reserved attribute xmlns to create two namespace prefixes.

Each namespace prefix is bound to a series of characters called a **Uniform Resource Identifier (URI)** that uniquely identifies the namespace.

Document authors create their own namespace prefixes and URIs.

The parser does not visit these URLs, nor do these URLs need to refer to actual web pages.

They each simply represent a unique series of characters used to differentiate URI names.

In fact, any string can represent a namespace.

< root xmlns:h=“HighSchool"
xmlns:m=“medicalschool">

<h:subject>Geometry</h:subject>
<m:subject>Cardiology</m:subject>

# Document Type Definitions (DTDs)

Document Type Definitions (DTDs) are one of two main types of documents you can use to specify XML document structure.

An XML document with correct syntax is called "Well Formed".

An XML document validated against a DTD is both "Well Formed" and "Valid".

A DTD describes the structure of an XML document and enables an XML parser to verify whether an XML document is valid (i.e., whether its elements contain the proper attributes and appear in the proper sequence).

DTDs allow users to check document structure and to exchange data in a standardized format.

DTDs are not themselves XML documents.

```
<!ELEMENT letter ( contact+, salutation,
paragraph+,  closing, signature )>

<!ELEMENT contact ( name, address1,
address2, city, state,  zip, phone, flag )>

 <!ATTLIST contact type CDATA
#IMPLIED>

 <!ELEMENT name ( #PCDATA )>
 <!ELEMENT address1 ( #PCDATA )>
 <!ELEMENT address2 ( #PCDATA )>
 <!ELEMENT city ( #PCDATA )>
```

```
<!ELEMENT state ( #PCDATA )>
 <!ELEMENT zip ( #PCDATA )>
 <!ELEMENT phone ( #PCDATA )>
 <!ELEMENT flag EMPTY>
 <!ATTLIST flag gender (M | F) "M">

 <!ELEMENT salutation ( #PCDATA )>
 <!ELEMENT closing ( #PCDATA )>
 <!ELEMENT paragraph ( #PCDATA )>
 <!ELEMENT signature ( #PCDATA )>
```

## Defining Elements in a DTD

The ELEMENT element type declaration defines the rules for element letter.

In this case, letter contains one or more contact elements, one salutation element, one or more paragraph elements, one closing element and one signature element, in that sequence.

The plus sign (+) occurrence indicator specifies that the DTD requires one or more occurrences of an element.

Other occurrence indicators include the asterisk (*), which indicates an optional element that can occur zero or more times,

and the question mark (?), which indicates an optional element that can occur at most once (i.e., zero or one occurrence).

If an element does not have an occurrence indicator, the DTD requires exactly one occurrence.

The contact element type declaration specifies that a contact element contains child elements name, address1, address2, city, state, zip, phone and flag— in that order.

The DTD requires exactly one occurrence of each of these elements.

# Defining Attributes in a DTD

**ATTLIST attribute-list declaration** to define an attribute named type for the contact element.

Keyword **#IMPLIED** specifies that if the parser finds a contact element without a type attribute, the parser can choose an arbitrary value for the attribute or can ignore the attribute.

Other keywords that can be used in place of #IMPLIED in an ATTLIST declaration include #REQUIRED and #FIXED.

Keyword #REQUIRED specifies that the attribute must be present in the element, and keyword #FIXED specifies that the attribute (if present) must have the given fixed value.

**<!ATTLIST** address zip **CDATA #FIXED "01757">**

# Character Data vs. Parsed Character Data

Keyword CDATA specifies that attribute type contains character data (i.e., a string).

A parser will pass such data to an application without modification.

Keyword **#PCDATA** specifies that an element (e.g., name) may contain **parsed character data** (i.e., data that's processed by an XML parser).

Elements with parsed character data cannot contain markup characters, such as less than (<), greater than (>) or ampersand (&).

The document author should replace any markup character in a #PCDATA element with the character's corresponding **character entity reference**.

Keyword EMPTY specifies that the element does not contain any data between its start and end tags.

Empty elements commonly describe data via attributes.

For example, flag's data appears in its gender attribute. The gender attribute's value must be one of the enumerated values (M or F) enclosed in parentheses and delimited by a vertical bar (|) meaning "or."

Note that gender has a default value of M.

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE note SYSTEM "Note.dtd">

<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

```
<!DOCTYPE note
[
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
```

# Using DTD for Entity Declaration

A doctype declaration can also be used to define special characters and character strings, used in the document:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE note [
<!ENTITY nbsp " ">
<!ENTITY writer "Writer: Donald Duck.">
<!ENTITY copyright "Copyright: W3Schools.">
]>

<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
<footer>&writer; &copyright;</footer>
</note>
```

Rich Internet Applications (RIAs) are web applications that approximate the look, feel and usability of desktop applications.

Two key attributes of RIAs are performance and a rich GUI.

RIA performance comes from **Ajax (Asynchronous JavaScript and XML)**, which uses client-side scripting to make web applications more responsive.

Ajax applications separate client-side user interaction and server communication and run them *in parallel*, reducing the delays of server-side processing normally experienced by the user.


AJAX

# AJAX Keywords

- JavaScript and XML
- Web browser technology
- Open web standards
- Browser and platform independent
- Better and faster Internet applications
- XML and HTTP Requests

# Traditional Web Applications

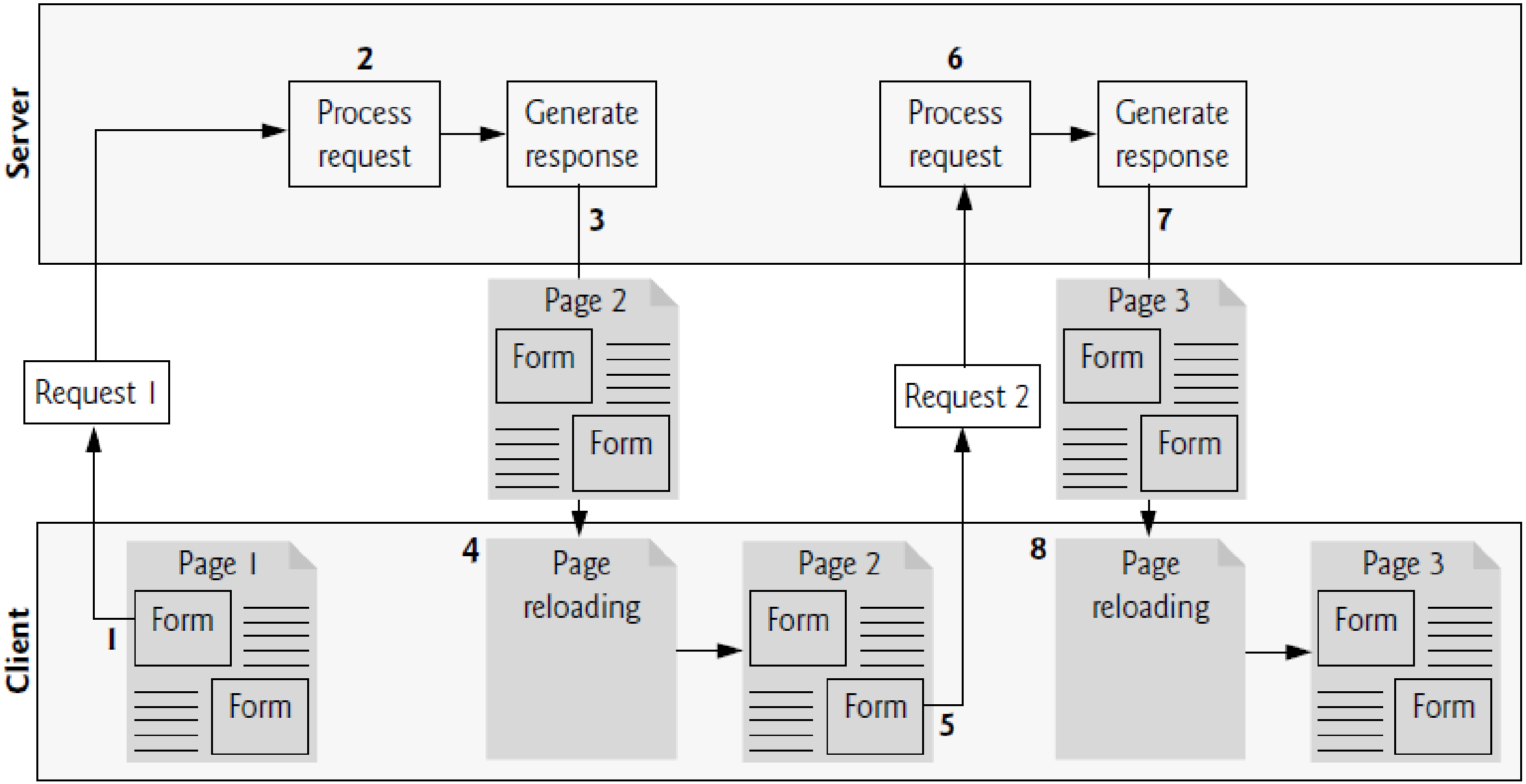
The client and the server in a traditional web application, such as one that employs a user registration form.

1. The user first fills in the form’s fields, then submits the form
2. The browser generates a request to the server, which receives the request and processes it

The server generates and sends a response containing the exact page that the browser will render which causes the browser to load the new page and temporarily makes the browser window blank.

Note that the client *waits* for the server to respond and *reloads the entire page* with the data from the response.

While such a **synchronous request** is being processed on the server, the user *cannot* interact with the client web page.


Server
2
Process request
Generate response
3
6
Process request
Generate response
7
Page 2
Form
Form
Page 3
Form
Form
Request 1
Request 2
Client
1
Page 1
Form
Form
4
Page reloading
Page 2
Form
Form
5
8
Page reloading
Page 3
Form
Form

# Ajax Web Applications

When the user interacts with the page, the client creates an XMLHttpRequest object to manage a request (Step 1).

The XMLHttpRequest object sends the request to the server (Step 2) and awaits the response.

The requests are asynchronous, so the user can continue interacting with the application on the client side while the server processes the earlier request concurrently.

Other user interactions could result in additional requests to the server (Steps 3 and 4).

Once the server responds to the original request (Step 5), the XMLHttpRequest object that issued the request calls a client-side function to process the data returned by the server.

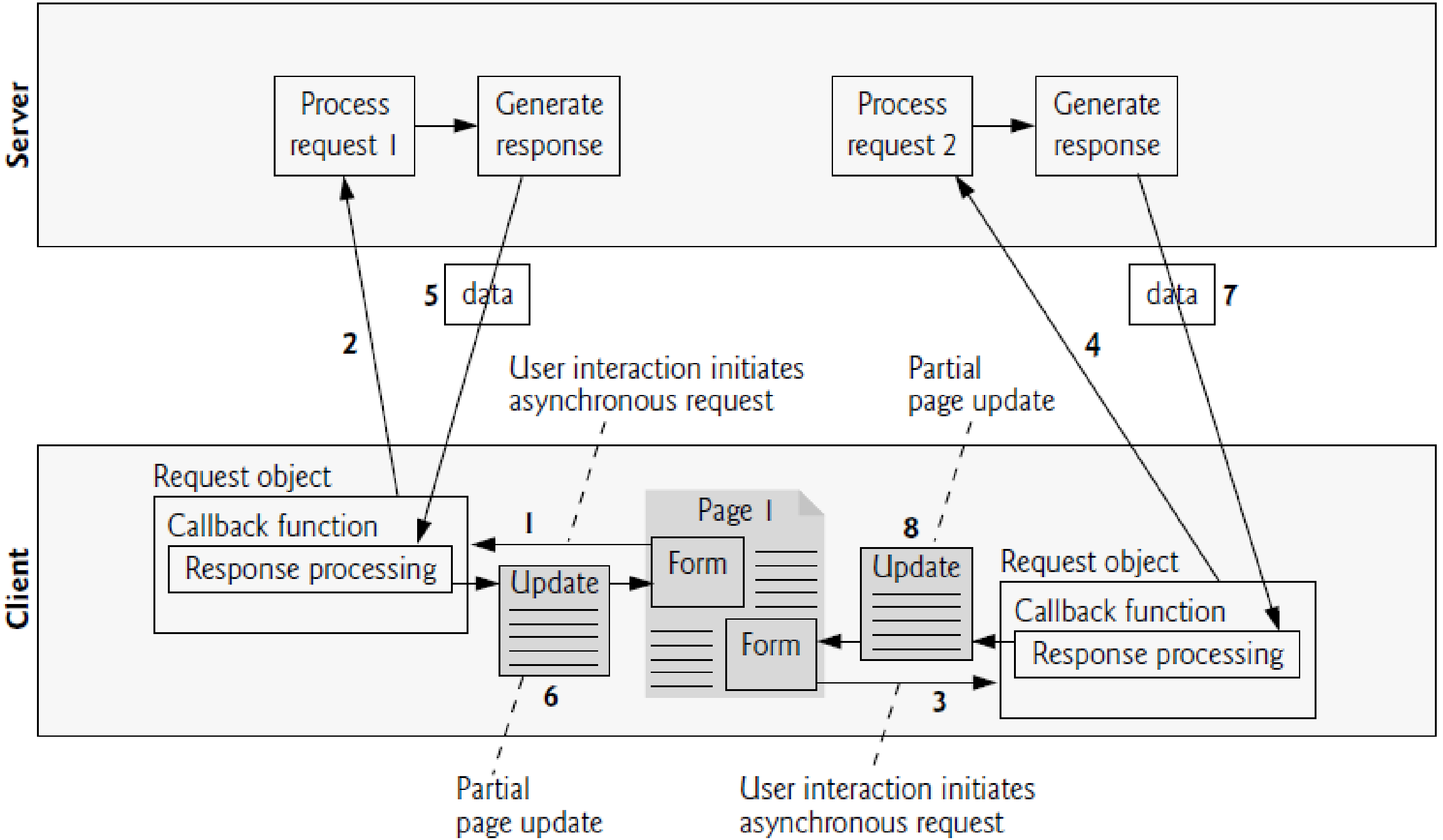
This function—known as a callback function—uses partial page updates (Step 6) to display the data in the existing web page without reloading the entire page.

At the same time, the server may be responding to the second request (Step 7) and the client side may be starting to do another partial page update (Step 8).

The callback function updates only a designated part of the page.

Such partial page updates help make web applications more responsive, making them feel more like desktop applications.

The web application does not load a new page while the user interacts with it.


Server
Process request 1
Generate response
Process request 2
Generate response
5
data
data
7
2
4
User interaction initiates asynchronous request
Partial page update
Client
Request object
Callback function
Response processing
1
Page 1
Form
Update
8
Update
Request object
Callback function
Response processing
Form
6
3
Partial page update
User interaction initiates asynchronous request

# How we interacted with the web till now…

- Typical browsing behavior consists of loading a web page, then selecting some action that we want to do, filling out a form, submitting the information, etc.

- We work in this sequential manner, requesting one page at a time, and have to wait for the server to respond, loading a whole new web page before we continue.

- This is also one of the limitations of web pages, where transmitting information between a client and server generally requires a new page to be loaded.

JavaScript is one way to cut down on (some of) the client-server response time, by using it to verify form (or other) information before it's submitted to a server.

One of the limitations of JavaScript is (or used to be) that there was no way to communicate directly with a web server.

Another drawback to this usual sequential access method  is that there are many situations where you load a new page that shares lots of the same parts as the old (consider the case where you have a "menu bar" on the top or side of the page that doesn't change from page to page).

# What is AJAX?

- AJAX is an acronym for **A**synchronous **J**avaScript **A**nd **X**ML.
- AJAX is not a programming language. but simply a development technique for creating interactive web applications.
- The technology uses JavaScript to send and receive data between a web browser and a web server.
- The AJAX technique makes web pages more responsive by exchanging data with a server behind the scenes, instead of reloading an entire web page each time a user makes a change.
- With AJAX, web applications can be faster, more interactive, and more user friendly.
- AJAX uses an XMLHttpRequest object to send data to a web server, and XML is commonly used as the format for receiving server data, although any format including and plain text can be used.

# AJAX Application Example

- Online test
  - Many multiple choice questions
  - All questions are displayed on one page
  - After the user answers one question, the correct answer and explanation about why the user answer is wrong is shown on the page
  - For all already-answered questions, their correct answers and explanations are always shown on the page
- Pure sever-side solution using conventional web application
  - For each question answer submission, the whole page with most of repeated data sent to the browser
- Pure client-side solution using conventional JavaScript
  - The user can read JavaScript source code to view what is correct answer
  - Large amount of explanation data will be carried by the JavaScript code
- AJAX solution
  - After the user answers a question, use XmlHttpRequest to ask the server to send the correct answer and explanation.
  - Display the correct answer and explanation received from the server

Will continue ajax again once PHP is over