MODULE – II

XML (Extensible Markup Language) is a markup language designed for storing and transporting data, making it both human-readable and machine-readable. It's used for defining and storing data in a shareable manner, facilitating information exchange between systems like websites, databases, and applications. Unlike HTML, XML doesn't have predefined tags; instead, you define your own tags to suit your specific needs.

Key Characteristics:

- **Markup Language:** XML is a set of rules for encoding documents, not a programming language.
- **Data Storage and Transport:** It's designed to store data in a structured format and facilitate its transmission between systems.
- **Self-Descriptive:** XML tags describe the data they contain, making it easy to understand and process.
- **Extensible:** You define your own tags, making it flexible for various data structures.
- **Software and Hardware Independent:** XML can be used on different platforms and with different software.

XML Importance

- **Data Interchange:** XML is a standard for exchanging data between different systems, enabling interoperability.
- **Configuration Files:** It's commonly used for storing configuration settings in applications.
- **Web Services:** XML is a key component in many web service technologies.
- **Data Storage:** XML can be used as a format for storing data in databases and files.

Comparing XML with other formats

- XML vs. HTML: While both are markup languages, HTML focuses on displaying data with a fixed set of tags, while XML is used to store and transport data, allowing for custom tags.
- XML vs. JSON: JSON (JavaScript Object Notation) is a lighter-weight format often preferred for web APIs due to its simplicity. XML, however, excels in situations requiring complex document structures, schema validation, namespaces, and richer data typing capabilities.

XML today:

Despite the rise of JSON, XML continues to be relevant and widely used, particularly in enterprise systems, financial institutions, document management, and areas where its strengths like schema validation, namespaces, and complex data type support are crucial.

Basic XML structure example

```
xml_introduction = """

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

<root>

  <element>Content</element>

</root>

"""
```

- The XML above is quite self-descriptive:
- But still, the XML above does not DO anything. XML is just information wrapped in tags.
- Someone must write a piece of software to send, receive, store, or display it

The Difference Between XML and HTML

1. XML and HTML were designed with different goals:
2. XML was designed to carry data - with focus on what data is
3. HTML was designed to display data - with focus on how data looks
4. XML tags are not predefined like HTML tags are
5. XML Does Not Use Predefined Tags

The XML language has no predefined tags.

The tags in the example above (like <to> and <from>) are not defined in any XML standard. These tags are "invented" by the author of the XML document.HTML works with predefined tags like <p>, <h1>, <table>, etc.With XML, the author must define both the tags and the document structure.

XML is Extensible:Most XML applications will work as expected even if new data is added (or removed).Imagine an application designed to display the original version of note.xml (<to> <from> <heading> <body>).

Then imagine a newer version of note.xml with added <date> and <hour> elements, and a removed <heading>.The way XML is constructed, older version of the application can still work

Defining XML tags, their attributes and values

XML (Extensible Markup Language) provides a flexible framework for structuring data, allowing users to define their own tags (elements), attributes, and values to represent information in a meaningful way.

1. XML tags (elements)

- Customization: Unlike HTML's predefined tags, XML allows you to create your own tags to describe your data precisely.
- Structure: Tags act as containers for data and define the structure of your XML document. They establish a hierarchy, with a single root tag enclosing all other elements.
- Syntax:
    - Start tags begin with < and end with >, for example: <book>.
    - End tags begin with </ and end with >, for example: </book>.
    - Empty elements can be represented with a self-closing tag, like <empty_element/>.
- Case Sensitivity: XML tags are case-sensitive. <book> is different from <Book>.
- Nesting: Elements must be properly nested. If an element starts inside another element, it must also end inside that element. For example, <parent><child></child></parent> is correct, while <parent><child></parent></child> is not.
- Root Element: Every XML document must have a single root element that encloses all other elements.

2. Attributes

Attributes are name-value pairs that provide additional information about an element and are placed within the element's start tag. They are written as attribute_name="attribute_value" using either single or double quotes for the value. An element cannot have duplicate attribute names.

3. Values

Values in XML can refer to attribute values or the text content within an element's tags. Attribute values are typically simple strings or numbers, while element content is the

data held between the start and end tags. Elements can also contain both text and other elements (mixed content).

Example

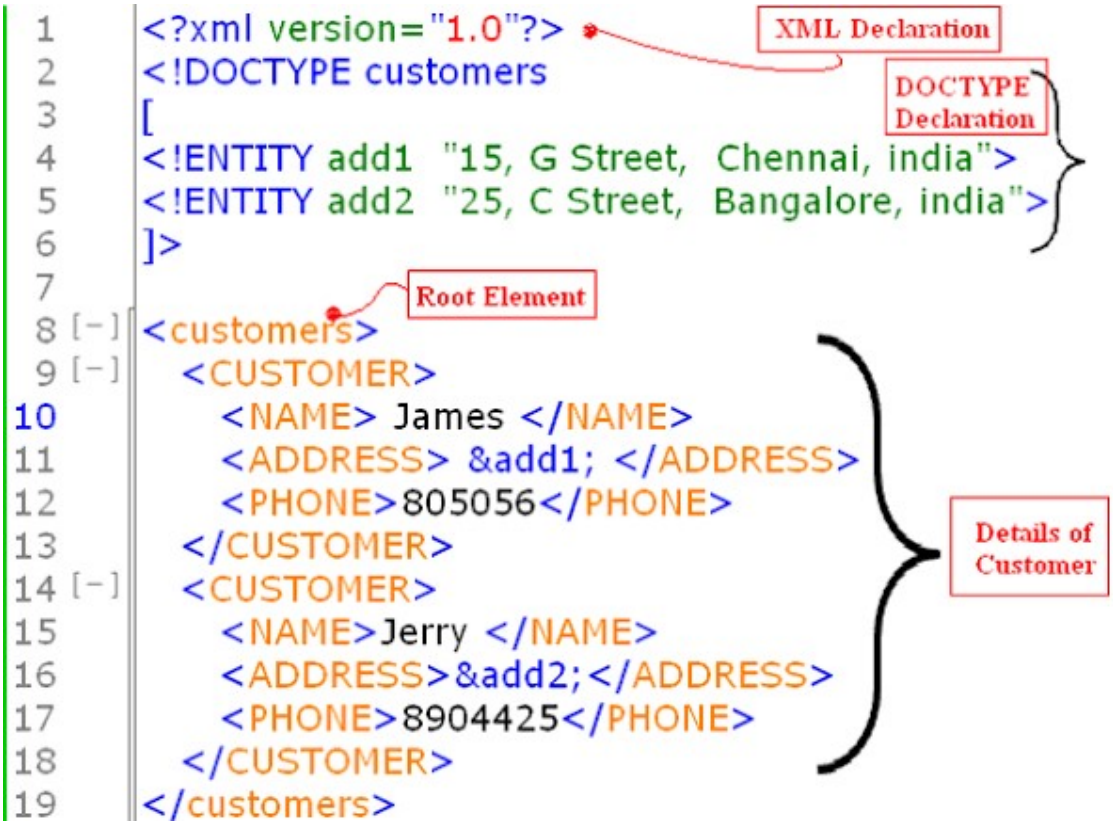
xml

```
<?xml version="1.0" encoding="UTF-8"?>
<library type="public">
 <book category="fiction" isbn="978-0140449179">
  <title>1984</title>
  <author>George Orwell</author>
  <year>1949</year>
 </book>
 <book category="science" isbn="978-0140449193">
  <title>Cosmos</title>
  <author>Carl Sagan</author>
  <year>1980</year>
 </book>
</library>
```

In this example, <library> is the root element with a type attribute. <book> elements have category and isbn attributes. Elements like <title>, <author>, and <year> contain text content as their values. This structure allows for flexible data representation using custom tags, attributes, and values.

Document Type Definition

A Document Type Definition (DTD) is a set of rules that defines the structure and the legal elements and attributes of an XML document. Think of it as a blueprint for an XML document.




```
1  <?xml version="1.0"?>
2  <!DOCTYPE customers
3  [
4  <!ENTITY add1  "15, G Street,  Chennai, india">
5  <!ENTITY add2  "25, C Street,  Bangalore, india">
6  ]>
7
8  <customers>
9    <CUSTOMER>
10     <NAME> James </NAME>
11     <ADDRESS> &add1; </ADDRESS>
12     <PHONE>805056</PHONE>
13   </CUSTOMER>
14   <CUSTOMER>
15     <NAME>Jerry </NAME>
16     <ADDRESS>&add2;</ADDRESS>
17     <PHONE>8904425</PHONE>
18   </CUSTOMER>
19 </customers>
```

Here's a breakdown of DTDs:

1. What it defines

- Elements: The tags you can use, like <note>, <to>, <from>, etc.
- Attributes: The properties of those elements, such as id or type.
- Relationships: How elements are nested and how they relate to each other, forming a tree structure.
- Data Types (limited): While XML Schema (XSD) offers robust data type definitions, DTDs primarily use #PCDATA (parsed character data) to signify text content.

2. Why use DTDs?

- Validation: DTDs enable XML parsers to validate whether an XML document conforms to the defined structure and rules. This is crucial for maintaining data integrity and consistency, especially when exchanging data between different systems or applications.
- Standardization: DTDs allow different groups or applications to agree on a common structure for data exchange, promoting interoperability.

- Documentation: DTDs act as a form of documentation, providing a clear understanding of the expected structure of an XML document.
- Entity Declarations: DTDs allow you to define reusable strings or special characters as entities, which can be referenced within the XML document.

3. How DTDs work

- DOCTYPE Declaration: An XML document associates itself with a DTD through a DOCTYPE declaration at the beginning of the document.
- Validation Process: A validating XML parser uses the DTD to check if the XML document adheres to the specified rules. If the document violates any of the rules, the parser reports an error, [Link: according to IBM https://www.ibm.com/docs/en/dmrt/9.5?topic=dtds-document-type-definitions-overview].

4. Types of DTD declarations

- Internal DTD: The DTD is declared within the XML file.

Example:

xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE note [
 <!ELEMENT note (to,from,heading,body)>
 <!ELEMENT to (#PCDATA)>
 <!ELEMENT from (#PCDATA)>
 <!ELEMENT heading (#PCDATA)>
 <!ELEMENT body (#PCDATA)>
]>
<note>
 <to>Tove</to>
 <from>Jani</from>
 <heading>Reminder</heading>
 <body>Don't forget me this weekend!</body>
</note>
```

```

- External DTD: The DTD is defined in a separate .dtd file and referenced from the XML document.

**Example XML with external DTD reference:**

xml

<?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>

```

Example note.dtd file:

xml

<!ELEMENT note (to,from,heading,body)>

<!ELEMENT to (#PCDATA)>

<!ELEMENT from (#PCDATA)>

<!ELEMENT heading (#PCDATA)>

<!ELEMENT body (#PCDATA)>

``` ## 5. DTD vs. XML Schema (XSD)

While DTDs are still in use, particularly for compatibility with older systems or simpler documents, XML Schema Definition (XSD) has largely replaced them. XSDs offer several advantages:

- Data Types: XSDs support various data types, such as integers and dates.
- Namespaces: XSDs allow namespaces, which help avoid naming conflicts.
- Complexity: XSDs can define more complex structures and validation rules.
- Extensibility: XSDs are more extensible and modular than DTDs.
```

DTDs provide a fundamental way to define the structure of XML documents and ensure their validity. However, for more complex applications requiring sophisticated data type handling, namespaces, and complex validation rules, XML Schema (XSD) is generally preferred.

XML Schemes

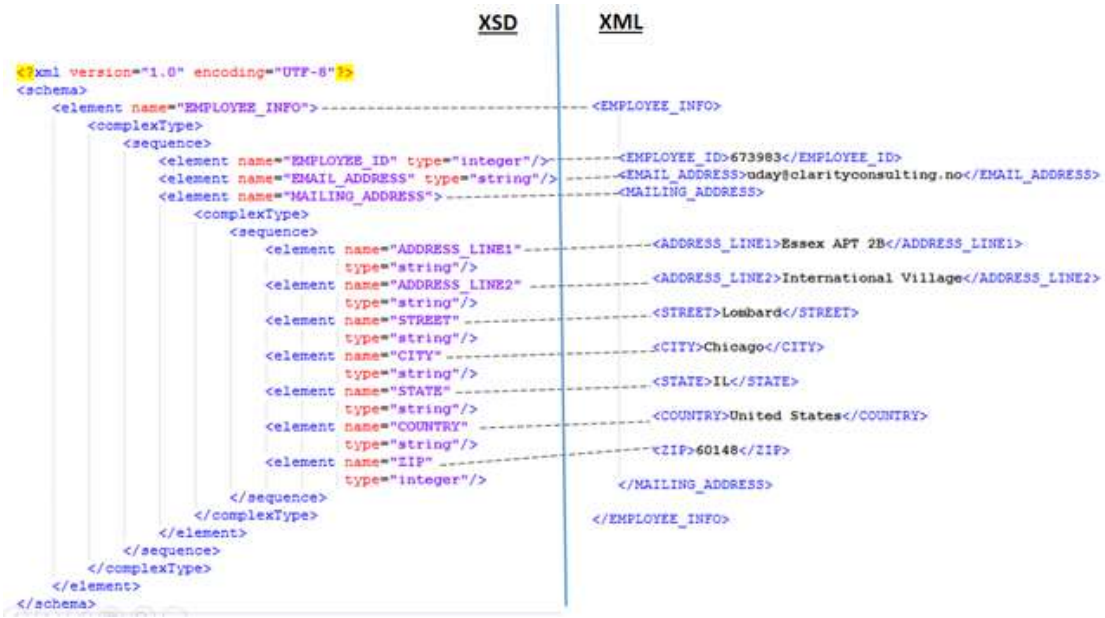
XML Schemas (XSD) are a powerful mechanism for defining the structure, content, and data types of XML documents. They act as a blueprint or rulebook, ensuring that XML data adheres to predefined formats and constraints, which is critical for data integrity and interoperability.

Key characteristics and benefits

- Structure Definition: XSDs precisely define the elements and attributes that can appear in an XML document, their order, and their nesting relationships.
- Data Type Support: XSD supports a rich set of built-in data types (like string, integer, date) and enables the creation of custom data types derived from these, allowing for precise control over data values. This is a significant advantage over DTDs, which lack data type support.
- Validation: XSDs enable validation of XML documents, ensuring that they conform to the defined schema. This helps to prevent errors and ensure data quality.
- Namespaces: XSDs support XML namespaces, which are crucial for preventing naming conflicts when combining XML documents or components from different sources.
- Extensibility: XSDs are extensible, allowing for the reuse of existing schema components and the derivation of new elements from existing ones.
- XML-based Syntax: XSDs are written in XML itself, making them easily processable by XML tools and parsers.

XML Schema vs. DTD

While DTDs (Document Type Definitions) were an earlier mechanism for defining XML document structure, XSDs offer several advantages that make them a preferred choice for many applications.



Feature	DTD	XSD
Syntax	Uses its own syntax, different from XML.	Uses XML syntax, allowing for the use of XML processing tools.
Data Types	Does not support data types.	Supports a rich set of data types and allows for custom data type creation.
Namespaces	Does not support namespaces.	Supports namespaces, enabling better organization and avoiding name conflicts.
Extensibility	Not extensible.	Extensible, supporting the reuse of schema components and the derivation of new elements.
Validation	Provides basic structural validation.	Offers a deeper level of validation, including patterns, lengths, and value constraints.

Uses and applications

- Data Exchange: XSDs facilitate reliable and consistent data exchange between various systems and applications by defining a common language for data organization and content verification.

- Web Services: XSDs are widely used in web services for defining the structure of messages exchanged between clients and servers.
- Configuration Files: Many software applications use XSDs to define the structure of their configuration files, ensuring proper format and data validity.
- Document Processing: XSDs aid in document processing by providing a blueprint for creating, storing, and exchanging structured documents.
- Database Integration: XSDs help in linking XML data with database layouts, easing data transfer between XML-based systems and databases.

XSD tools

Several tools are available for creating, editing, validating, and visualizing XML Schemas, making it easier to work with these complex definitions. These tools often offer features like:

- Graphical editors: Visual representations of the schema structure, allowing for drag-and-drop editing.
- Validation and error checking: Identifying and reporting errors in the schema or in XML documents validated against it.
- Code generation: Generating code in various programming languages from the XSD, enabling easier data binding.
- Documentation generation: Creating human-readable documentation from the XSD, making it easier for developers to understand the schema.

Document Object Model

The Document Object Model (DOM) is a programming interface for web documents, specifically HTML and XML. It provides a structured, tree-like representation of the document, allowing programs (primarily JavaScript) to access, modify, and interact with the content, structure, and style of a web page dynamically**.**

1. What the DOM represents

When a web page is loaded, the browser creates the DOM, which is an in-memory representation of the HTML document. It acts as a bridge, or interface, between the static HTML document and the dynamic capabilities of scripting languages like JavaScript.

The DOM represents a document as a logical tree where:

- The document itself is the root node.
- HTML elements (like <html>, <head>, <body>, , <p>, <img>) are represented as element nodes.
- Text content within elements are text nodes.

- Attributes of HTML elements are attribute nodes.
- Comments within the HTML are comment nodes.

2. Key functions of the DOM

The DOM empowers JavaScript to control and manipulate web pages in powerful ways, including:

- Accessing and selecting elements: Finding specific HTML elements within the document based on their ID, class name, tag name, or CSS selectors.
- Modifying content: Changing the text content of elements (using textContent or innerHTML), updating HTML structure within elements, or altering attribute values.
- Modifying style: Dynamically applying CSS styles to elements to change their appearance.
- Adding and removing elements: Inserting new HTML elements into the document or deleting existing ones.
- Handling events: Attaching event handlers to elements to respond to user interactions like clicks, form submissions, or key presses.

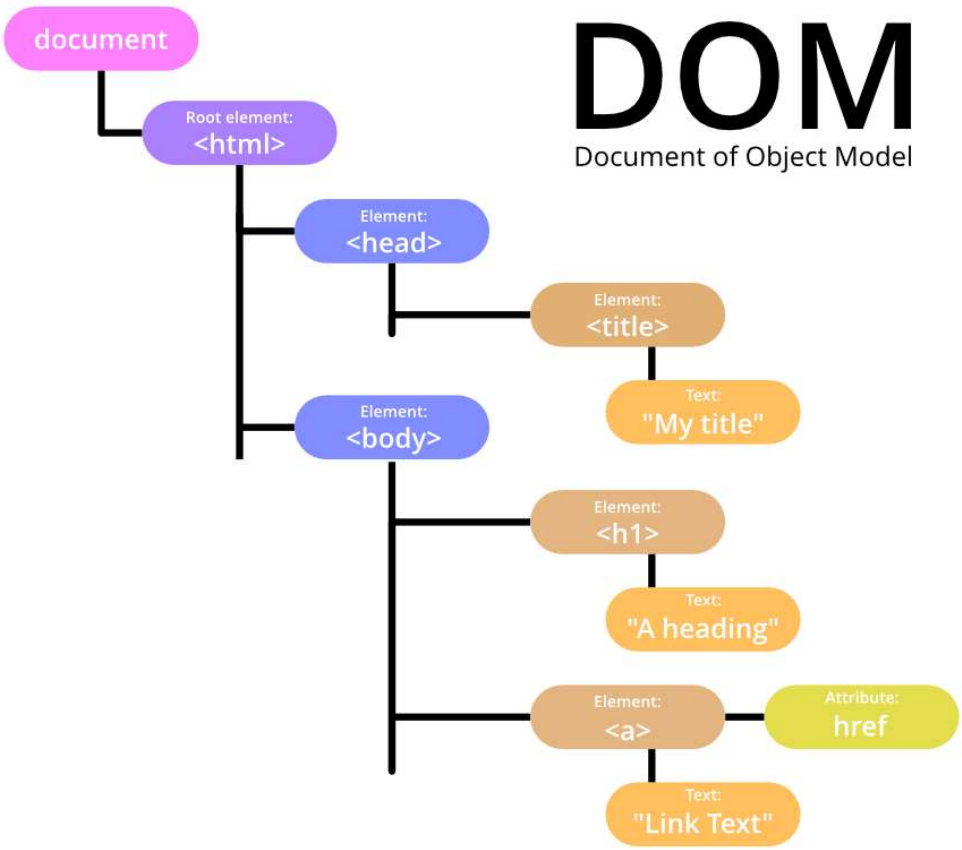
3. How the DOM works

1. Parsing: The browser receives and parses the HTML document.
2. DOM Tree Construction: Based on the parsed HTML, the browser constructs the DOM tree in memory, representing all elements, attributes, and text as nodes.
3. CSS Parsing: The browser also parses CSS stylesheets, creating a CSSOM (CSS Object Model).
4. Render Tree Creation: The DOM tree is combined with the CSSOM to form a render tree that contains visual information about each element.
5. Layout and Painting: The browser calculates the layout of elements and paints the page visually.
6. JavaScript Interaction: JavaScript can then interact with the DOM using its API to access and modify elements. Changes to the DOM are immediately reflected in the render tree, triggering the browser to re-render the page4. Importance of the DOM

- Dynamic Content: The DOM enables web pages to update content without needing full page reloads, like dynamic forms or data displays.
- Interactivity: It allows developers to build interactive elements such as animations, dynamic menus, form validations, and other features that respond to user actions.

- Foundation for Web Development: The DOM is the bedrock for many modern web development libraries and frameworks like React, Angular, and Vue.js.
- Event Handling: It provides the mechanism for capturing and responding to various user and system events on a web page.

In essence, the DOM is what makes web pages come alive and interactive. It provides a structured, programmatic way for scripts to interact with the web document, creating engaging and responsive web experiences.



Simple API for XML:

SAX (Simple API for XML) parsers are a type of XML parser that operate on an event-driven model. Unlike DOM (Document Object Model) parsers, which build an in-memory tree representation of the entire XML document, SAX parsers read through the XML document sequentially and trigger events as they encounter different parts of the document.

Key characteristics of SAX parsers:

- Event-driven:

SAX parsers trigger events
(e.g., startElement, endElement, characters, startDocument, endDocument) as they process the XML stream.

- Sequential processing:

The parser reads the XML document from beginning to end and does not allow random access to elements.

- Low memory consumption:

Because SAX parsers do not load the entire document into memory, they are well-suited for parsing large XML files.

- Read-only:

SAX parsers are primarily used for reading and extracting data from XML documents; they do not provide mechanisms for modifying the XML structure.

- Handler-based:

Users implement a handler class (often by extending DefaultHandler in Java) that defines the logic for processing the events triggered by the parser.

How SAX parsing works:

- A SAX parser instance is created, typically through a SAXParserFactory.
- A custom handler class is created, extending a default handler (e.g., DefaultHandler in Java) and overriding specific callback methods to handle desired events.
- The parse() method of the SAX parser is invoked, providing the XML input source and the custom handler.
- As the parser reads the XML, it triggers the corresponding callback methods in the handler (e.g., startElement() when an opening tag is encountered, characters() for text content, endElement() for a closing tag).
- The logic within these callback methods processes the extracted data or performs other actions based on the events.

CRUD operations :

CRUD operations (Create, Read, Update, Delete) can be performed on XML data using various programming languages and libraries. The core idea is to manipulate the XML structure to reflect the desired changes.

1. Create:

- Adding new elements/nodes:

New XML elements or attributes can be added to an existing XML document. This typically involves creating a new node object and appending it to a parent node within the document's structure.

- Example:

In Python, using xml.etree.ElementTree, you could create a new Element and use append() to add it to another element.

2. Read:

- Parsing and accessing data:

XML documents are parsed into a tree-like structure, allowing navigation and retrieval of data from specific elements or attributes. This often involves using XPath expressions or similar querying mechanisms to locate desired information.

- Example:

In C#, LINQ to XML allows querying and extracting data from XML documents using familiar LINQ syntax.

3. Update:

- Modifying existing data:

The content or attributes of existing XML elements can be changed. This involves locating the target element and then updating its text content or attribute values.

- Example:

In Java, using DOM parsers, you would get a reference to the element and then call methods like setTextContent() or setAttribute().

4. Delete:

- Removing elements/nodes:

Specific elements or attributes can be removed from the XML document. This typically involves locating the node to be deleted and then calling a removal method on its parent node.

- Example:

In PHP, using SimpleXML, you can use unset() on a SimpleXMLElement object to remove an element.

Common Approaches and Tools:

- DOM (Document Object Model):

Provides a tree-based representation of the XML document, allowing programmatic access and manipulation of nodes. Available in many languages (e.g., Java, JavaScript, Python).

- SAX (Simple API for XML):

An event-driven parser, more efficient for large XML files as it processes the document sequentially without loading the entire structure into memory. Primarily used for reading.

- LINQ to XML (C#):

A powerful and intuitive API for querying and manipulating XML data using Language Integrated Query (LINQ).

- XPath:

A language for navigating and selecting nodes in an XML document, often used in conjunction with other XML processing libraries.

- XML-RPC:

A protocol for remote procedure calls using XML to encode calls and responses, enabling CRUD operations on data stored remotely.

Angular

Pre-Requisites:

HTML 5

CSS 3

JavaScript (ES6)

TypeScript

Lab Setup:

NodeJS 20 or above

VSCode as IDE

NodeJS:

NodeJS is a runtime environment for java script.

It uses NPM - node package manager as a build tool

To Create a node application

md app-folder

cd app-folder

npm init -y

'npm init' will initialize 'package.josn' file which holds the project meta data and list of dependencies and list of applicateion life cycle scripts.

Install a dependency

npm i thrid-party-package-name

Install a dev-time dependency

npm i thrid-party-package-name --save-dev

Uninstall a dependency

npm uninstall thrid-party-package-name

Installing a dependency globally

npm i --global thrid-party-package-name

'node_modules' is the folder that holds the downloaded dependencies in our application.

npm-scripts

npm start is a customizable script to launch our application

npm test is a customizable script to invoke test cases of our application

npm build is a customizable script to invoke build of our appliation

npm run script-name will allow us to trigger scripts of our own

TypeScript

is a microsoft product and is a suepr set of javascript with typesafty.

typescript = javascript + typesafty

Data Types

number

string

boolean

bigint

void

null

undefined

any

unknown

User Defined Data Types

class

interface

enum

Angular JS :

Angular and AngularJS are both popular frameworks developed by Google, but they have key differences. AngularJS, released in 2010, is based on JavaScript and follows an MVC (Model-View-Controller) architecture. It is mainly used for building single-page applications (SPAs).

Angular, on the other hand, is a complete rewrite introduced in 2016 and is a typescript-based framework. By leveraging static typing, Angular provides type safety, improves developer productivity, and enhances code organization, offering enhanced performance and features like two-way data binding, modular architecture, and improved dependency injection.

While AngularJS is suitable for smaller projects, Angular is designed for modern web applications with advanced features, making it ideal for scalable and complex web applications with better tooling and development support.

The main difference between Angular and AngularJS is that Angular is a complete rewrite of AngularJS based on Typescript, offering improved performance and a component-based architecture, whereas AngularJS relies on plain JavaScript and a directive-based approach. The differences between Angular and AngularJS include their underlying architecture, performance capabilities, and modern tooling support. You can compare Angular and AngularJS to see how each framework has evolved and which is more suitable for current web development needs.

Expressions:

In AngularJS, expressions are JavaScript-like code snippets used to bind application data to HTML. They are primarily used to display data directly within the HTML view and can be written in two main ways: Double Curly Braces (Interpolation).

Expressions are enclosed within {{ }}. AngularJS evaluates the expression and inserts the resulting value directly into the HTML where the expression is placed.

From Angular 2 onwards we had interpolation , we can use to bind the data.

```
<p>Total: {{ quantity * price }}</p>
```

AngularJS HTML DOM:

AngularJS interacts with the HTML Document Object Model (DOM) primarily through its directives. The DOM represents the structure and content of an HTML document as a tree of objects, allowing JavaScript to access and manipulate them. AngularJS leverages this to dynamically update the user interface based on application data.

Key aspects of AngularJS HTML DOM interaction:

- **Directives:**

AngularJS directives are the core mechanism for manipulating the DOM. They extend HTML with custom attributes and elements, binding application data to HTML DOM element attributes and controlling their behavior.

  - ng-disabled: Binds application data to the disabled attribute of HTML elements, enabling or disabling them based on a condition.
  - ng-show and ng-hide: Control the visibility of HTML elements based on a boolean expression, showing or hiding them dynamically.
  - ng-click: Handles click events on HTML elements, triggering functions or expressions in the application's scope.
  - **Custom Directives**: Developers can create their own directives to encapsulate DOM manipulation logic and create reusable components.

Events:

In AngularJS, events are actions or occurrences that happen within the web application, typically in response to user interactions with the Document Object Model (DOM). These events allow the application to react dynamically to user input and provide a responsive user experience. AngularJS provides a set of directives that facilitate the binding of event listeners to HTML elements.

Common AngularJS Event Directives:

AngularJS offers directives for various types of events, including:

- Mouse Events:
  - ng-click: Executes when an element is clicked.
  - ng-dblclick: Executes on a double-click.
  - ng-mousedown: Executes when a mouse button is pressed down on an element.
  - ng-mouseup: Executes when a mouse button is released on an element.
  - ng-mouseenter: Executes when the mouse pointer enters an element.
  - ng-mouseleave: Executes when the mouse pointer leaves an element.
  - ng-mousemove: Executes when the mouse pointer is moved over an element.

- ng-mouseover: Executes when the mouse pointer is moved over an element or its children.

Angular Introduction

is a single-page-app framework.

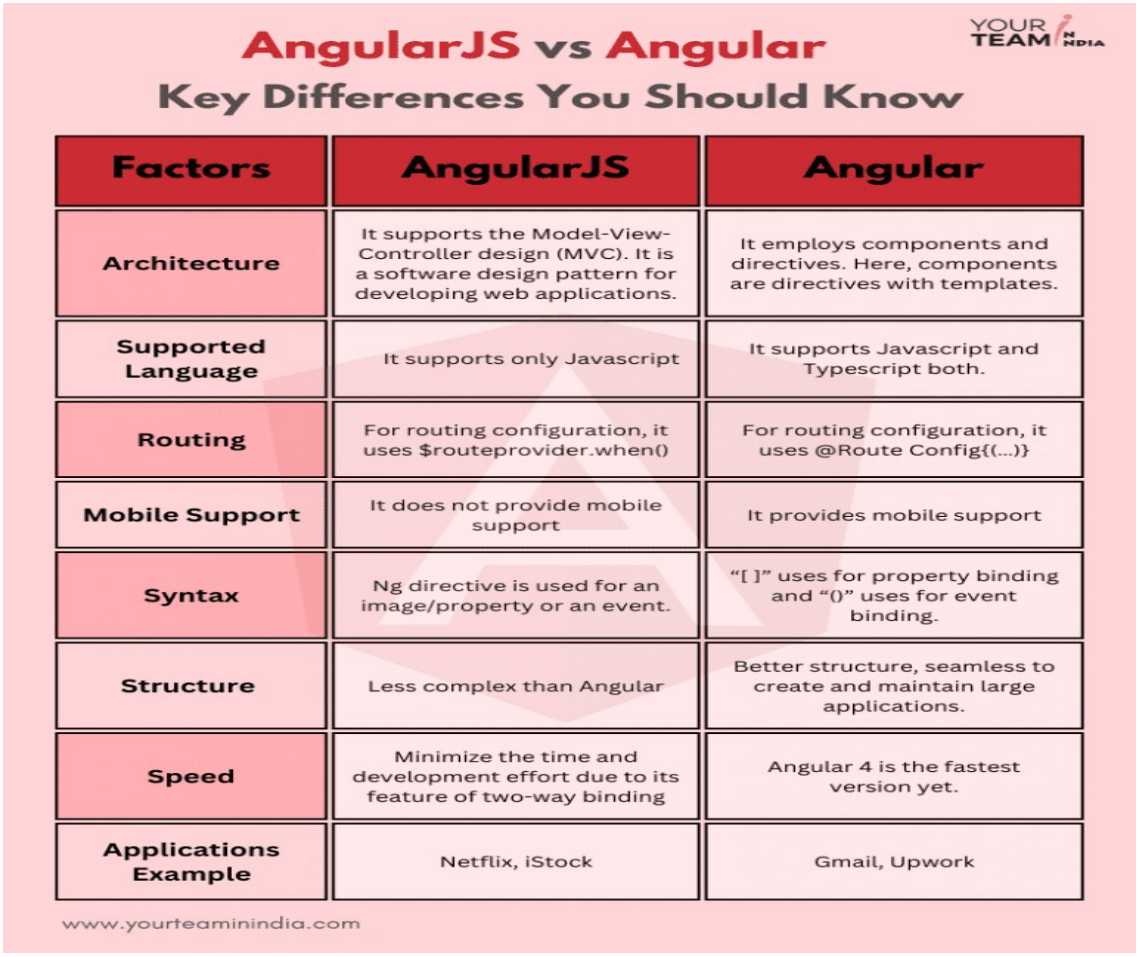
An app that works inside a browser and does not require page reload during use. Loads a single HTML page and dynamically update the page.

Need:

1. SPA's (Single page application)are fast as resources are loaded only once, when the data is being transmitted back and forth.

2. Simplified and streamlined development

3. Cache local storage effectively.



Its features are

- Cross-Platform: The angular cross-framework platform enables you to create stunning UIs for web and native mobile and desktop applications. Also, the framework is convenient for developing macOS, Windows, and Linux operating systems applications.

- Makes Use of TypeScript: Angular uses the programming language TypeScript, ensuring fewer errors during the compile time as the types of variables are defined beforehand. Also, any piece of JavaScript code is valid TypeScript code.
- Angular CLI (Command Line Interface): This feature lets you speed up the development process. From setting up the project to adding components, there are multiple tasks that you can carry out by simply using Angular's native CLI.
- Unit Testing Support: With Angular, it becomes feasible to execute unit testing hassle-free and, thus, ensure that your code has minimal bugs.
- Dependency Injection Feature: Angular's dependency injection feature allows efficient management of dependencies for components and services, enhancing modularity and code reusability by enabling services like data fetching and validation to be injected into multiple components.
- Data Binding Feature: Angular provides a powerful data binding feature that synchronizes the model and view, improving performance and responsiveness. It supports different types of data binding, such as one-way and two-way, to suit various application needs.
- One Way Data Binding: By default, Angular uses one way data binding, which optimizes performance by reducing unnecessary DOM updates and limiting data flow to a single direction, while still allowing two-way binding when needed via ngModel.

Angular Architecture:

An angular application is built with resources like

Component

Directive

Service

Pipe

Module

Each and every artifact is a typescript class

Each of these are marked with a decorator to indicate the role of the artifact.

The configuration of each of these artifacts is passed as a json--obj to the decorators and is called meta-data.

Example:

Component

```
@Component({
  selector:"",
  templateUrl:"",
  standalone:true
})
class DashboardComponent {
    //body containing function , events
}
```

Directive

```
@Directive({
  selector:"",
  standalone:true
})
class StockStatusDirective {
}
```

Service

```
@Injectable({
 providedIn:'root'
```

```
})
class StockService {
}
```

Pipe

```
@Pipe({
  name:"".
  standalone:true
})
class IntoWordsPipe {
}
```

Module

```
@NgModule({
  declarations:[],
  imports:[],
  exports:[],
  providers:[]
})
class SalesModule {}
```

Angular CLI

is a command-line-interface that works like a frontier of commands used to manage the life cycle of an angular application.

tools like angular-cli, testing tools (karma and jasmine), minification and build tools ..etc.,

are executed on nodejs and after building the app, the app runs on a browser.

modern angular also offers SSG and SSR to reduce intial loading time, and the SSR is executed

as well on nodejs.

ng new app-name => to create a new angular app

ng add feature-name => to add a new module or a feature

ng build => to compile ts into js and builds the app into 'dist' folder

ng serve => to compile ts into js and luanch the app on a dev-server at 4200

ng serve --port 9999 => to compile ts into js and luanch the app on a dev-server at 9999

ng serve --port 9999 -o => to compile ts into js and luanch the app on a dev-server at 9999, opens the browser

ng test => to invoke test cases

ng g c ComponentName => generate a new stand-alone component

--skip-tests will avoid generateing test cases

--no-standalone to create a component inside a module

--module will carry the module name

ng g directive DirectiveName

generate a new stand-alone directive

--skip-tests will avoid generateing test cases

--no-standalone to create a component inside a module

--module will carry the module name

ng g pipe PipeName

generate a new stand-alone pipe

--skip-tests will avoid generateing test cases

--no-standalone to create a component inside a module

--module will carry the module name

ng g service ServiceName

generate a new service

--skip-tests will avoid generateing test cases

ng g module ModuleName

generate a new module

Angular Components

a component in angular is a angular built html-element.

each component is made up of three parts

the component-class dashboard.component.ts holds the state and behaviour of the component

the template dashboard.component.html holds the html-dom to be rendered for this component

the styleSheet dashboard.component.css holds the style local to this component

dashboard.component.ts

```
@Component({
  selector:"app-dashboard",
  templateUrl:"dashboard.component.html",
  styleSheets:["dashboard.component.css"],
  standalone:true
})
export class DashboardComponent{
  //state as fields and behaviour as methods
  String userName;
}
```

```
<app-dashboard></app-dashboard>
```

Data Binding

is to access the fields and methods of a component-class in the component-template.

Interpolation or Expressions

is to render the value of an angular-expression in the content of an html-element.

<tag-name> {{angular-expression}} </tag-name>

 The current user is {{userName}}

Two-Way Data Binding

is to bind the value of a field to an input-element and vice-versa.

'ngModel' is a built-in directive from 'FormModule' that is used to execute two-way dta binding.

<input [(ngModel)]="field" />

One-Way Data Binding

is to bind a field or method on to non-editable attributes of dom.

Attribute Binding

is to bind a field with a attribute of an element.

<tagName [attribute]="angularExpression"> content </tagName>

 This is a para <!- this is not binding -->

<p [title]="paraTitle"> This is a para <!- this is binding the value of 'paratitle' -->

Event Bidning

is to bind a method to an event directives

Event-Driectives are built-in Angular defiend attributes to handle events.

- click
- dblClick
- focus
- change
- blur
- ngSubmit
- mouseover

- mouseup
- mousedown

<tagName (eventDirective)="method()"> content </tagName>

<button type="button" (click)="doSomething()"> clicke me </button>

Style Binding

is to bind a field with a css-property or 'ngStyle' directive.

<tagNAme [style.cssProperty]="angularExpression"> content </tagName>

<p [style.textAlign]="myTextalignField"> content </p>

<tagNAme [ngStyle]="aJsonObject"> content </tagName>

```
@Component({ .. })
class MyComponent {
  myParaStyle:any;
  construcotr(){
    this.myParaStyle = {border:"1px solid black",textAlign:"right"};
  }
}
```

<p [ngStyle]="myParaStyle"> content </p>

Class Binding

is to bind a field to eh 'class' attribute of an element.

this allows the dev to add or remove css-class dynamically.

```
<tagNAme [class.className]="boolenaAngularExpression"> content
</tagName>


@Component({ .. })
class MyComponent {
  isImportant:boolean
  construcotr(){
    this.isImportant = true;
  }
}

<p [class.important]="isImportant"> This is a para </p>
<tagNAme [ngClass]="anArrayOfClassesOraJsonObj"> content </tagName>

@Component({ .. })
class MyComponent {
  myParaClasses:string[];

  construcotr(){
    this.myParaClasses=["importnat","highlight"];
  }
}

<p [ngClass]="myParaClasses"> This is a para </p>

@Component({ .. })
class MyComponent {
  myParaClasses:any;
```

```
  construcotr(){
    this.myParaClasses={importnat:true,highlight:false};
  }
}

<p [ngClass]="myParaClasses"> This is a para </p>
```

Integrating Bootstrap

bootstrap is a css-js library that offers responsive web design.

bootstrap-icons is a css library that offers icons.

npm i bootstrap bootstrap-icons

these are installed in the node_modules folder.

the .css files of this library msut be added to the 'styles' section of angular.json file

the .js files of this library msut be added to the 'scripts' section of angular.json file

Angular Routing

Routing is to map a component to a url, and render the mapped component only when its url is requested.

Angular provides RouterModule for this priupose.

RouterModule

```
Route       object    {
                path:'urlToBeMapped',
                pathMatch:'startsWith|full'
                component:Component,
                redirectTo:''
                children:][],
```

```
                loadChildren :  lazyLoadingFunction,
                canActive: routerGuardArray,
                canLoad: routerGuardArray,
                canDeactive: routerGuardArray,
              }
```

Routes Route[]

Router buitl-in service used to navigate progrmatically

navigate("url");

navigateTo(["segment1","segment2"]);

ActivatedRoute built-in service used to read url-paramter, or url related data like path, querystring ..etc.,

RouterLink built-in directive to be used on 'a' element instead of its href

RouterLinkActive built-in directive to be used on 'a' element to apply a css-class only when a link is visited

RouterOutlet built-in component that reserve place on the layout, to be replaced by the mapped component of the current url.

Angular Flow Controls

Legacy Directives from CommonsModule

NgIf

NgFor

NgSwitch NgSwtichCase NgDefault

Modern Flow Controls

are built-in angular native controls that need to additonal improts to use

```
@if(cond) {
  //html dom
} @else {
  //alternate html dom
}
@switch(exp){
  @case (case1) {
    //html dom if exp===case1
  }
  @case (case2) {
    //html dom if exp===case2
  }
  @default {
    //html dom for default senario
  }
}

@for(loopingVar of array; track $index){
  //html dom we wnat to repeat one for each value in the array
}@empty{
  //html dome that shall render incase the array is empty
}
```

variables injectable by for

$index the index of the current element

$even is the current element index is even

$odd is the current element index is odd

$first is the current element index is the first

$last is the current element index is the last

$count the number of elements that are iterated over .

Inter Component Communication via @Input decorator

When a parent component has to share some object with a child component, it does it

through attributes, An attribute of a component is a field of the component class marked with

@Input decorator.

navbar.component.ts

```
@Component({
  selector:"nav-bar",
  ....
})
class NavBar {
  @Input()
  title!:string;
}
```

app.component.html

```
<nav-bar title="title can be passed here"></nav-bar>
<nav-bar [title]="aVariableFromParentComponent"></nav-bar>
```

Angular LifeCycle Hooks

a lifecyle hook is a method that get invoked automatically at a spacific stage of

a component or directives's lifecycle.

```
constructor()
  ↓
  ngOnChanges() from OnChanges      /* is to detect any chagnes that may occur on @input */
    ↓
    ngOnInit() from OnInit        /* is used to excute a task after the component is
      |                    loaded  initially */
      ↓
      ngOnChanges() from OnChanges   /* invokes everytime when a change occur on @input */
        ↓
        ngDoCheck()            /* is sued to detect any chagnews that angular couldn't */
          ↓
          ngAfterContentInit()
            ↓
            ngAfterContentChecked()
              ↓
              ngAfterViewInit()
                ↓
                ngAfterViewChecked()
                  |
                  ..... /*once the component is closed or removred */
                  ↓
                  ngOnDestroy()


@Component({
  selector:"dashboard",.....
})
class Dashboard {
```

```
  /*....*/
}
```

dashboard template

```
  <section>
    <h3>Some heading</h3>
  </section>
```

app component template

```
  <dashboard>
    <nav>
    </nav>
  </dashboard>
```

View is any dom declared in the template of the component

the section and the h3 are said to be the view

we can access these in the dashboard component class using

@ViewChild decorator

Content is any dom passed to the body of a component

the nav is called the content.

we can access these in the dashboard component class using

@ContentChild decorator

Angular Directives

A directive is any angular defined element or attribute.

Types Of Directives

(a) Component Directives are otherwise called Components - angular defiend elements

(b) Structural Directives are used to control the appearence of an element

NgIf, NgFor, NgSwitch

(c) Attribute Directives are angular defiend attributes

builtin attribute directives like NgModel, NgStyle, NgClass ...etc.,

we can create a custom attribute directive as well

ng g directive DirectiveName --skip-tests

```
@Directive({
  selector:"[attribute-name]"
})
class DirectiveName {
}
```

Angular Forms

Angular supports two types of forms:

- Template Driven Forms

are constructred in html and are bound to fields using ngModel directiove from FormsModule.

FormsModule

ngForm

ngModel

And a few validation related properties These forms cannot accommodate complicated object structures like arrays or nested objects.

These forms are not easy to tests as well.

These forms are recommended to handle a case that has not more than two fields.

- **Reactive Forms (Model Driven Forms)**

ReactiveFormsModule

FormControl

FormGroup

FormBuilder

These forms are built for any complicated object structure.

These forms are built on component class and are bound to the html dom.
These forms are built on component class and are bound

to the html dom.

Testing these form easy. 99% we use these forms in angular. Validation

related properties

FormGroup valid,invalid

FormControl valid,invalid,touched,untouched,pristine,dirty

MODULE - III

Introduction to servlets:

Servlets are Java programs that enhance server functionalities by processing client requests and generating dynamic responses, primarily in web applications. They are a fundamental component of Java web development and operate within the server's Java Virtual Machine (JVM). This makes them platform-independent and highly scalable. Servlets essentially act as a middle layer between the web browser and the server, handling the communication and processing the business logic behind web applications.

Key functions of servlets:

- Handling Client Requests: Servlets receive requests from clients (typically web browsers) and process the data sent with the request, such as information submitted through HTML forms.
- Generating Dynamic Content: Based on client input or backend logic, servlets dynamically generate and format responses, often as HTML, XML, or other data formats, and send them back to the client.
- Server-side Logic: Servlets are well-suited for implementing business logic, connecting with databases using JDBC, performing authentication and authorization, and managing sessions.
- Middleware in MVC: In the Model-View-Controller (MVC) architecture, servlets typically act as controllers, handling incoming requests and managing the communication between the view (JSP) and the model (Java classes/databases).

Advantages of servlets:

Servlets offer several advantages, including portability due to being written in Java. They are efficient because they run within the server's memory and use a multithreaded model for handling requests, which is more resource-friendly than creating new processes for each request. Java's features contribute to servlet robustness, and their thread-based processing allows for efficient handling of multiple concurrent requests, making them scalable. Servlets also integrate well with other Java technologies like JDBC and JavaBeans.

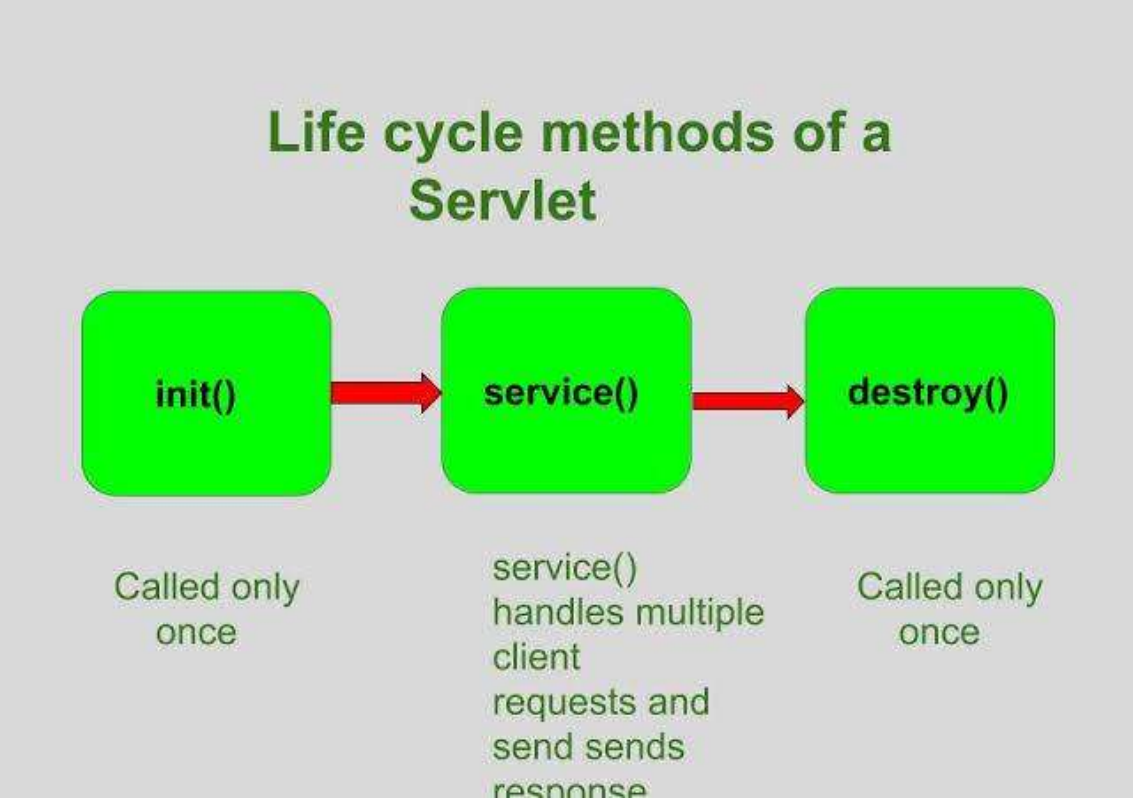
Servlet lifecycle:

A servlet's lifecycle is controlled by a servlet container. This lifecycle involves initialization when loaded (init() method), handling requests through the service() method (which dispatches to methods like doGet() or doPost()), and destruction (destroy() method) when no longer needed or when the server shuts down.

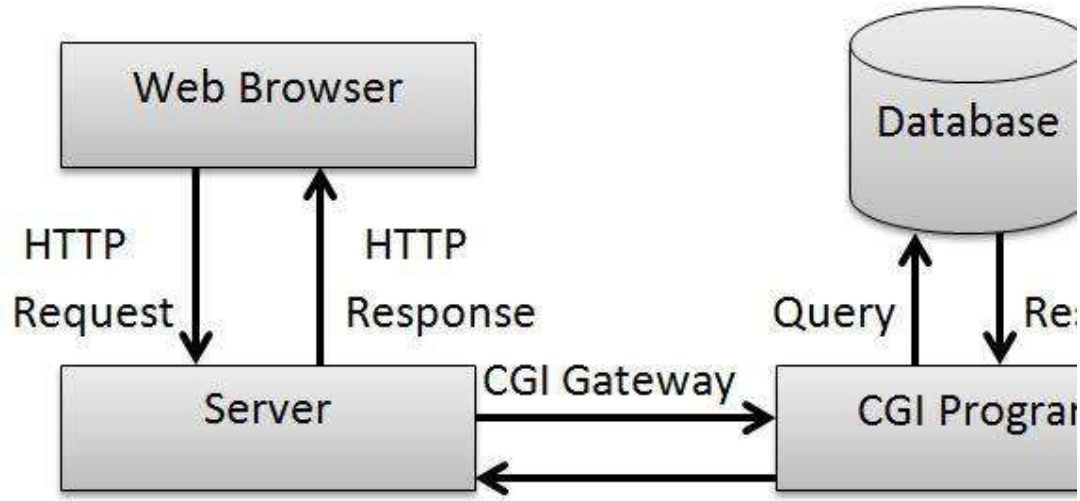
In essence Servlets are a fundamental Java technology for building dynamic web applications. They facilitate server-side processing, manage client-server communication, and enable the creation of robust and scalable web services.

The life cycle of a servlet is managed by the servlet container (e.g., Apache Tomcat) and consists of distinct stages:

- **Loading and Instantiation:**
    - When the web server starts or the first request for a servlet is received, the servlet container loads the servlet class.
    - Subsequently, a single instance of the servlet class is created. This instance is typically a singleton, handling all subsequent requests.
- **Initialization (init() method):**
    - After instantiation, the container calls the init(ServletConfig config) method on the servlet instance.
    - This method is called only once throughout the servlet's life cycle and is used for one-time initialization tasks, such as establishing database connections or loading configuration parameters.



Common Gateway Interface (CGI):



The Common Gateway Interface (CGI) is a standard protocol that enables web servers to execute external programs or scripts to generate dynamic content.

How CGI works

1. Client Request: A user's browser makes a request for a web page, typically by clicking a link or submitting a form.

2. Server Processing: The web server receives the request and identifies it as a CGI request.
3. Execution: The server launches the associated CGI script in a separate process.
4. Data Handling: The CGI script processes the request, potentially interacting with databases or other applications to retrieve or manipulate data. It receives information via environment variables for GET requests or standard input for POST requests.
5. Response Generation: The script generates dynamic content, typically in the form of HTML, based on the processed data.
6. Server Response: The server receives the generated content from the script and sends it back to the client's browser as its response.

Key features and characteristics

- Language Flexibility: CGI scripts can be written in various languages, including Perl, Python, C, and more.
- Dynamic Content: It enables the creation of web pages that change based on user input or other variables, facilitating interactive web applications.
- Form Handling: Commonly used for processing HTML forms submitted by users and generating customized responses.
- Middleware Role: It acts as middleware, facilitating communication between web servers and external databases or information sources.

Advantages of CGI

- Ease of Implementation: Relatively simple to set up for basic tasks.
- Reusability: Allows leveraging existing code or scripts written in various languages.
- Wide Compatibility: The CGI standard is widely supported across different systems and platforms.
- Reliability: Can be dependable for small, low-traffic websites or applications.

Disadvantages of CGI

- Performance Overhead: Launching a new process for every request can lead to slower response times, especially for high-traffic websites.
- Limited Caching: Difficult to cache data in memory between page loads, potentially impacting performance.

- Security Concerns: Requires careful coding practices and server configurations to avoid vulnerabilities like command injection or unauthorized access.
- Scalability Issues: Not ideal for handling large volumes of concurrent requests due to process creation overhead.

Modern alternatives

While CGI was crucial in the early days of dynamic web development, more efficient alternatives have emerged:

- Server-Side Scripting: Technologies like PHP, ASP.NET, and Ruby on Rails offer frameworks for handling web requests with better performance.
- Web Frameworks: Frameworks like Node.js and Django provide streamlined development, improved performance, and enhanced security.
- Java Servlets: Java-based alternatives that leverage threads for better scalability and efficiency.
- FastCGI, SCGI, AJP: Allow long-running application processes outside the web server, reducing process creation overhead.

Despite the emergence of newer technologies, understanding CGI remains valuable for comprehending the historical evolution of web development and the underlying principles that continue to influence modern practices.

Deploying Servlet

Deploying a servlet involves making it accessible and runnable within a web server or application server environment, typically a Servlet container like Apache Tomcat. The general process is as follows:

- **Develop the Servlet:**

Create your Java servlet class, extending HttpServlet and overriding methods like doGet() or doPost() to handle HTTP requests.

- **Compile the Servlet:**

Compile your Java servlet source code into a .class file. Ensure the servlet API JAR (e.g., servlet-api.jar or jakarta.servlet-api.jar) is included in your classpath during compilation.

- **Create Web Application Structure:**

Organize your web application files into a standard directory structure. This typically includes:

- **Root Directory:** For static content (HTML, JSP, CSS, JavaScript, images).
- **WEB-INF Directory:** A secure directory not directly accessible by clients.
    - **WEB-INF/classes:** Contains compiled servlet .class files and other Java classes, preserving package structure.
    - **WEB-INF/lib:** Contains any external JAR libraries required by your servlet.
    - **WEB-INF/web.xml:** The deployment descriptor, crucial for configuring your servlet.

- Configure web.xml (Deployment Descriptor):
    - Define your servlet using the <servlet> tag, specifying a unique name and the fully qualified class name.
    - Map a URL pattern to your servlet using the <servlet-mapping> tag, associating the servlet name with a specific URL pattern that clients will use to access it.
    - Alternatively, for Servlet 3.0+ containers, you can use the @WebServlet annotation directly on your servlet class to achieve the same mapping without web.xml entries.
- **Package the Web Application (Optional but Recommended):**

Create a Web Application Archive (WAR) file. This is a standard way to package web applications for easy deployment. Use build tools like Maven or Gradle, or manually create a ZIP file with a .war extension containing your web application structure.

- **Deploy to Servlet Container:**
    - **Manual Deployment:** Copy the web application directory or the WAR file to the appropriate deployment directory of your Servlet container (e.g., webapps directory in Apache Tomcat).
    - **IDE Deployment:** Most Integrated Development Environments (IDEs) provide built-in features to deploy web applications directly to a configured server.
- **Start the Server:**

Ensure your Servlet container (e.g., Tomcat) is running.

- **Test the Servlet:**

Access your servlet through a web browser using the configured URL pattern (e.g., http://localhost:8080/your_app_name/your_servlet_url).

The Servlet API:

The Servlet API provides the interfaces and classes for developing Java servlets, which are Java programs that extend the capabilities of a server. Servlets are primarily used to handle requests and generate dynamic responses in web applications. Key interfaces and classes within the API include:

- Servlet:

The core interface that all servlets must implement, defining lifecycle methods like init(), service(), and destroy().

- GenericServlet:

A convenience class that implements the Servlet and ServletConfig interfaces, providing a generic, protocol-independent servlet.

- HttpServlet:

An abstract class extending GenericServlet that provides methods specifically for handling HTTP requests (e.g., doGet(), doPost()).

- ServletConfig:

An object created by the container to pass initialization information to a servlet during its instantiation.

- ServletContext:

An object that defines a set of methods a servlet uses to communicate with its container, providing access to web application-wide information.

- ServletRequest and ServletResponse:

Interfaces representing the client request and the server's response, respectively.

Reading Servlet Parameters (Request Parameters)

Servlet parameters, also known as request parameters, are data sent from a client to a servlet, typically as part of an HTTP GET or POST request (e.g., form data, query string parameters). These parameters are accessed via the ServletRequest object.

- String getParameter(String name): Retrieves the value of a single parameter as a String. If the parameter does not exist, it returns null.

- String[] getParameterValues(String name): Retrieves an array of String values for a parameter that may have multiple values (e.g., a multi-select list or multiple checkboxes with the same name).
- Enumeration<String> getParameterNames(): Returns an Enumeration of all parameter names in the current request.

Servlet parameters are key-value pairs that a client sends with an HTTP request, typically from an HTML form or a URL's query string. They are temporary and vary with each request. You can access these parameters using methods from the HttpServletRequest object, which is passed to a servlet's doGet() or doPost() method.

Key methods

- request.getParameter("paramName"): Retrieves the value of a specific parameter as a String. Returns null if the parameter does not exist.
- request.getParameterValues("paramName"): Returns an array of String objects containing all values for a given parameter name. Use this for checkboxes or other inputs that can have multiple values.
- request.getParameterNames(): Returns an Enumeration<String> of all parameter names in the current request. This is useful for iterating through all available parameters.
- request.getParameterMap(): Returns a Map<String, String[]> of all parameters in the request, with parameter names as keys and an array of their values.

  Example:

  ```
  import java.io.IOException;

  import java.io.PrintWriter;

  import javax.servlet.ServletException;

  import javax.servlet.http.HttpServlet;

  import javax.servlet.http.HttpServletRequest;

  import javax.servlet.http.HttpServletResponse;


  public class FormDataServlet extends HttpServlet {

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
  ```

```
        throws ServletException, IOException {

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    // Read a single-value parameter
    String username = request.getParameter("username");
    out.println("<h3>Username: " + username + "</h3>");

    // Read a multi-value parameter (e.g., from checkboxes)
    String[] languages = request.getParameterValues("language");
    if (languages != null) {
      out.println("<h3>Favorite Languages:</h3>");
      out.println("<ul>");
      for (String lang : languages) {
        out.println("<li>" + lang + "</li>");
      }
      out.println("</ul>");
    }
  }
```

Reading initialization parameters :

Initialization parameters, or "init parameters," are configuration settings defined for a specific servlet or the entire web application. They are set when the servlet is initialized and remain constant for its entire lifecycle.

- Servlet-specific parameters: Defined in the web.xml deployment descriptor within the <servlet> tag or by using the @WebInitParam annotation. They are accessed via the ServletConfig object, which is passed to the init() method.
- Context-wide parameters: Global to the entire web application and accessible by all servlets. They are defined within the <web-app> tag in web.xml. They are accessed via the ServletContext object.

Key methods

- config.getInitParameter("paramName"): Retrieves a servlet-specific init parameter value via the ServletConfig object.
- config.getInitParameterNames(): Returns an Enumeration<String> of all init parameter names for that specific servlet.
- context.getInitParameter("paramName"): Retrieves a context-wide init parameter value via the ServletContext object.

Example with web.xml:

First, define the parameters in web.xml.

xml

```
<web-app>
  <servlet>
    <servlet-name>ConfigServlet</servlet-name>
    <servlet-class>com.example.ConfigServlet</servlet-class>
    <init-param>
      <param-name>emailSupport</param-name>
      <param-value>support@example.com</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>ConfigServlet</servlet-name>
    <url-pattern>/config</url-pattern>
  </servlet-mapping>

  <context-param>
    <param-name>appName</param-name>
    <param-value>MyWebApp</param-value>
  </context-param>
</web-app>
```

Then, access the parameters in your servlet code.

Java

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ConfigServlet extends HttpServlet {
   private String supportEmail;
   private String applicationName;

   @Override
   public void init(ServletConfig config) throws ServletException {
      super.init(config);

      // Read servlet-specific init parameter
      this.supportEmail = config.getInitParameter("emailSupport");

      // Read context-wide init parameter
      this.applicationName = config.getServletContext().getInitParameter("appName");
   }

   protected void doGet(HttpServletRequest request, HttpServletResponse response)
```

```
        throws ServletException, IOException {


    response.setContentType("text/html");
    PrintWriter out = response.getWriter();


    out.println("<h3>Application Name: " + this.applicationName + "</h3>");
    out.println("<h3>Support Email: " + this.supportEmail + "</h3>");
  }
}
```

Handling Http Request & Responses:

In servlets, handling HTTP requests and responses is done through the

HttpServletRequest and HttpServletResponse objects, which are passed as arguments to a servlet's doGet(), doPost(), and other do methods. The HttpServletRequest object provides access to the client's request details, while the HttpServletResponse object is used to construct and send the server's response back to the client.

Handling HTTP requests

Key HttpServletRequest methods

- Reading request parameters: Retrieve data sent from an HTML form or a URL's query string.
    - request.getParameter("paramName"): Gets the value of a parameter as a String.
    - request.getParameterValues("paramName"): Returns a String array for parameters with multiple values (e.g., from checkboxes).
- Accessing request headers: Retrieve metadata sent by the client, such as the user-agent or content type.
    - request.getHeader("headerName"): Gets the value of a specific header.
    - request.getHeaderNames(): Returns an enumeration of all header names.
- Handling the request body: For POST requests, you can read the body directly as a stream.

    - request.getReader(): Gets a BufferedReader for character data.
    - request.getInputStream(): Gets a ServletInputStream for binary data.
    - Note: You can use either getReader() or getInputStream() on a request, but not both. Calling getParameter() on a POST request also consumes the request body, so it should not be combined with stream-reading methods.
- Getting request metadata:
    - request.getMethod(): Returns the HTTP method (e.g., "GET" or "POST").
    - request.getRequestURI(): Gets the URI of the requested resource.
    - request.getCookies(): Returns an array of Cookie objects sent with the request.

Handling HTTP responses

Key HttpServletResponse methods

- Writing the response body:
    - response.setContentType("text/html"): Sets the MIME type of the content, which should be called before writing the response.
    - response.getWriter(): Returns a PrintWriter to send character-based text (like HTML) to the client.
    - response.getOutputStream(): Returns a ServletOutputStream for sending binary data.
    - Note: You can only use either getWriter() or getOutputStream() for a response, not both.
- Setting response headers and status codes:
    - response.setStatus(int statusCode): Sets the HTTP status code, such as 200 for OK or 404 for Not Found.
    - response.addHeader(String name, String value): Adds a header to the response.
- Redirecting the client:
    - response.sendRedirect(String location): Sends a temporary redirect response, telling the client's browser to navigate to a new URL.
- Managing cookies:

    - response.addCookie(Cookie cookie): Sends a Cookie from the server to the client, which the client's browser may store.
- Sending error messages:
    - response.sendError(int sc, String msg): Sends an error response with a specified status code and message.

Example: A servlet handling GET and POST requests

This example demonstrates a servlet that handles both GET and POST requests, retrieving form data and sending a response.

The HTML form (index.html)

html

```
<!DOCTYPE html>
<html>
<head>
  <title>Form Example</title>
</head>
<body>
  <h2>GET Request Form</h2>
  <form action="MyServlet" method="GET">
    First Name: <input type="text" name="firstName"><br>
    <input type="submit" value="Submit">
  </form>

  <h2>POST Request Form</h2>
  <form action="MyServlet" method="POST">
    Last Name: <input type="text" name="lastName"><br>
    <input type="submit" value="Submit">
  </form>
</body>
</html>
```

---------- ----------

The servlet (MyServlet.java)

java

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class MyServlet extends HttpServlet {
   protected void doGet(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {

      // Retrieve parameter from the GET request
      String firstName = request.getParameter("firstName");

      // Set response content type
      response.setContentType("text/html");

      // Get a PrintWriter to write the response
      PrintWriter out = response.getWriter();

      // Write the HTML response
      out.println("<html><body>");
      out.println("<h2>Hello, " + (firstName != null ? firstName : "Guest") + "!</h2>");
      out.println("<p>This is a response to your GET request.</p>");
      out.println("</body></html>");
```

```
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        // Retrieve parameter from the POST request
        String lastName = request.getParameter("lastName");

        // Set response content type
        response.setContentType("text/html");

        // Get a PrintWriter to write the response
        PrintWriter out = response.getWriter();

        // Write the HTML response
        out.println("<html><body>");
        out.println("<h2>Thank you, " + (lastName != null ? lastName : "User") + ".</h2>");
        out.println("<p>This is a response to your POST request.</p>");
        out.println("</body></html>");
    }
}
```

Connecting to a database using JDBC:

Connecting to a database in a servlet involves using the Java Database Connectivity (JDBC) API to establish a connection, execute SQL queries, and process the results. The most reliable method for a web application is to use a connection pool provided by the server, rather than opening and closing a new connection for every request.

Prerequisites

Before you start, you'll need the following:

- A JDBC driver JAR file for your specific database (e.g., MySQL, PostgreSQL, or Oracle).
- The driver file added to your project's classpath. If you're using a modern web container like Tomcat, you can simply place the JAR file in the WEB-INF/lib directory.

For optimal performance and resource management, especially in web applications, you should use a DataSource for connection pooling. This is configured in your application server (e.g., Tomcat) and looked up in your servlet using JNDI (Java Naming and Directory Interface).

1. Configure the DataSource

In your web server's configuration (e.g., context.xml for Tomcat), define the database connection details in a Resource element.

Example context.xml for MySQL:

xml

```
<Context>
  <Resource name="jdbc/YourDB"
    auth="Container"
    type="javax.sql.DataSource"
    maxActive="100"
    maxIdle="30"
    maxWait="10000"
    username="db_user"
    password="db_password"
    driverClassName="com.mysql.cj.jdbc.Driver"
    url="jdbc:mysql://localhost:3306/your_database"
  />
</Context>
```

2. Define the resource in web.xml

In your web.xml deployment descriptor, add a resource-ref to reference the DataSource defined in the server.

Example web.xml snippet:

xml

```
<resource-ref>
  <res-ref-name>jdbc/YourDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

3. Look up the DataSource in your servlet

Use the init() method to perform the JNDI lookup once during servlet initialization. This prevents repeated lookups for every request.

java

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.SQLException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;

public class MyDataServlet extends HttpServlet {
  private DataSource dataSource;

  public void init() throws ServletException {
    try {
      Context initContext = new InitialContext();
      Context envContext = (Context) initContext.lookup("java:/comp/env");
      this.dataSource = (DataSource) envContext.lookup("jdbc/YourDB");
    } catch (Exception e) {
```

```
            throw new ServletException("Failed to get DataSource", e);
        }
    }


    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        try (Connection conn = dataSource.getConnection()) {
            // Your JDBC code goes here
        } catch (SQLException e) {
            // Handle exceptions appropriately
        }
    }
}
```

Basic steps for using JDBC

Regardless of whether you are using a pooled DataSource or a direct DriverManager connection, the core JDBC steps remain the same:

1. Get a connection: Get a Connection object from the DataSource or DriverManager.
2. Create a statement: Create a Statement or PreparedStatement object from the Connection. PreparedStatement is recommended to prevent SQL injection.
3. Execute a query: Use executeQuery() for SELECT statements or executeUpdate() for INSERT, UPDATE, or DELETE statements.
4. Process results: If a ResultSet is returned, iterate through it to process the data.
5. Close resources: Ensure that Connection, Statement, and ResultSet objects are closed to release database resources.

Example using try-with-resources

The try-with-resources statement, available since Java 7, automatically closes JDBC resources, making your code cleaner and more reliable.

java

```
import java.io.IOException;
```

```
import java.io.PrintWriter;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import javax.annotation.Resource;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.sql.DataSource;

public class UserServlet extends HttpServlet {
  // Inject the DataSource using annotation
  @Resource(name = "jdbc/YourDB")
  private DataSource dataSource;

  protected void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    String query = "SELECT id, username FROM users";

    try (Connection conn = dataSource.getConnection();
        PreparedStatement ps = conn.prepareStatement(query);
        ResultSet rs = ps.executeQuery()) {
```

```
            out.println("<html><body><h2>User List</h2><ul>");
            while (rs.next()) {
                out.println("<li>ID: " + rs.getInt("id") + ", Name: " + rs.getString("username") +
"</li>");
            }
            out.println("</ul></body></html>");
        } catch (SQLException e) {
            out.println("Error accessing the database: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```