1. **Introduction to ZapJS**:

ZapJS is a JavaScript library designed to simplify and enhance AJAX interactions in web applications. AJAX enables asynchronous communication between web pages and servers, allowing parts of a page to be updated without requiring a full page reload. ZapJS offers utility functions and a user-friendly API to streamline AJAX operations and simplify HTML element manipulation, making web applications more interactive and efficient.

2. **What ZapJS Does:**

ZapJS facilitates communication between web pages and servers using AJAX. It provides features such as modals, custom attributes , accordion panes, and pagination. These features are designed based on established UI/UX patterns, enhancing code reusability and maintainability.

3. **Why ZapJS is Needed:**

Modern web development can be complex, and ZapJS aims to simplify it, especially for developers new to the field. It offers clear methods for essential tasks like server communication and improving webpage functionality.

4. **How ZapJS Helps:**

ZapJS integrates seamlessly with various web elements, making it flexible and compatible with different server technologies such as Java, Python, PHP, or Node.js. This versatility makes it suitable for a wide range of websites, including single-page applications, content management systems, online stores, and interactive forms.

5. **Advantages of ZapJS:**

ZapJS is expected to ease web development by enabling developers to create websites with smoother communication, pop-up messages, smart forms, collapsible sections, and organized pagination.

6. **Inspiration and Technologies Used:**

ZapJS draws inspiration from other popular tools like React and Angular.js for building user interfaces, as well as technologies like Node.js and Express.js for server-side development.

**Features Implementation:**

1. **Ajax Calls**:

- The `$$$.ajax` function simplifies AJAX-style coding by providing a basic declaration. It allows developers to specify the URL, method type, and success/failure callbacks.

2. **Modal:**

- ZapJS offers a straightforward way to create modal pop-up windows using the `$$$.modal` function. Developers can specify the content of the modals and trigger them with simple function calls.

3. **DS Combo Box:**

- The `$$$(element).fillComboBox` function populates a dropdown list with data from a data source. It validates various properties such as data source existence, text and value types, and the presence of a first option.

4. **Accordion Pane:**

- ZapJS provides a way to convert a specified element into an accordion pane using the `$$$.toAccordian` function. It hides content under headings, allowing users to expand/collapse sections as needed.

5. **Pagination:**

- The `$$$.grid` function generates a paginated grid/table based on the provided data model. It enables pagination, sets the page size, and allows row selection.

By utilizing ZapJS, developers can create modern, user-friendly, and efficient websites with enhanced functionality and interactivity.

**AJAX Example-1**

# HTML Structure:

<!DOCTYPE HTML>

<html lang='en'>

<head>

<meta charset='utf-8'>

<title>AJAX EXAMPLES</title>

<link rel="stylesheet" href="style.css">

<script src='script.js'></script>

<script>

function populateDesignations() {

// AJAX request to fetch designations data

$$$.ajax({

"url": "servletOne",

"methodType": "GET",

"success": function(responseData) {

// Parse response data

var designations = JSON.parse(responseData);

var designationsComboBox = document.getElementById("designationCode");

var obj;

// Populate dropdown list with retrieved data

for (var i = 0; i < designations.length; i++) {

obj = document.createElement("option");

obj.value = designations[i].code;

obj.text = designations[i].title;

designationsComboBox.appendChild(obj);

}

},

"failure": function() {

// Handle failure case

alert("Some Problem");

}

});

}

// Trigger population of designations on page load

window.addEventListener('load', populateDesignations);

</script>

</head>

<body>

<div class="container">

<h1>AJAX Example-1</h1>

<div class="select-wrapper">

<select id="designationCode"></select>

</div>

</div>

</body>

</html>

This example demonstrates the use of AJAX to populate a dropdown list of designations dynamically. When the page loads, an AJAX request is made to retrieve data from the server, which is then used to populate the dropdown list.

# Explanation:

- The HTML structure consists of a dropdown list element with the id "designationCode".

- When the page loads, the `populateDesignations()` function is triggered.

- Within the function, an AJAX request is made to a servlet named "servletOne" using the `$$$.ajax()` function.

- Upon successful response, the received JSON data is parsed, and each designation is added as an option to the dropdown list.

- In case of failure, an alert is displayed to indicate an issue.

- CSS styles can be applied as per the design requirements.

This example showcases how to utilize AJAX to dynamically populate HTML elements, enhancing user interaction and providing real-time data updates without page reloads.

**Combo Box Example**

<!DOCTYPE HTML>

<html lang='en'>

<head>

<meta charset='utf-8'>

<title>AJAX EXAMPLES</title>

<link rel="stylesheet" href="style.css">

<script src='script.js'></script>

<script>

function populateDesignations() {

// AJAX request to fetch designations data

$$$.ajax({

"url": "servletOne",

"methodType": "GET",

"success": function(responseData) {

// Parse response data

var designations = JSON.parse(responseData);

// Fill combo box with retrieved data

$$$("designationCode").fillComboBox({

"dataSource": designations,

"text": "title",

"value": "code",

"firstOption": {

"text": "<Select designation>",

"value": "-1"

}

});

},

"failure": function() {

// Handle failure case

alert("Some Problem");

}

});

}

// Trigger population of designations on page load

window.addEventListener('load', populateDesignations);

</script>

</head>

<body>

<div class="container">

<h1>Combo Box Example</h1>

<div class="select-wrapper">

<select id="designationCode"></select>

</div>

</div>

</body>

</html>

This example demonstrates the usage of ZapJS's `fillComboBox()` function to dynamically populate a combo box (dropdown list) with data retrieved from the server using AJAX. The combo box is populated with designations fetched from the server.

# Explanation:

- The HTML structure consists of a combo box element with the id "designationCode".

- When the page loads, the `populateDesignations()` function is triggered.

- Within the function, an AJAX request is made to a servlet named "servletOne" using the `$$$.ajax()` function.

- Upon successful response, the received JSON data (designations) is parsed.

- The `fillComboBox()` function of ZapJS is then used to populate the combo box with the fetched designations.

- The combo box is populated with the title of each designation as the display text and its code as the value.

- Additionally, a default option ("<Select designation>") is provided at the top of the combo box.

- In case of failure, an alert is displayed to indicate an issue.

This example illustrates how to dynamically populate a combo box using ZapJS's `fillComboBox()` function and AJAX, enhancing user experience by providing real-time data updates without page reloads.

**Accordion Examples**

<!DOCTYPE HTML>

<html lang='en'>

<head>

<meta charset='utf-8'>

<title>Accordion Examples</title>

<link rel="stylesheet" href="style.css">

<script src='script.js'></script>

<style>

/\* Basic styling for the accordion \*/

.accordion {

border: 1px solid #ccc;

border-radius: 5px;

margin-bottom: 20px;

overflow: hidden; /\* Hide overflow for smooth animation \*/

}

.accordion h3 {

background-color: #f0f0f0;

padding: 15px;

margin: 0;

cursor: pointer;

border-bottom: 1px solid #ccc;

transition: background-color 0.3s; /\* Add transition effect for smooth hover \*/

}

</style>

</head>

<body>

<div id='mymy' class='accordion' accordian='true'>

<!-- First accordion group -->

<h3>Heading 1</h3>

<div>

<!-- Content for Heading 1 -->

</div>

<h3>Heading 2</h3>

<div>

<!-- Content for Heading 2 -->

</div>

<h3>Heading 3</h3>

<div>

<!-- Content for Heading 3 -->

</div>

</div>

<div id='gogo' class='accordion' accordian='true'>

<!-- Second accordion group -->

<h3>Heading 111</h3>

<div>

<!-- Content for Heading 111 -->

</div>

<h3>Heading 222</h3>

<div>

<!-- Content for Heading 222 -->

</div>

<h3>Heading 333</h3>

<div>

<!-- Content for Heading 333 -->

</div>

</div>

</body>

</html>

This example demonstrates the implementation of accordion panels using HTML, CSS, and JavaScript. Each accordion group consists of multiple headings, and upon clicking a heading, the corresponding content expands or collapses.

# Explanation:

- The HTML structure consists of two accordion groups, each containing multiple headings and corresponding content.

- Each accordion group is enclosed within a `<div>` element with a unique id and the class "accordion".

- The `<h3>` elements within each accordion group serve as the headings for the accordion panels.

- The `<div>` elements following the `<h3>` elements contain the content for each accordion panel.

- CSS styles define the appearance of the accordion panels, including border, padding, and background color.

- JavaScript functionality can be added to handle the accordion behavior, such as expanding or collapsing panels upon clicking the headings.

Accordion panels provide a compact and organized way to present content, allowing users to easily navigate through different sections. This example demonstrates a basic implementation of accordion panels using HTML, CSS, and JavaScript.

**Modal Example - 1**

<!DOCTYPE HTML>

<html lang='en'>

<head>

<meta charset='utf-8'>

<title>Modal EXAMPLES - 1</title>

<link rel="stylesheet" href="style.css">

<style>

</style>

<script>

function abBeforeOpening()

{

alert('Modal with ab is about to be opened');

return true;

}

function abBeforeClosing()

{

alert('Modal with ab is about to be closed');

return true;

}

function abOpened()

{

alert('Modal with ab is opened');

}

function abClosed()

{

alert('Modal with ab is closed');

}

function createModal1()

{

$$$.modals.show("ab");

}

</script>

<script src='script.js'></script>

</head>

<body>

<button onclick='createModal1()'>Show First Modal</button>

<div id='ab' forModal='true' >

god is great<br>

god is great<br>

god is great<br>

god is great<br>

god is great<br>

<input type='text' id='myTextBox' value='Great'>

god is great<br>

god is great<br>

god is great<br>

god is great<br>

god is great<br>

Last Line

</div>

</body>

</html>

# Explanation:

- The HTML structure includes a button with an `onclick` attribute calling the `createModal1()` function to display the modal.

- The modal content is enclosed within a `<div>` element with the id "ab" and the attribute `forModal='true'`.

- JavaScript functions are defined to handle different events related to the modal:

- `createModal1()`: Function to display the modal window.

- `abBeforeOpening()`: Function triggered before the modal opens.

- `abBeforeClosing()`: Function triggered before the modal closes.

- `abOpened()`: Function triggered after the modal is opened.

- `abClosed()`: Function triggered after the modal is closed.

- CSS styles can be added to customize the appearance of the modal window.

Modal windows are commonly used for displaying important information or interactive content without navigating away from the current page. This example demonstrates how to create and manage a modal window using HTML, CSS, and JavaScript.

**Modal Example - 2**

<!DOCTYPE HTML>

<html lang='en'>

<head>

<meta charset='utf-8'>

<title>Modal EXAMPLES - 2</title>

<link rel="stylesheet" href="style.css">

<style>

</style>

<script>

function abBeforeOpening()

{

alert('Modal with ab is about to be opened');

return true;

}

function abBeforeClosing()

{

alert('Modal with ab is about to be closed');

return true;

}

function abOpened()

{

alert('Modal with ab is opened');

}

function abClosed()

{

alert('Modal with ab is closed');

}

function createModal1()

{

$$$.modals.show("ab");

}

</script>

<script src='script.js'></script>

</head>

<body>

<button onclick='createModal1()'>Show First Modal</button>

<div id='ab' forModal='true' size="200x300" header="Some heading" footer="Some footer" maskColor="#3355ff" modalBackgroundColor="#549933" closeButton="true" afterOpening="abOpened()" afterClosing="abClosed()" beforeOpening="abBeforeOpening()" beforeClosing="abBeforeClosing()" >

god is great<br>

god is great<br>

god is great<br>

god is great<br>

god is great<br>

<input type='text' id='myTextBox' value='Great'>

god is great<br>

god is great<br>

god is great<br>

god is great<br>

god is great<br>

Last Line

</div>

</body>

</html>

This example showcases the creation and customization of a modal window using JavaScript. Upon clicking the "Show First Modal" button, a modal window appears with customized properties such as size, header, footer, colors, and event handlers.

# Explanation:

- The HTML structure includes a button with an `onclick` attribute calling the `createModal1()` function to display the modal.

- The modal content is enclosed within a `<div>` element with the id "ab" and the attribute `forModal='true'`.

- Various custom properties are specified for the modal:

- `size`: Specifies the dimensions of the modal window (200px width and 300px height).

- `header`: Sets the header text of the modal.

- `footer`: Sets the footer text of the modal.

- `maskColor`: Defines the color of the modal backdrop.

- `modalBackgroundColor`: Sets the background color of the modal window.

- `closeButton`: Indicates whether the close button should be displayed (true/false).

- `afterOpening`: Specifies the function to be executed after the modal is opened.

- `afterClosing`: Specifies the function to be executed after the modal is closed.

- `beforeOpening`: Specifies the function to be executed before the modal is opened.

- `beforeClosing`: Specifies the function to be executed before the modal is closed.

- JavaScript functions are defined to handle different events related to the modal, such as before opening, before closing, after opening, and after closing.

- CSS styles can be added to customize the appearance of the modal window.

Customizing modal windows allows developers to tailor the appearance and behavior of modal dialogs according to specific requirements. This example demonstrates how to create a modal window with custom properties and event handlers using HTML, CSS, and JavaScript.

**ZapJs Grid Documentaton**

**Introduction**

ZapJs Grid Prototype is a customizable and interactive grid system developed using JavaScript and CSS. It allows users to display tabular data in a structured format with pagination support. This documentation provides a comprehensive overview of the grid prototype, including its features, usage, and customization options.

**Features**

1. Customizable Grid Layout: The grid layout can be customized using CSS styles to define the appearance of headers, body cells, pagination controls, and overall dimensions.

2. Pagination Support: The grid supports pagination to handle large datasets efficiently. Users can navigate through multiple pages of data using intuitive pagination controls.

3. Horizontal Scrolling Synchronization: The grid synchronizes horizontal scrolling between the header and body divisions, ensuring that column headers remain aligned with the corresponding data columns.

Usage

To use the ZapJs Grid Prototype, follow these steps:

1. Include Dependencies: Ensure that the required JavaScript and CSS files are included in your HTML document. These files contain the grid implementation and styling.

2. Define Data Structure: Define the structure of the data to be displayed in the grid. In the provided examples, the `Student` class represents individual student records with properties like roll number, name, mother's name, and father's name.

3. Initialize Grid: Create an instance of the `Grid` class, passing the ID of the HTML elements for the grid's header, body, and pagination sections, along with the dataset and page size.

4. Customize Styles (Optional): Customize the grid layout and appearance by modifying the CSS styles provided in the documentation. You can adjust dimensions, colors, borders, and other visual elements according to your requirements.

5. Load Data: Populate the grid with data by providing an array of objects representing the dataset. Each object should adhere to the defined data structure (e.g., `Student` objects in the provided examples).

6. View and Interact: Once the grid is initialized and populated with data, users can view and interact with the tabular representation. They can navigate between pages using pagination controls and scroll through large datasets seamlessly.

Customization

Users can customize various aspects of the grid, including:

- Layout: Adjust dimensions, borders, spacing, and alignment to achieve the desired visual presentation.

- Styling: Modify colors, fonts, backgrounds, and other visual attributes to match the design aesthetics of the application.

- Pagination Controls: Customize the appearance and behavior of pagination controls, such as size, position, and styling.

- Data Display: Implement custom logic to format and display data in the grid cells according to specific requirements.

**Example**

<!DOCTYPE HTML>

<htmal lang='en'>

<head>

<meta charset='utf-8'>

<title>ZapJs Grid Prototyping</title>

<style>

.zapjs\_tmgrid\_header\_division

{

overflow-x: hidden;

overflow-y:hidden;

width:300px;

}

.zapjs\_tmgrid\_body\_division

{

overflow-x:auto;

overflow-y:auto;

width:300px;

height:400px;

}

.zapjs\_tmgrid\_head

{

border:1px groove green;

border-bottom:0px;

border-spacing:0px;

border-collapse:collapse;

background:lightgray;

width:750px;

}

.zapjs\_tmgrid\_head td

{

border-left: 1px groove red;

border-right: 1px groove red;

border-top:1px groove red;

border-bottom:0px;

width:50px;

text-align:right;

}

.zapjs\_tmgrid\_head td+td

{

width:100px;

text-align:left;

}

.zapjs\_tmgrid\_head td+td+td

{

width:200px;

text-align:left;

}

.zapjs\_tmgrid\_head td+td+td+td

{

width:200px;

text-align:left;

}

.zapjs\_tmgrid\_head td+td+td+td+td

{

width:200px;

text-align:left;

}

.zapjs\_tmgrid\_body

{

border:1px groove green;

border-spacing:0px;

border-collapse:collapse;

width:750px;

}

.zapjs\_tmgrid\_body td

{

border:1px groove red;

width:50px;

text-align:right;

}

.zapjs\_tmgrid\_body td+td

{

width:100px;

text-align:left;

}

.zapjs\_tmgrid\_body td+td+td

{

width:200px;

text-align:left;

}

.zapjs\_tmgrid\_body td+td+td+td

{

width:200px;

text-align:left;

}

.zapjs\_tmgrid\_body td+td+td+td

{

width:200px;

text-align:left;

}

.zapjs\_tmgrid\_pagination td

{

width:20px;

text-align:center;

}

.zapjs\_tmgrid\_pagination\_division

{

margin-top:20px;

}

.zapjs\_tmgrid\_pagination\_division a

{

background-color:red;

color:white;

padding:10px 18px;

text-decoration:none;

text-transform:UpperCase;

}

</style>

<style>

.someStyle

{

width:900px;

height:600px;

border:1px solid red;

overflow:auto;

}

</style>

<script>

class Student

{

constructor(rollNumber,name,nameOfMother,nameOfFather)

{

this.rollNumber=rollNumber;

this.name=name;

this.nameOfMother=nameOfMother;

this.nameOfFather=nameOfFather;

}

}

class Grid

{

constructor(dataTableId,dataTablePaginationId,students,pageSize)

{

this.dataTableId=dataTableId;

this.dataTablePaginationId=dataTablePaginationId;

this.students=students;

this.pageSize=pageSize;

this.pageNumber=1;

this.numberOfPaginationControls=5;

this.update();

this.updatePagination();

}

setPage(pageNumber)

{

this.pageNumber=pageNumber;

this.update();

this.updatePagination();

return false;

}

update()

{

var dataTable=document.getElementById(this.dataTableId);

while(dataTable.rows.length>0) dataTable.deleteRow(0);

var tr;

var td;

var startFromIndex=(this.pageNumber-1)\*this.pageSize;

var endAtIndex=startFromIndex+this.pageSize-1;

if(endAtIndex>=this.students.length) endAtIndex=this.students.length-1;

var x;

for(x=startFromIndex;x<=endAtIndex;x++)

{

tr=document.createElement("tr");

td=document.createElement("td");

td.innerHTML=(x+1);

tr.appendChild(td);

td=document.createElement("td");

td.innerHTML=this.students[x].rollNumber;

tr.appendChild(td);

td=document.createElement("td");

td.innerHTML=this.students[x].name;

tr.appendChild(td);

td=document.createElement("td");

td.innerHTML=this.students[x].nameOfMother;

tr.appendChild(td);

td=document.createElement("td");

td.innerHTML=this.students[x].nameOfFather;;

tr.appendChild(td);

dataTable.appendChild(tr);

}

}

updatePagination()

{

function createPageChangeFunction(obj,pageNumber)

{

return function(){

obj.setPage(pageNumber);

};

}

var startFrom=((Math.floor((this.pageNumber-1)/this.numberOfPaginationControls)+1)-1)\*this.numberOfPaginationControls+1;

var endAt=startFrom+this.numberOfPaginationControls-1;

var numberOfPages=Math.floor(this.students.length/this.pageSize);

if(this.students.length%this.pageSize!=0) numberOfPages++;

if(endAt>numberOfPages) endAt=numberOfPages;

var dataTablePagination=document.getElementById(this.dataTablePaginationId);

while(dataTablePagination.rows.length>0) dataTablePagination.deleteRow(0);

var x;

var tr;

var td;

var anchor;

tr=document.createElement("tr");

if(startFrom>1)

{

td=document.createElement("td");

anchor=document.createElement("a");

anchor.text="prev";

anchor.href='javascript:void(0)';

anchor.onclick=createPageChangeFunction(this,startFrom-1);

td.appendChild(anchor);

tr.appendChild(td);

}

for(x=startFrom;x<=endAt;x++)

{

td=document.createElement("td");

if(x==this.pageNumber)

{

td.innerHTML="<b>"+x+"</b>";

}

else

{

anchor=document.createElement("a");

anchor.text=x;

anchor.href='javascript:void(0)';

anchor.onclick=createPageChangeFunction(this,x);

td.appendChild(anchor);

}

tr.appendChild(td);

}

if(endAt<numberOfPages)

{

td=document.createElement("td");

anchor=document.createElement("a");

anchor.text="next";

anchor.href='javascript:void(0)';

anchor.onclick=createPageChangeFunction(this,endAt+1);

td.appendChild(anchor);

tr.appendChild(td);

}

dataTablePagination.appendChild(tr);

}

}

var model={

"grid":null

};

function loader()

{

var zapjs\_tmgrid\_header\_division=document.querySelector(".zapjs\_tmgrid\_header\_division");

var zapjs\_tmgrid\_body\_division=document.querySelector(".zapjs\_tmgrid\_body\_division");

zapjs\_tmgrid\_body\_division.addEventListener(

'scroll',function(){

zapjs\_tmgrid\_header\_division.scrollLeft=zapjs\_tmgrid\_body\_division.scrollLeft;

});

var i;

var students=new Array();

for(i=101;i<1000;i++)

{

students.push(new Student(i,i+"Name",i+"Mother",i+"Father"));

}

model.grid=new Grid("dataTable","dataTablePagination",students,35);

}

window.addEventListener('load',loader);

</script>

</head>

<body>

<div class='someStyle'>

<div class='zapjs\_tmgrid\_header\_division'>

<table class='zapjs\_tmgrid\_head'>

<tr>

<td>S.No.</td>

<td>Roll No.</td>

<td>Name</td>

<td>Mother</td>

<td>Father</td>

</tr>

</table>

</div>

<div class='zapjs\_tmgrid\_body\_division'>

<table class='zapjs\_tmgrid\_body' id='dataTable'>

</table>

</div>

<div class='zapjs\_tmgrid\_pagination\_division'>

<table class='zapjs\_tmgrid\_pagination' id='dataTablePagination'>

</table>

</div>

</div>

</body>

</html>

Conclusion

The ZapJs Grid Prototype provides a flexible and efficient solution for displaying tabular data in web applications. With its customizable layout, pagination support, and seamless scrolling synchronization, it offers a user-friendly interface for visualizing datasets of any size. By following the usage guidelines and exploring customization options, developers can integrate the grid prototype into their projects and enhance the data presentation experience for end users.