# Getting Started with Xtens

## Deprecated Functions

The following functions had been used in previous Xtens but are now deprecated and would be removed in future versions. These should not be used.

* wxfns.addClass(element, cls) -> Use element.addClass(cls) instead
* wxfns.removeClass(element, cls) -> Use element.removeClass(cls) instead
* wxfns.hasClass(element, cls) -> Use element.hasClass(cls) instead
* wxfns.toggleClass(element, cls) -> Use element.toggleClass(cls) instead
* wxfns.transaction(element, ...) -> Use element.transaction(...) instead
* wxfns.upload(element, ...) -> Use element.upload(...) instead

These class related functions are attached to element prototype in element.js and Xten related functions are attached in wx.js during initialiseXtens.

## Introduction

Wohola X uses WXML as a programming language. By default any normal html, css and javascript can be served using the WX server. However, that is not of much use, as that can by done with any other webserver. The main purpose of WX server is to serve WXML files which are enriched with additional features.

WXML is simply and enhanced version of HTML, where all standard HTML tags are a part of the core language by default. Over and above the standard HTML tags the option exists to define new HTML like tags which can be used in the code. These new tags are called Xtens, and each Xtens essentially enhances the HTML in some way. The combination of HTML and the Xtens is what is called a new language, WXML.

## What is an Xten and what are the advantages?

Xten is a new HTML tag which can be defined by Wohola X programmers and used in WXML. HTML by itself does not contain any functionality, but Xtens are required to contain all the features and functions related to the tag within itself.

So for example if there is an Xten called <image> then all the functions and features related to image should be encapsulated inside the <image> xten. The WXML concept is that once the programmer uses <image> Xten then managing the image, saving data, retrieving data for that image and any other functions and features related to image should be taken care of within that Xten.

The advantage of this approach is that once an Xten is defined then it can be called anywhere and the related features and functions become automatically enabled. So for each feature of function code needs to be written once and then it can reused in all projects from that point on.

It also reduces the time required for coding or testing. Because instead of having to write similar code again and again for each project, which programmers end up doing in all projects, the programmers can reuse the previously written code in all future projects. Also, once any Xten is coded and tested then from that point on that should work so there is reduced probability of defects and less time spend on testing.

An Xten encapsulates all the functionality of the relevant feature. In general for implementing a typical feature there is a need for some HTML, some CSS, some client side javascript, some server side code and some database related code. In a typical project developers use disparate technologies for implementing these and hence there end up multiple code files, some running on server and some on client side. Often it become difficult to understand the flow of data or the overall logic is such a code. In an Xten everything related to an Xten is together in the same file. So HTML, CSS, client side JS, server side JS, database functions are there together, so it is easier to understand the flow of information and data and the logic of the Xten. Being able to understand and have control over the logic is the most critical aspect of development, which is often missed by a lot of popular technologies which rely on making things more complex for developers and hiding things behind black boxes, so often developers end up taking a long time to debug code or find flaws in logic in such systems. In Wohola X such issues should ideally be minimum as the flow is simple and visible in a single place within an Xten. So overall building new features through Xtens typically should take less time than building components or extensions in other much more complex systems. As such there is no value addition that comes from the relative complexity. If a complex system can be build software then Wohola X can also be used to build the same software without having the need for unnecessary complexity.

## Understanding the flow of HTTP requests.

Normally the client side code is written with different technologies and server side code is written in different technologies and often there are different programmers who do this. Also, it is common to see that many programmers do not know how, when and where each component of this code runs. Since an Xten takes care of both the client side and server side functionality in a single component it is important for programmers to understand the process flow.

### HTTP Requests

The job of the web server is to serve files. The image, video and other such media files are typically served as is. However, the functionality and features are implemented by sending HTML, CSS and JS files. In general the files like JS, HTML and CSS are nothing more than text files. Only difference is that each of these text files follow a different syntax.

An application program like PHP or Python or node.js may either serve these files as is or modify the JS, HTML and CSS dynamically or create this code dynamically, but each of these programs essentially sends back these files in text format but with the syntax of the relevant language.

So when an HTTP or HTTPS request is typed in a browser the domain name and its linked server IP is resolved through nameservers and DVS entries and the request is sent to the server. The HTTP requests are sent to port 80 and HTTPS requests are sent to port 443.

The server then does whatever processing it needs to do, including the database operations and then server sends back HTML, CSS and JS back to client browser. Once client browser gets the HTML, CSS and JS it first renders the page using HTML, then applies CSS styles to that page and finally it runs the javascript code that it received back from server.

### Manipulating the output of HTTP requests

The easiest to understand are programs like Python or PHP. The only job of these programs is to generate some text output. This text output is then sent back to the server. If this text output follows the correct syntax of HTML, CSS and JS then the browser is able to successfully run this code on client side.

Since its only text output which is basically a string, its possible for server side programs to create this string dynamically or manipulate such a string before sending the final text to the client. As long as whatever text is being sent to the client has the correct HTML, CSS and JS syntax it would run fine in the browser.

### Manipulating the output through Xtens

Mainly the responsibility of each Xten is to do whatever backend and database processing it needs to do and then based on that send back correct HTML, CS and JS to browser, but only in relation to a single node within the whole file. Most of this manipulation is simply string manipulation. For example if you create a string variable in your server program which contains the correct JS code and then send that string back to client then browser would be able to run that JS you sent back which was actually simple text string in a variable.

So each Xten manages the correct HTML, CSS and JS related to itself and Wohola X server combines all the HTML, CSS and JS and sends it back to client and that becomes the HTML, CSS and JS that the browser runs which ends up creating the page that the user sees in the browser.

### Wohola X control flow

There are various point of execution at which points certain Xten functions are executed in Wohola X. These points are:

1. When WX server is started – This point of execution is called server start
2. When browser sends a page request to server – This point of execution is called page request
3. When server loads the data that is relevant to that page – This point of execution is called page load
4. When browser receives the HTML, CSS and JS back from WX server – This point of execution is called page render
5. When browser receives an event due to a user action or a message it receives – This point of execution is called an event
6. When browser calls a transaction on server as part of event handling – This point of execution is called a transaction

These are the main points of execution where browser and client interact and these are the times when the correct data needs to be passed from browser to server or vice versa. You need to be aware of these points of execution and the Xtens would need to handle each of these points correctly if these Xtens have to be able to handle all functionality related itself.

## Modular Architecture

Wohola X uses Modular Architecture or it can be called a plug-in architecture. Out of all the various design patterns and architectures this type of architecture is most powerful, easiest to understand and maintain and most scalable. Also, with such an architecture there is less of a need to have complete requirements of the output system to be built and yet it can be made to work. To understand this architecture consider the following simple example.

Let us say you were given a requirement to create a library for formatting variables in different formats to a consistent output format. The client is not sure which all different formats there may be, and you are not sure either. Also there is another problem that in future as customer requirements change new formats may be required. So as a starting point you assume some basic formats may be required like date, time, currency, integer and string etc.

So in general programmers either create a file where they create various formatting functions like:

* Function format\_date(input)
* Function format\_time(input)
* Function format\_int(input)

Or they may create a class and put these in the class as methods. Then typically most programmers would use if conditions or switch statements to call different functions. For simple functions it may work fine but when instead of formatting these end up complex operations then it starts to become unwieldy.

Using modular architecture is perfect solution to solve this problem. In this case if you put all formatting functions in separate files outside of the main code directory, then as and when new formats are added you can keep adding more and more such files.

The calling program simply loads the file dynamically and calls the correct function dynamically. In this case the logic could be such that a variable contains which formatting function is required and using that variable the correct file can be loaded and called. This way you can create a system without knowing all the different formats that may be needed and keep adding more and more formats as and when needed without touching the rest of the code.

Wohola X uses such an architecture for obvious reasons. We don’t know what all Xtens the programmers may create in future. When creating those Xtens the programmers should not be touching the core WX code. So by creating a modular architecture in Wohola X the capability for adding more and more Xtens is enabled.

## Xten Interface

### Xten Features

Each xten may have some HTML, some CSS and some client side and some server side JS. Any requirement that the Xten fulfils we call it a feature.

The features may have 2 types of scope: a global feature or feature with global scoope and a node feature or a feature with node scope. A node is an individual WXML or HTML element.

A global feature is something that is applicable to all Xtens, i.e. it has a global scope, while node features are applicable to individual node and hance they have node scope. For example if you are using an Xten that implements a text box and you want that all text boxes should have a red border then ‘red border’ is a global feature. But if you decide that every text box could have a different font color then ‘font color’ becomes a node feature.

There are different Xten properties to handle global features and node features. Let us understand basic properties through our first Xten <hello-world>.

### Hello World Xten

As is typical for learning any programming language let us try to create a simple hello-world Xten that only does one thing. It puts text Hello World inside a div. Let us take a look at the code:

<xten name="hello-world" element="div">

    <init>

        node.text = 'Hello World';

    </init>

</xten>

<page url="xten-training">

    <hello-world/>

</page>

We have a new page with url xten-training. We are also defining an xten named hello-world and using it in our page.

The Xtens need to be defined anywhere before they are used but it is not necessary for these to be in separate files. The pages and Xtens are put in separate files for readability purposes but all the WXML could be in a single file as well.

In the example we are using <element> property of the Xten and setting it to div. This property allows you to control which HTML element your output code would be wrapped in.

Then we are using <init> property and putting a simple javascript code in there which sets the text of the node to Hello World. Here node is a predefined variable that <init> property receives from the WX server. In this case node is the WXML element <hello-world/>. Within that node the above code would change the element tag from hello-world to div and put text inside that node. So the output of this Xten would be:

<div>

    Hello World

</div>

So where you see <hello-world/> That part of the code would be replaced with above div code before sending the data back to browser.

It is necessary to do so since the browser does not understand WXML. The browsers understands standard HTML, CSS and JS. This is what all server side systems do, whether its Python, PHP, .Net or any other system. The browsers do not understand Python, PHP or other languages. The server side code may have views in whatever format that is understood by PHP or .Net for example but each of these views are processed by backend system and finally all backend technologies generate some standard HTML, CSS and JS and send it back to browser. This is exactly what WX server also does.

So regardless of the processing that is required the main purpose of the Xten is to ensure that correct standard HTML, CSS and JS is generated in relation to the functional requirements of the Xten. The Xten manages the output being generated through Xten properties.

You as a programmer know what requirement you are trying to implement so you would define the Xten properties in such a way that your Xten meets the functional requirements you have.

Specifying Complex HTML

In a simple example like above it is relatively easy to set the text of the node. But some Xtens may require a complex HTML as a starting point. Such HTML can be specified using <view> property. The same example above can be changed to provide same output by using a <view> property.

<xten name="hello-world" element="div">

    <view>

        Hello World

    </view>

</xten>

The <view> can contain any WXML or HTML code. So you can create complex structures within the <view> property.

An example could be you want your Xten to have a heading ‘Hello World’ and a text below it saying ‘Welcome to the Wonderful World of Wohola X!’ so you may have a code such as this.

<xten name="hello-world" element="div">

    <view>

        <inline name="hello-world-heading">Hello World</inline>

        <textedit>Welcome to the Wonderful World of Wohola X</textedit>

    </view>

</xten>

Now your view contains WXML that has an editable heading and an editable text block.

### Xten properties

So far we have seen 4 Xten properties, <name>, <element>, <init> and <view>. Now let us delve a bit deeper into these properties.

Based on the 2 scope types, global and node, and the various points of execution namely server start, page request, page load, page render, event and transaction, there are different properties which are linked to each of these points of execution.

The ‘Xten properties reference.doc’ lists all the Xtens features that are currently implemented. But for ease of learning we would start with use case based properties so that the learning could be in the form of which properties to use to implement what feature.

Since Xten generate standard HTML, CSS and JS there needs to be a way for a programmer to specify the relevant HTML, CSS and JS snippets which get plugged into the final output being sent to the browser.

We would look at each of these one at a time and understand the relevant Xten properties for implementing the required features.

# CSS manipulation

### Applying styles at global scope

Let us say in the hello world xten we want to have a red border. For this we can use <style> property like so:

<xten name="hello-world" element="div">

    <view>

        <inline name="hello-world-heading">Hello World</inline>

        <textedit>Welcome to the Wonderful World of Wohola X</textedit>

    </view>

    <style>

        border: 1px solid red;

    </style>

</xten>

<style> property has same syntax as inline styles in HTML. In this property any standard CSS properties can be used.

This style is applied at global scope. So this feature of having a red border is a global feature. The feature that Xten adds a text Hello Word to the text element of the node is also a global feature.

### Applying styles at node scope

Now let us assume the same Xten is being used in 3 different places on same page or in different pages and we want to give a flexibility to WXML programmer that he can configure the font color so every Xten could have a different font color as desired by WXML programmer.

In that case look at this code example:

<xten name="hello-world" element="div">

    <view>

        <inline name="hello-world-heading">Hello World</inline>

        <textedit>Welcome to the Wonderful World of Wohola X</textedit>

    </view>

    <style>

        border: 1px solid red;

    </style>

    <nodeStyle>

        color: [$color||initial];

    </nodeStyle>

</xten>

<page url="xten-training">

<xten name="hello-world" />

    <xten name="hello-world" color="blue" />

    <xten name="hello-world" color="green" />

</page>

We added property <nodeStyle> to the Xten but we used a strange syntax in standard CSS “[$color||initial]”. What we have done is that we used standard CSS property which is color but in its value we used a variable named $color.

### Variables within CSS

We don’t need to define variables, we can simply start using non-existent variables and WX would create those variables for us. The syntax of a using a variable is as shown. Here we are setting the color CSS property to whatever value WXML developer passes to the Xten in $color variable. If the WXML developer does not pass any value in $color variable then we default to initial. “initial” is a standard CSS property which means whatever the default value that CSS property is supposed to have.

If you look at the page code where this Xten is being used, there are 3 instances of that Xten being used. In one instance there is no value being passed so color CSS property would default to initial. In the other two instances the <nodeStyle> would resolve to color: blue; and color: green; respectively and hence the font color would change accordingly.

This way by using variables the Xten programmers can make their Xten configurable which WXML programmers can configure as per their requirements while using the Xten.

**You can use variables within node scope but not within global scope.**

### Applying complex CSS

The <style> and <nodeStyle> apply style to individual node of the Xten with difference being at global scope i.e. same property being applicable to all instances vs at node scope i.e. each node could have a different value.

These properties only manipulate styles of individual nodes though. For a complex Xten the relevant node may have multiple children or a complex HTML structure. In such cases there may be a requirement for manipulating styles of certain elements within that complex structure.

Let us take an example where we want our Xten to have a feature that the heading has a light grey background and the text is bold. We also want to give flexibility to the WXML programmer to be able to change the font color of the heading and the text separately.

Now we cant have the color property in the Xten <style>, but we now need to apply a selector to find the correct element within the view and apply these styles. Which means we would need to use CSS syntax that allows us to use selectors. This is done using <css> and <cssNode> properties, with the difference being <css> property applies the given CSS at global scope and <cssNode> property applies the CSS at node scope.

Here is the updated example code:

<xten name="hello-world" element="div">

    <view>

        <inline name="hello-world-heading">Hello World</inline>

        <textedit name="hello-world-text">

Welcome to the Wonderful World of Wohola X

</textedit>

    </view>

    <style>

        border: 1px solid red;

    </style>

    <css>

        .wx-xten-hello-world [name="hello-world-heading"]

        {

            background-color: lightgrey;

            font-weight: bold;

        }

    </css>

    <cssNode>

        #[$id] [name="hello-world-heading"]

        {

            color: [$heading-color||initial];

        }

        #[$id] [name="hello-world-text"]

        {

            color: [$text-color||initial];

        }

    </cssNode>

</xten>

<page url="xten-training">

    <xten name="hello-world" heading-color="blue" />

    <xten name="hello-world" heading-color="green" text-color="red" />

</page>

In this code <css> property would apply the relevant CSS at global scope, i.e. same CSS is applicable to all instances of that Xten. While <cssNode> property would apply the relevant CSS to individual nodes and these can be configured by WXML programmer.

In <css> property we are selecting all instances of the xten through selector ‘.wx-xten-hello-world’ and applying the relevant CSS. WX adds ‘.wx-xten-xten-name’ class to all Xtens automatically so ‘.wx-xten-hello-world’ class exists and can be used to select all instances of the Xten.

The CSS shown is standard CSS with standard CSS selectors. For example to select an element by id we use #id, to select an element by class we use ‘.classname’ and to select an element by attribute and value we use ‘[attributename=”attributevalue”]’. These are all standard CSS and CSS selectors which we would have normally written in the .css files if we were not using Wohola X.

In <cssNode> property we want to apply CSS individually to each node so we select each node separately using #[$id]. It is standard syntax to select any element in CSS using #id-of-the-element, and same is being used here. [$id] variable resolves to the id of the individual Xten element. The ids themselves are added and managed by WX automatically so you do not need to and should not give id to any WXML element. The main thing to remember is for node scope you get the individual node through #[$id].

Then we have defined 2 variables for heading-color and text-color and these are being used by WXML programmer to configure individual Xten instance to have different colors in the <page> code.

**You can use variables within node scope but not in global scope.**

### Responsive CSS

Each Xten is responsible for making sure it is responsive. So to make that Xten responsive if any CSS needs to be applied then that must also be defined within the Xten.

The properties that allow this are <tablet>, <phone> at global scope and <nodeTablet> and <nodePhone> at node scope. Like in normal CSS the selector at global scope is .wx-xten-name-of-the-xten and at node scope is #[$id]

In <nodePhone> and <nodeTablet> you can use variables that user can pass values to.

Whether to apply normal CSS or tablet CSS or phone CSS is dependent on the width of the screen. By default the tablet width is defined as 800px and phone width as 600px. For your project if you want to change these values, you can change these default values by specifying a different value in the main project xml file like this:

<tablet-width>850</tablet-width>

<phone-width>550</phone-width>

These must be defined before any <page> xtens.

**You can use variables within node scope but not in global scope.**

## Other basic Xten properties

Before we delve into functional Xten properties, i.e. properties through which we can implement logic, let us look at some basic properties. We would look at a code that has all these values in use.

<mandatory>: any variables that are mandatory can be listed here so that if WXML programmer does not provide values to those variables the WX server would give a syntax error while starting. This way the WXML programmer can fix that error by passing value to the mandatory variable.

Example below makes mandatory name and type variables so while calling this Xten the WXML programmer must define name and type attributes.

<mandatory>name, type</mandatory>

<allow>: this contains the list of user roles that are allowed to edit, if the Xten allows editing of content.

Example below means that owner of the object i.e. who created the object and the admin have rights to edit the contents managed by that Xten.

<allow>owner, admin</allow>

<\_class>: this would be used if you want to add your own class to all Xten instances. This may be useful if you want to have your own external CSS. This has a node scope.

Example here would add a class to each node of the Xten. If orientation is row then navclass-row is added to the node and if orientation is column then navclass-column is added the node.

<\_class>navclass-[$orientation||row]</\_class>

<parentClass>: this would add your own class to the parent node of the Xten node.

Example: For correct tooltip positioning the CSS needs to be put in the parent element of the tooltip so the following code adds a class to parent node of the tooltip Xten node and then within <css> property or in JS code we can add the code for positioning the tooltip relative to an element of the tooltip-parent class.

<parentClass>tooltip-parent</parentClass>

<attribs> : this would add your own custom attributes the Xten node.

<attribs>

    <myattr1>value 1</myattr1>

    <myattr2>value 2</myattr2>

</attribs>

<useView>: this would be used if you want to use a view for that Xten which is defined outside of Xten. This would be applicable if you want a view for your Xten which you want WXML programmer to be able to override without affecting the Xten code. This is also useful in situations if more than one Xten use the same view so view can be defined outside and then both Xtens can call it using <useView> property.

Example in loginform Xten the view of login form may be overridden on per project basis so it is defined separately and useView is used to load it from outside. This way the WXML programmer can override the view without overriding the Xten.

<useView>v.loginform</useView>

<link>: if you want your Xten to load external CSS files. An example is:

<link>

[https://fonts.googleapis.com/css2?family=[$name]:wght@[$weight]&display=swap](https://fonts.googleapis.com/css2?family=%5b$name%5d:wght@%5b$weight%5d&display=swap)

</link>

<libs> : If you want your Xten to load some external js or css files. The paths would be relative to the public folder of the site directory. Example:

<libs>libs/myjs.js, libs/mycss.css</libs>

<toolbar> : to add a toolbar to your Xten’s main element. If you want the toolbar in a child element of your Xten it should be added there. Using <toolbar> property would only add toolbar in the node of the Xten. The syntax and usage of <toolbar> is as defined in <toolbar> Xten in xten list document.

<append>: It works like a view except that instead of replacing node content with view content, it appends the WXML to the node content.

<prepend>: It is similar to append except it prepends the given content to the node content.

## Wohola X Flow

The idea is that each Xten would generate the correct HTML, CSS and Javascript that would be sent to the browser. This output should be such that if it runs in browser it achieves the desired functionality. So Wohola X does not perform any functions directly like most libraries or APIs. Instead it only provides a framework with which you can generate HTML, CSS and JS which would perform the required functions.

### The flow of execution for Wohola X

1. When the WX server starts, for each page and for each Xten within that page the <init> function is called on server side. When <init> is called there is no database available and this is called only once at the server start. So any initialization that needs to be done that is not database dependent can be done here.
2. The other properties are also applied to the given node at the server start.
3. When the page is loaded for each Xten the <load> function is called on server side. Other properties and <init> function is not called. At the time of <load> function the database connection is available. The primary purpose of the <load> function is to read data from database and put it in appropriate place in the correct HTML within the given node.
4. The server side code like <init> function and <load> function is run in node.js platform using Javascript.
5. The primary function of <init> and <load> function is to convert the node which is in WXML format to standard HTML format which can be understood by the browser. The CSS properties are automatically applied as defined in the Xten.
6. Any JS code defined in Xten are packaged and sent as to the browser so that browser can execute those JS functions.
7. Once the HTML and CSS has been sent to the browser the <initElement> function is called on each Xten on client side. The purpose of the <initElement> is to do any client side initializations and then attach events to the element.
8. For adding events there are also shortcuts available through <event> and <click> Xten properties. The events can also be added through ‘addEventListener’ Javascript function. If the events are to be added to main element then these can be added through <event> and <click> properties of Xten. If these are to be added to a descendant element of the main element then these must be added through addEventListerner only.
9. Wohola X Javascript is standard Javascript only and does not use any external frameworks.
10. The main purpose of any Javascript program is to manipulate and initialise the HTML content it received from the browser and then perform some actions based on events. So <initElement> function also does exactly that, you can put any initialization code in there and then attach events to the element, which would be executed when the events occur. The important thing is to make sure that <initElement> function only acts on the given element and only manages the requirements of the given Xten only.
11. After initialization code is run by the browser and events are loaded then if any event occurs the attached event listener is executed. Often during the event execution some code needs to be run on the server. These are called transactions. The examples could be that on click of a button you may want to save the data or delete data on the server.
12. The transactions can be defined using <txn> property of the Xten. The code in <txn> property runs on the server and is in node.js Javascript format. To call the defined transaction it can be called from within the event listener by using element.transaction function. So the control flow is easily traceable from the client side Javascript in the same Xten to the server side Javascript.
13. This flow is sufficient to be able to do any and all logic for any projects. Wohola X Xten interface has limited number of consistent touchpoints and once these are understood Xtens for any functionality could be built. For logic flow <init>, <load>, <initElement>, <event>, <click> and <txn> are the only things that need to be defined and is sufficient to implement any cross platform logic.

### WX server flow

1. When the WX server first starts it runs the preprocessor. As of now there are only 2 preprocessor directives #include and #define. These are handled first during preprocessor phase. The files are included and any variables [#varname] are substituted. The main server file may have many files included in it, but after running the includes WX server ends up with one file for the whole site which is in WXML format.
2. After getting the file the WX server executes various commands in that file, e.g. <db> command creates the database if it does not exist and creates a database connection. Any <xten>, <view> or <page> elements are simply stored and indexed by the server at this stage and they are not executed at this point.
3. After this the WXML is converted to JSON format. This JSON is written to \_cache folder of the site. This allows for manipulation of DOM elements. This JSON is then loaded into an internal DOM format using class DomNode. This class allows for navigation and manipulation of various nodes within the page.
4. After this each page is parsed and Xtens are identified. The Xten properties except <load> are called. For CSS related properties the CSS is stored internally by the server for purpose of serving later when the page request comes from client. At this point <init> is also called.
5. After applying all the properties the resultant page is very close to the standard HTML format except for remaining manipulation that is dependent on the <load> function. This data is however in JSON format in DomNode classes instead of an HTML text string. This is stored for serving later on. This is the compiled page.
6. At this stage any CSS and JS which was defined in Xten is also collated and saved by the WX server. Any CSS which is not dependent on individual elements data can be safely written to external files and their URL could be included in the HTML through <script> or <link> tags. Any CSS or JS which is dependent on element data so could change as data changes is later on added in <script> and <style> tag in the output HTML.
7. The WX server also opens a socket server and attaches to the main webserver.
8. The WX server then listens for client requests and when any requests come in they are handled by the server and resulting data sent back to the client.
9. If the request is for static files like JS files, CSS files, image files, video files etc then these files are sent back to the client as is.
10. If the request is for WXML page then server checks if the output HTML is available in its internal cache. If the HTML is available in cache then the HTML from cache is chosen.
11. If it’s the first time that page has been called after server start then this page would not be there in the cache. So in this case the <load> property is called for each Xten on that page and it results in all Xten elements to be fully in HTML format.
12. Once <load> has been called the DOM is parsed element by element and for each JSON representation corresponding HTML is created which is a text string. This HTML is then stored in cache, so that subsequent requests do not require the page to be recompiled.
13. This output HTML is then served to the client and this may contain links to CSS and JS files which WX server created based on the CSS and JS related properties defined in Xten.
14. If the incoming request is for a transaction then related <txn> function is invoked and any data returned by <txn> is sent back to the client.

### WX client side flow

1. Once the browser receives the HTML from WX server the <initElement> is called on each Xten element. This is called each time the page is loaded in the browser.
2. Within <initElement> you may have some code which does some initialization of the element. For example you may have code to check if editing is allowed or not and based on that turn the toolbar or editing functions on or off.
3. Apart from that you may have some events on the element itself or its children. These are not executed at this point. Only when related events occur this code is executed. The examples of events could be user clicked something or hovered over something or typed something.
4. The browser then waits for the events. When there is an event and if there is an event handler assigned for that event on that element then this event handler code is executed.
5. If during some event some server side transaction is required then element.transaction can be called from event handler to run the transaction you have defined using <txn> property.

## How to implement different types of logic in WXML

### Changing styles based on events

This is a common requirement where based on user events the look and feel of some elements may change. It could be that some background color changes or element becomes bigger or some animation runs. Majority of these things can be done by CSS itself and wherever possible in CSS these things should not implemented using Javascript.

If you have 2 different sets of styles to be applied to an element then you can define a class for your element which indicates a changed state and then define CSS with styles for default state and CSS using this class as qualifier for modified state.

For example if you want that when user hovers over an element then the background color changes and the element becomes bigger. This element has 2 states, one default and one state where it becomes bigger. If your element has a class ‘my-class. the element qualifier is ‘.my-class’ which can be used to define the default CSS. You define another class ‘bigger-element’ for example and use this qualifier where the background color is different and element is bigger. Once you have defined the CSS, one without ‘bigger-element’ class and one with it then your in your Javascript you only need to add and remove the class ‘bigger-element’ from your element to make it change its look and feel.

Here are 3 variations of achieving the same functionality.

<xten name="make-bigger-on-hover" element="div">

    <style>

        background-color: lightblue;

        width: 200px;

        height: 200px;

    </style>

    <css>

        .wx-xten-make-bigger-on-hover.bigger-element {

            background-color: red;

            width: 220px;

            height: 220px;

        }

    </css>

    <initElement>

        element.addEventListener('mouseover', () => {

            element.addClass('bigger-element');

        });

        element.addEventListener('mouseout', () => {

            element.removeClass('bigger-element');

        });

    </initElement>

</xten>

<xten name="make-bigger-on-hover" element="div">

    <style>

        background-color: lightblue;

        width: 200px;

        height: 200px;

    </style>

    <css>

        .wx-xten-make-bigger-on-hover.bigger-element {

            background-color: red;

            width: 220px;

            height: 220px;

        }

    </css>

    <event name="mouseover">

        const element = event.target;

        element.addClass('bigger-element');

    </event>

    <event>

        const element = event.target;

        element.removeClass('bigger-element');

    </event>

</xten>

<xten name="make-bigger-on-hover" element="div">>

    <style>

        background-color: lightblue;

        width: 200px;

        height: 200px;

    </style>

    <css>

        .wx-xten-make-bigger-on-hover.bigger-element {

            background-color: red;

            width: 220px;

            height: 220px;

        }

    </css>

    <event name="mouseover">

        const element = event.target;

        element.addClass('bigger-element');

    </event>

    <event>

        const element = event.target;

            element.removeClass('bigger-element');

    </event>

</xten>

In above code addEventListener is a standard Javascript function and mouseover and mouseout are standard Javascript events. For adding or removing the class element.addClass and element.removeClass functions are used.

Either one of the above code can be saved in make-bigger-on-hover.xml file in some folder of your choice in your site and then included in your server XML file. If an Xten is meant to be generic and used across various projects then it can be added to Wohola X system folder. Any project specific Xtens can continue to be in your site folder.

Now to use this Xten the WXML programmer can do so:

<make-bigger-on-hover>

    <!-- any content that needs to be made bigger on hover -->

    <div style="height: 100%; width: 100%">

        <image name="my-image" style="height: auto; width: 100%">/>

        <inline name="my-text" />

    </div>

</make-bigger-on-hover>

In this case there could be any content within the node and in this case as the Xten node becomes bigger the div inside it would also become bigger and so would the image inside it.

This way by simply adding or removing the class the desired effect was produced. Even many animations or transitions can be done using CSS itself and for all such things possible through CSS that should be used so that code logic should be minimal.

### Notifying and confirming things from user

Let us take a simple example of an Xten which displays a number 0 in a span element and has a button next to it. Each time the button is clicked it first asks the user whether he wants to continue and then increments the number and then tells the user that it is done.

<!-- this is the comment format in WXML -->

<!-- this is the code in xten file -->

<xten name="increment-number" element="div">

<!—

define the view that contains markup you need

all HTML tags are part of WXML by default so they can be used freely

in this case there is no functionality attached to span and button

so we are using normal HTML tags for these in this view

-->

    <view>

        <row>

            <column>

                <span name="number-value">0</span>

            </column>

            <column>

                <button>

                    <span>Click To Increment</span>

                    <click>

                    <%

                        /\*

                        in Javascript code this comment syntax is valid

                        '//'' comments are not implemented in WXML

                            so do not use '//' for comments

                        \*/

                        /\*

                        you get the event object in events.

                        you can find the element on which event occurred by

                            event.target and then closest

                        closest returns the first matching ascendant element

in this case the event happened on button

we need to find the Xten node which is somewhere

above the button in the parent list

closest is Javascript function that does that

the Xten node can be found by using either

.wx-xten-xten-name or [xten=”xten-name”] selector

                        \*/

                        const button = event.target;

                        const element = button.closest(

                            '.wx-xten-increment-number'

                        );

                        /\* confirm whether user wants to continue \*/

                        const confirm = await wxfns.confirm(

                            'Should I increment this number?'

                        );

                        if (!confirm) {

                            return;

                        }

                        /\*

                        now the span which has the number needs to be found

                        span[name="number-value"] is standard CSS selector

                        you should look for the span in current element only

                        not the whole document

                        \*/

                        const span = element.querySelector(

                            'span[name="number-value"]'

                        );

                        /\* increment the value \*/

                        var value = parseInt(span.innerHTML);

                        span.innerHTML = (value + 1).toString();

                        /\* tell the user that job is successfully done \*/

                        wxfns.inform(

                            'I have incremented the value from '

                            + value + ' to ' + (value + 1)

                        );

                    %>

                    </click>

            </column>

        </row>

    </view>

</xten>

<!-- this is how to use this Xten in code in page file -->

<increment-number />

First notice how <click> event is applied directly to the button. <click> or <event name=”event-name”> Xtens can be used to attach any events to any elements directly within that element.

In the above code wxfns.confirm and wxfns.inform functions are used for confirming from user and informing him. More about these and other such functions are covered in ‘Client side functions reference.doc’. Rest is standard Javascript and the selector being used is standard CSS selector.

In the Xten you often need to find the element from the element where the event occurred. WX automatically adds a class wx-xten-xten-name and an attribute xten=”xten-name” so using either ‘.wx-xten-xten-name’ or ‘[xten=”xten-name”] you can find the Xten node from any of the child elements using the closest Javascript function.

The most important point is that we are only trying to act on children of the Xten element. Most Javascript code examples you would see would show all actions being performed on document. In this case that should not be done because each Xten only acts on itself. In the example above if you are using the same Xten more than once on the page and then you did

document.querySelector(‘span[name=”number-value”]’)

then this would always return the span for the first instance of the Xten node so incrementing the value would always happen on first element and not the second one.

**The scope example given above is critical to remember.**

### Saving and retrieving data from the database

Now let us try to save the incremented data in database and retrieve it.

<!-- this is the code in xten file -->

<xten name="increment-number" element="div">

    <mandatory>name</mandatory> <!-- all db related xtens need name -->

    <allow>logged\_in</allow> <!-- allow any logged\_in user to increment -->

    <view>

        <row>

            <column>

                <span name="number-value">0</span>

            </column>

            <column>

                <button name="my-button">

                    <span>Click To Increment</span>

                    <click>

                    <%

                        const button = event.target;

                        const element = button.closest(

                            '.wx-xten-increment-number'

                        );

                        const span = element.querySelector(

                            'span[name="number-value"]'

                        );

                        var value = parseInt(span.innerHTML);

                        value++;

                        /\*

                        save-page-data is a standard txn

                        so no need to define it in Xten

                        it is used by almost every Xten

                        so this has been inbuilt into WX

                        for most cases this is sufficient

                        notice how no query was needed

                        WX figures out where and how to save

                        \*/

                        const result = await element.transaction(

                           'save-page-data', value

                        );

                        /\*

                        all transactions return a JSON object

                        rc is mandatory in the returned object

                        rc can be 'success' or any string

                        if rc is not success then its an error

                        no need to show errors

                        they are shown automatically

                        \*/

                        if (result.rc != 'success') {

                            return;

                        }

                        span.innerHTML = value.toString();

                    %>

                    </click>

            </column>

        </row>

    </view>

    <css>

        .wx-xten-increment-number [name="my-button"] {

            display: none;

        }

        .wx-xten-increment-number.wx-state-editable [name="my-button"] {

            display: inline-block;

        }

    </css>

    <load>

    <%

        /\*

        this gets the correct value for the node except in special cases

        you don’t need to decide which collection, which data etc

        \*/

        const value = await database.inlineValue(node, context, xten);

        /\*

        we now need to put this value in the span element of the node

        here we are using internal DomNode library for DOM manipulation

        in initElement we used querySelector a standard browser function

        \*/

        const span = node.findNamedChild('span', 'number-value');

        /\*

        this value needs to be set in the text field of the span

        in other xtens you may want to put this value elsewhere in your HTML

        \*/

        span.text = value;

        /\* done \*/

    %>

    </load>

</xten>

<!-- this is how to use this Xten in code in page file -->

<increment-number name="my-number" />

We defined <mandatory>name</mandatory> so that name attribute is mandatory while using the Xten hence we provided a name while calling it.

Also for any operation that requires changes to database access must be granted. By default database access is not allowed. You need to allow it using <allow> property. In this case anyone who is logged in can increment the number.

Also to make the button if the user does not have access .wx-state-editable is used in <css> property along with the default Xten class. This way the button would only become visible if the user has access to it.

The client side code uses standard Javascript for DOM traversal and manipulation, while on server side there is an internal DomNode class. DomNode reference is in ‘DomNode reference.doc’. The syntax is different in both and so is how the elements are handled. So on server side the element is called node while on client side its called element. This way there is distinction between the two.

To retrieve data simply call

const value = await database.inlineValue(node, context, xten);

The node, context and xten variables are predefined. WX returns the correct value and there is no need to think about queries and selectors except in some special cases.

To save data from client side simply call

const result = await element.transaction('save-page-data', value);

Normally the data is saved only because of some client operation so these two mechanisms above are sufficient for most of the database related operations.

## Showing popups or modals

There is a common need for popups or modals. We saw wxfns.confirm and wxfns.inform for this before. To get data values from user we need other types of popups and modal dialogs.

### Getting single values

To get single values from user on client side we can use wxfns.getUserValue function. The above example is changed to take value from user instead of incrementing the value.

<xten name="increment-number" element="div">

    <mandatory>name</mandatory>

    <allow>logged\_in</allow>

    <view>

        <span name="number-value">0</span>

        <button name="my-button">

            <span>Click To Increment</span>

            <click>

            <%

                const button = event.target;

                const element = button.closest('.wx-xten-increment-number');

                const span = element.querySelector(

'span[name="number-value"]'

);

                const prevValue = parseInt(span.innerHTML || 0);

                /\* get single value from user \*/

                const value = await wxfns.getUserValue(

                    'my-value-name', /\* any name of your value \*/

                    'number', /\* input type

standard HTML input types taken

\*/

                    'Get Number', /\* title of the popup \*/

                    { value: prevValue } /\* previous value \*/

                );

                /\* get single value from user \*/

                const result = await element.transaction(

'save-page-data', value);

                if (result.rc != 'success') {

                    return;

                }

                span.innerHTML = value.toString();

            %>

        </row>

    </view>

    <css>

        .wx-xten-increment-number [name="my-button"] {

            display: none;

        }

        .wx-xten-increment-number.wx-state-editable [name="my-button"] {

            display: inline-block;

        }

    </css>

    <load>

    <%

        const value = await database.inlineValue(node, context, xten);

        const span = node.findNamedChild('span', 'number-value');

        span.text = value;

    %>

    </load>

</xten>

<increment-number name="my-number" />

The function wxfns.getUserValue was wxfns.getValue in version beta 1.1.9 and below. wxfns.getValue is now deprecated. From release beta 1.2 use wxfns.getUserValue.

wxfns.getUserValue takes any standard input type as type argument. For example if you pass the type as ‘color’ it opens up a color picker. You can check the reference for standard input types in HTML.

### Getting multiple values

If you want a simple popup that gives some predefined values then you can also create a popup on the fly like this.

    var data = {}; /\* some data object with previous values \*/

    const list = [

        {

            name: 'smtp-host',

            label: 'SMTP Host',

            tag: 'input',

            type: 'text',

            value: data['smtp-host'],

            required:'true'

        },

        {

            name: 'smtp-port',

            label: 'SMTP Port',

            tag: 'input',

            type: 'number',

            value: data['smtp-port'],

            required: 'true'

        },

        {

            name: 'user-name',

            label: 'SMTP User Name',

            tag: 'input',

            type: 'text',

            value: data['user-name'],

            required:'true'

        },

        {

            name: 'password',

            label: 'SMTP Password',

            tag: 'input',

            type: 'password',

            value: data['password']

        },

        {

            name: 'ssl',

            label: 'SSL?',

            tag: 'input',

            type: 'checkbox',

            value: data['ssl'],

        },

        {

            name: 'from',

            label: 'From Email',

            tag: 'input',

            type: 'email',

            value: data['from'],

        }

    ];

    const formData = await wxfns.getMultipleValues(list, 'SMTP Settings');

    if (!formData) {

        return;

    }

    result = await element.transaction('save-field-value', formData);

    if (result.rc != 'success') {

        return;

    }

In tag field you can decide to put input, textarea, radio or checkbox etc. You cannot put <select> in this form but other input types are supported. The type field specified the input type which is standard.

This creates a popup and returns the values entered by the user and you can do further processing based on the returned values.

### Creating complex modal dialogs

For complicated popups <modal> Xten can be used.

If you want a modal popup to be linked to your Xten then you can create a view with <modal> Xten in it like this:

<initElement>

<%

    element.modal = element.querySelector('.wx-xten-modal');

%>

</initElement>

<view>

    <!-- you can have other WXML code that is needed in your view here -->

    <modal name="image-list-dialog" heading="Image List">

        <div>

<!-- you can put any WXML code that you need -->

</div>

        <div>

<!-- you can put any buttons and do anything in click -->

            <button name="select-image">

                <span>Select</span>

                <click>

                <%

                    const element = event.target.closest(

'.wx-xten-image-list');

                    const data = {}; /\* get updated data for your code here \*/

                    element.modal.save(data);

                %>

                </click>

            </button>

            <button name="cancel">

                <span>Cancel</span>

                <click>

                <%

                    const element = event.target.closest(

'.wx-xten-image-list');

                    element.modal.cancel();

                %>

                </click>

            </button>

        </div>

    </modal>

</view>

The modal was found using the class name and then the value was stored in a variable in the element itself so this value is available everywhere else including the toolbar. You can store any values you need in the element and these values can then be accessed from other functions of your Xten.

The modal has 2 functions: cancel and save. In save you can pass some data and this data would be available to the calling program.

To open this modal on let us say a button click, you can put the following code in the click event of your icon.

    const result = await element.modal.open(previousData);

    if (result.action == 'cancel') {

        return;

    }

    const newData = result.data;

Since the actual WXML you are passing to the modal is your own code and your own logic you need to figure out which data is initial value and what to do with the changed value received after user modified the values in the modal and clicked save.

You can have multiple modals with different names in your Xten. Simple select the correct modal based on the name and call open on it.

## Passing data from client to server and back

### To pass data from client to server

To pass data from client to server define a transaction using <txn> and then pass whatever data is required. The code in <txn> would run on server and process this data.

<!-- this could be some icon in some toolbar in your code -->

<icon name="cross">

    <click>

    <%

        /\*

            you can call the transactions in your event handlers

            or any client side Javascript code

get the element from the event.target first

        \*/

        const result = await element.transaction(

            'my-transaction',

            {'key1': 'some data', 'some-other-key': 'other data'});

        if (result.rc != 'success') {

            return null;

        }

        const output = result.data;

        /\* do what needs to be done based on this data \*/

    %>

    </click>

</icon>

<!-- transactions are defined as Xten properties -->

<txn name="my-transaction">

<%

    /\*

        transaction has following variables available

        txn: the name of the transaction

        attribs: attributes of the calling node

        data: data you passed from client side

        user: user details of the user or null if not logged in

        context: system generated context which is assigned to all pages

        the decisions must be taken based on these variables about what to do

    \*/

    /\*

        find the object key, it should be in attribs or in context

        this needs to be passed to database functions

    \*/

    const objKey = attribs['object-name'] || context['object-name'];

    const name = attribs.name; /\* name of the node \*/

    /\*

        function to get field value in the transaction

        database.inlineValue cannot be used in transaction

        because that requires a node as input

    \*/

    const value = await database.getFieldValue(objKey, name, attribs);

    /\* do your own processing as needed \*/

    /\* to save a field value \*/

    var result = await database.saveFieldValue(objKey, name, dataToSave, attribs);

    if (result.rc != 'success') {

        return result;

    }

    /\* to remove a field value \*/

    var result = await database.removeFieldValue(objKey, name, attribs);

    if (result.rc != 'success') {

        return {rc: 'Error Heading',

message: 'Some error message to be shown'};

    }

    /\*

        you do not need to pass any query parameters or where clause

        just pass the objKey, name and attribs you receive to the database

    \*/

    /\* return whatever data you need to return \*/

    const {rc: 'success', data: {

'key1': 'some data',

'some-other-key': 'other data'

}};

%>

</txn>

In this example a custom transaction was created. All transactions must return a Json object with ‘rc’ field being mandatory. If transaction is successful it should return {rc: ‘success’}. If there is an error then rc can be error string which would be displayed automatically on client side.

For getting the data in <load> function we used database.inlineValue, but that function requires the node to be passed. Since in the transaction we do not have the node we need to use different functions for data manipulation.

* database.saveFieldValue
* database.getFieldValue
* database.removeFieldValue

These functions decide based on objKey, name and context. The example of how to find these variables is given in above code. So even at this point you do not need to worry about which data fields to select and how. WX server does that on its own.

### To pass data from server to client through attributes

You can set the attribute value of the node in <init> or <load> function and access it on client side using element.getAttribute function.

<initElement>

<%

    var myData = element.getAttribute('my-attribute');

%>

</initElement>

<load>

<%

    node.attribs['my-attribute'] = 'my-value';

%>

</load>

In attribute you can only pass string data.

### Passing data from server to client through HTML

Technically you can create a <pre> tag and set it to ‘display:none’. You can put the whole Json object in the node text and retrieve it on client side and parse the Json. Another alternative is that you can create a <template> HTML tag and append it to your node. The <template> tag is not visible on client side. You can put the Json in its text field and retrieve it on client side.

<initElemnt>

<%

    var jsonData = element.innerHTML;

    const myData = JSON.parse(jsonData);

%>

</initElement>

<load>

<%

    const myData = {}; /\* a Json object \*/

    const template = new DomNode('template');

    template.text = JSON.stringify(myData);

    node.addChild(template);

%>

</load>

This way you can pass complex Json structures back to the client.

### Passing dynamically generated scripts or CSS

Normally based on data you need to do certain things. An example could be that based on some data you may want to change look and feel or based on data you may want to perform a different operation.

One way of doing this is to pass the data back and on client side put the logic for doing things differently, but in Wohola you have an even better alternative. You can pass the whole CSS or Javascript back to client.

Since Wohola X is only doing DOM manipulation and <style> and <script> tags are essentially elements within DOM so you can create these tags and the content inside it dynamically and add the resulting <script> or <style> to your node.

<init>

<%

    /\* get data from database \*/

    const nodeId = node.attribs.id;

    if (someCondition) {

        /\* we need client to have different CSS or JS \*/

        var mycss = `

                #${id} {background-color: blue;}

        `;

        var mycode = `

            const element = document.querySelector('#${id}');

            element.addEventListener('event', (event) => {

                /\* event listener code \*/

            })

        `;

        const style = new DomNode('style');

        style.text = mycss;

        node.insertChild(0, style); /\* add at the start of node \*/

        const script = new DomNode('script');

        script.text = mycode;

        node.addChild(script); /\* add at the end of the node \*/

    }

    else

    {

        /\* we need to pass something different \*/

    }

%>

</init>

When the page loads on the client side automatically the CSS and JS passed would be applied. In this case we are using #my-id as a selector so that the passed CSS and JS only applies to that element specifically. Since we are using an id as a selector its safe to do a querySelector on the document. Another reason is that at this stage the element variable which is available in the <initElement> is out of scope and there is no event that has occurred. So using id is an option in such a scenario.

The ids are automatically assigned and managed by Wohola so you don’t need to define them. You just need to find those ids from the node or element wherever needed.

The most important things to remember is that we are only generating code which would be run in the browser. The code is nothing but text. So you can do text manipulation to generate the code dynamically and as long as the output follows the correct syntax it would work.

## Handling events

After the page code has been created using the Xten properties and it is served to the browser. The browser then calls the <initElement> function and does whatever initialisation that needs to be done. After this the browser waits for events. The events can be user events based on which further actions are taken, or socket events coming from server or events generated by the elements within the page.

As these events occur further logic needs to be added to handle the relevant events.

### Handling user events

User events are like user typed something, clicked on something, hovered in or out of something, the focus was on some item or other such actions user takes. For handling these events these are the different techniques. These options are covered in the following example.

<!--

    different event handling options are given below

    only one of them must be used not all options together

-->

<xten name="my-form" element="form">

    <view>

        <label>My Data</label>

        <input type="text" />

        <!-- option 1: attach the event handler directly to the button -->

        <button name="my-button" type="submit">

            <span>Submit</span>

            <tooltip>Submit this form</tooltip>

            <click>

        </button>

    </view>

    <initElement>

    <%

        /\*

        option 2: submit event is sent to the on the form

        handle this event at form level using JS

        \*/

        element.addEventListener('submit', (event) => {

            event.preventDefault();

            /\* this stops from default click handler to be invoked \*/

            event.stopPropagation();

            /\* this stops from default submit handler to be invoked \*/

            const element = event.target;

            /\* validate form data \*/

            /\* process that data as per your logic \*/

        });

        /\*

        option 3: click event is sent to the form

        handle this event at button level using JS

        \*/

        const button = element.querySelector('button[name="my-button"]');

        button.addEventListener('submit', (event) => {

            event.preventDefault();

            const button = event.target;

            const element = button.closest('.wx-xten-my-form');

            /\* validate form data \*/

            /\* process that data as per your logic \*/

        });

    %>

    </initElement>

    <!-- option 4: attach the event handler to the xten -->

    <event name="submit">

    <%

        /\* no need to call stopPropagation and preventDefault \*/

        const button = event.target;

        const element = button.closest('.wx-xten-my-form');

        /\* validate form data \*/

        /\* process that data as per your logic \*/

    %>

    </event>

</xten>

If you need to attach the event to the main Xten element itself it can be done by using <click> or <event> properties in the Xten itself. If you need to attach the events to a child element you can add them directly in your WXML code as shown.

stopPropagation and preventDefault stop the default Javascript processing for that event from taking place. It is very important if you want to handle the event yourself otherwise after executing your code some default handler code would be executed which could cause problems in many cases. If you are using <event> or <click> Xten or properties then these are called automatically, but if you are using your own event handlers then you must call these as appropriate. It is safe to call both even if in that situation only one may be relevant.

The option to also add events using Javascript also exists. Standard HTML, CSS and Javascript are all part of WXML so these can be used where needed.

### Custom events

Sometimes the elements need to talk to each other. For example if one element needs to be updated if the data in the other element changes then we can use custom events. Any custom event can be defined and any element can listen to these events.

Let us assume that there is an input field. The requirement is that whenever that field changes its value should be shown in a different span element. To achieve this you can generate your own custom event with any name and pass any data to it.

You can send this event with element.notify function. This function is a Wohola function attached to element prototype through element.js file. So this function can be called on any element.

<xten name="xten-1">

    <view>

        <input type="text">

            <event name="change">

            <%

                /\*

                the event is attached directly to input

                so event.target is input itself

                \*/

                const value = event.target.value;

                element.notify('input-changed', value);

            %>

            </event>

        </input>

    </view>

</xten>

<xten name="xten-2">

    <initElement>

    <%

        /\* this event can be handled in the same way like standard events \*/

        element.addEventHandler('input-changed', (event) => {

            const value = event.detail;

/\* your data is available in detail \*/

            const span = element.querySelector('span');

            span.innerHTML = value;

        });

    %>

    </initElement>

    <view>

        <span name="automatically-updated"></span>

    </view>

</xten>

In this case we attached the event handler in Javascript but did not call preventDefault or stopPropagation because these are not standard events so Javascript does not have any default handler defined for those. The custom events can also be handled like standard events using <event> property or Xten.

### Socket messages

If you want to send a message from server to client then socket messages are the best way to do it. You can use socket.emit function on server to send a message and socket.on function to listen to that message on client side.

<initElement>

<%

    socket.on('some-data-updated', (data) => {

        /\* data contains the data passed by emit \*/

        /\* do whatever processing you need to do \*/

    });

%>

</initElement>

<txn name="update-some-data">

<%

    /\* code for updating data \*/

    socket.emit('some-data-updated', data);

%>

</txn>

You can pass any data to the socket and it is available to you in the callback function.

This way you server can send messages or data to the client and client can listen to them and act accordingly.

## Extending standard Javascript objects

The notify function described in previous example has been attached to the element prototype in element.js file in \_system/public/wx folder. This contains other Wohola functions that have been attached to the element prototype as well. This is an example of how you can extend the standard objects in Javascript.

Element.prototype.notify = function (message, data) {

    const event = new CustomEvent(message, {

    bubbles: true,

        detail: data

    });

    this.dispatchEvent(event);

};

This way you can also define your own functions and attach them to elements. These functions get the ‘this’ scope of the element.

The main thing to remember is that for this you cannot use arrow functions as they do not inherit the ‘this’ scope. So this would not work.

Element.prototype.notify = (message, data, target) => {

    const event = new CustomEvent(message, {

        bubbles: true,

          detail: data

     });

     this.dispatchEvent(event);

};

It is now a days considered ‘cool’ to use arrow functions everywhere but you need to be aware of places where they would not work.

These functions do not inherit the ‘this’ scope of the object so when you are trying to pass the scope of the parent class or object dynamically to the dynamically added methods like in this case these do not work.

In element.js file you can check out other helper functions like getId, getName, addClass, removeClass, toggleClass and hasClass. These are commonly required actions so have been defined as helper functions to make things easier.

The previous functions wxfns.addClass, wxfns.removeClass, wxfns.hasClass and wxfns.toggleClass are not deprecated and would be removed in future. They should not be used.

## Uploading files

To upload files there are steps to be followed.

1. Select the file from the local machine of the user
2. If this file needs to be shown in browser before saving it on server then do that
3. Upload the file to the browser and save it

An example of these steps is given here.

<xten name="my-upload-xten" element="div">

    <view>

        <img />

        <button>

            <span>select image</span>

            <click>

            <%

                /\*

                select file - it opens a file selector dialog

                the flag image/\* means all types of image files

it is a browser standard MIME type

                this flag can be list of file extensions separated by comma

                wxfns.selectFile('.jpg, .bmp, .png')

                or other MIME formats

                \*/

                const file = await wxfns.selectFile('image/\*');

                /\*

save this file in element so other code in Xten can use it

\*/

                element.selectedFile = file;

                /\*

                this file is in binary format at this time

                to display it in the placeholder without sending to server

                convert this binary blob into a url format

                \*/

                const url = URL.createObjectURL(file);

                /\* show this image in <im> tag of our view \*/

                const element = event.target.closest(

'.wx-xten-my-upload-xten');

                const img = element.querySelector('img');

                img.setAttribute('src', url);

                /\* now the image selected by user would be shown in img tag \*/

            %>

            </click>

        </button>

        <button>

            <span>upload image</span>

            <click>

            <%

                const element = event.target.closest(

'.wx-xten-my-upload-xten');

                /\*

                the file was in element.selectedFile

                save-image is a standard transaction

it links the image url to the node

                \*/

                const result = await element.upload('save-image', null,

element.selectedFile);

                if (result.rc != 'success') {

                    return;

                }

                /\* do something else \*/

            %>

            </click>

        </button>

    </view>

    <load>

    <%

        /\* get value of node data \*/

        const value = await database.inlineValue(node, context, xten);

        /\* put this value in src tag of img child of the node \*/

        const img = node.findChild('img');

        img.attribs.src = value;

    %>

    </load>

</xten>

URL.createObjectURL function is a Javascript function. Using this function a file in binary format can be converted into a url which can then be used in ‘src’ tag of <img> or <video> tag.

The input flag to the wxfns.selectFile is either one or more MIME types or file extensions separated by commas. In the file select dialog that user sees only those files which match the passed criteria would be visible.

In this case element.upload function is using a predefined ‘save-image’ transaction but if needed you can define your own transaction to handle the upload as well.

<xten name="my-upload-xten" element="div">

    <view>

        <img />

        <button>

            <span>select image</span>

            <click>

            <%

                const file = await wxfns.selectFile('image/\*');

                element.selectedFile = file;

                const url = URL.createObjectURL(file);

                const element = event.target.closest(

'.wx-xten-my-upload-xten');

                const img = element.querySelector('img');

                img.setAttribute('src', url);

            %>

            </click>

        </button>

        <button>

            <span>upload image</span>

            <click>

            <%

                const element = event.target.closest(

'.wx-xten-my-upload-xten');

                /\*

you can define your own transaction for upload

and pass custom data as well

\*/

                const result = await element.upload('my-transaction',

{myData1: 'some data'}, element.selectedFile);

                if (result.rc != 'success') {

                    return;

                }

                /\* the transaction has sent the new url in output so set it \*/

                const img = element.querySelector('img');

                img.setAttribute('src', result.output);

            %>

            </click>

        </button>

    </view>

    <load>

    <%

        const value = await database.inlineValue(node, context, xten);

        const img = node.findChild('img');

        img.attribs.src = value;

    %>

    </load>

    <txn name="my-transaction">

    <%

        /\*

        you get txn, attribs, data, user, context as arguments

        txn is name of the transaction

        attribs are attributes of your node

        user is user details or null if user is not logged in

        \*/

        /\*

        first do validations of values being recieved

        if there is an error return the error

        \*/

        if (!attribs.name) {

            return { rc: 'File Upload Error',

message: 'missing name in txn: ' + txn };

        }

        /\* get the objKey just like transaction \*/

        const objKey = attribs['object-name'] || context.objKey;

        /\* the file data is available in context.file \*/

        const file = context.file;

        /\*

        path where the file was uploaded to on server

        it is a file in \_tmp folder in this case

        \*/

        const tempFileName = file.path;

        /\* original name of the file \*/

        const fileName = file.originalname;

        /\*

        figure out where to store the file and generate url and path

        the files that are to be made available through URL on client side

        should be in public/uploads folder of the site

        the url for files in public/uploads folder should start with uploads/

        so the file path would be

context.siteRoot + 'public/uploads/' + filename

        and the url for this file would be uploads/filename

        the files that are to be kept private and

client cannot open them with url

        should be in uploads folder of the site

        so the filepath would be context.siteRoot + 'uploads/' + filename

        and the url would be null or not set in the database

        the special characters and spaces are to be replaced with '\_' as well

        \*/

        const list = fileName.split('.');

        const ext = list.pop().toLowerCase();

        const newfilename = list.join('.')

.toLowerCase().replace(/[^a-z0-9]+/g, '-')

            .replace(/^[\-]+/, '').replace(/[\-]+$/, '')

+ common.uuid() + '.' + ext;

        const newFilePath = context.siteRoot + 'public/uploads/images/' +

newfilename;

        const url = 'uploads/resized/' + newfilename;

        const resizedPath = context.siteRoot + 'public/uploads/resized/' +

newfilename;

        /\*

copy the file from \_tmp folder to final destination folder

calculated above

\*/

        var result = files.rename(tempFileName, newFilePath);

        if (result.rc != 'success') {

            return result;

        }

        files.makedir(context.siteRoot + 'public/uploads/resized');

        /\*

in this case we also resize the file if the uploaded file is too big

\*/

        if (data && data.width && ext != 'svg') {

            await Jimp.read(newFilePath)

                .then(image => {

                    if (image.bitmap.width > data.width \* 1.2) {

                        return image.resize(data.width \* 1.2, Jimp.AUTO)

                            .write(resizedPath);

                    } else {

                        files.copy(newFilePath, resizedPath);

                        return Promise.resolve();

                    }

                })

                .catch(err => {

                    console.error(err);

                });

        }

        else

{

            files.copy(newFilePath, resizedPath);

        }

        /\* save the value \*/

        const previous = await database.getFieldValue(objKey, attribs.name,

attribs);

        result = await database.saveFieldValue(objKey, attribs.name, url,

attribs);

        if (result.rc != 'success') {

            return result;

        }

        /\*

        remove previous file if there was one

        otherwise users can keep uploading files and old files would stay

        so junk would keep gathering on server

        \*/

        if (previous) {

            files.remove(context.siteRoot + 'public/' + previous);

        }

        /\* send the url of the file in output \*/

        return { rc: 'success', output: url };

    },

    %>

    </txn>

</xten>

To point to note is that you either upload file to the server or run a transaction on the server. element.upload and element.transaction work in an identical manner with only one difference, in element.upload the file is also sent in the context.file and this needs to be handled apart from the other data being sent.

## Streaming files

Normally when you call a transaction you get all of the output data at once. This would not work in case of video or audio files. If the video is 1 hour long it could take a very long time and a very large amount of memory to wait until the whole video file is sent back before playing the video. In such cases the video needs to be streamed, where the server keeps sending video file in chunks and the broser keeps playing those chunks as and when they are recieved.

To stream the files instead of using the regular URL we prefix the URL with ‘stream/’ to let the server know that this file needs to be streamed. An example can be found in quickplay Xten.

<load>

<%

    const value = await database.inlineValue(node, context, xten);

    if (value) {

        node.src = value;

        const video = node.findChildren('video')[0];

        /\*

        instead of setting src tag directly with the file url

        we put stream/ before the url

        now wohola server would stream this file

        and video can be played soon after first chunk is received

        \*/

        video.attribs.src = 'stream/' + value;

    }

%>

</load>

The same thing would apply to audio files as well as any large files.

Important thing to note is that if for video files <video> tag has capability to handle streams in built but if you are trying to get other types of large files from browser to client then you need your own mechanism to handle chunks of data. This is a very rare requirement so is not explained in this section.

## Calling external APIs

### APIs that are called through some functions

Some APIs require you to call the functions directly from given Javascript file. To call these APIs simply include the Javascript file in the site folder, load it using <script> tag in your HTML. This can be done with <libs> property of the Xten. This would add the <script> tag for these files in the HTML. Then as per documentation of these libraries you can initialize them and call the functions in your Javascript code.

If the files are located in ‘public/libs’ folder in your site folder then you can include them in your Xten through url ‘libs/’. If its external link you can put the whole link starting from https://.

<xten name="textedit" element="div">

    <libs>

libs/some-js-lib.js, libs/some-css.css,

https://bigcompany.com/some-library.js

</libs>

    <!-- other Xten properties -->

</xten>

After these libraries are loaded you can follow the documentation of these libraries and in your <initElement> or event handlers you can call the functions from these libraries as per their documentation.

### Calling REST APIs

To call external URLs whether in REST format or any custom format you can use common.curl function on server side and wxfns.curl function from client side. The example is as given.

<xten name="call-api">

    <initElement>

    <%

        const response = wxfns.curl(

'https://external.com/something:3000?abc=123&hello=world',

'POST');

        if (response.success) {

            const data = response.data;

            /\* do what you need to do with this data \*/

        }

        else

        {

            /\* do what you need to in case of error \*/

        }

    %>

    </initElement>

    <txn name="txn">

    <%

        const inputData = {}; /\* some input needed by API \*/

        /\* external apis can be called in <init> <load> or <txn> \*/

        const response = wxfns.curl(

            'https://external.com/something:3000?abc=123&hello=world',

            'POST',

            JSON.stringify(inputData),

            /\*

            json is default so no need to pass the content type

            in case of json. this is just shown as an example

            \*/

            'application/json'

        );

        if (response.success) {

            const data = response.data;

            /\* do what you need to do with this data \*/

        }

        else

        {

            /\* do what you need to in case of error \*/

        }

    %>

    </txn>

</xten>

So you can call the external APIs on demand where necessary in your logic and get the data from them.

The contentType is set to default contentType for your data. In case of json data this is set as default so contentType arugment can be ommitted.

You get the data back in response.data object if response was successful, otherwise you get the error in response.error.

## Reloading, redirecting the page or refreshing the Xten

* For reloading the page on client side use standard Javascript function window.location.reload().
* For redirecting the page on client side use standard Javascript window.location.href = ‘new-url’;
* For redirecting the page on server side use context.res.redirect(‘new-url’) in the server side code including <init>, <load> and <txn>.
* If you only want to refresh the Xten after some data has been updated then use element.refresh().

<icon name="add">

    <click>

    <%

        /\*

        let us say you ran a transaction which changed the data in database

        now you need to refresh your client side view for that Xten

        so you can call refresh()

        \*/

        element.refresh();

    %>

    </click>

</icon>

## Overriding Xtens or default system files

In Wohola X all the system files, including Xten files, icons and default images, can be overriden by the WXML developer. WX first looks for the included file in the site folder and if it finds it there that files is used, but if that file is missing then it serves that file from \_system folder.

This is a very powerful feature as it allows the WXML programmer to override files based on project needs without affecting other projects or affecting system files. So the core functionality could stay the same while based on project requirements some particular file(s) could be different in each project.

If for example there is a requirement that for a particular project the delete icon needs to be changed. For this simply find the the file path of that icon file in \_system folder and replicate the same filepath in your site folder with same file name. Now you can put a different SVG file here while keeping the same name and this file would be used in your project. The core system or the other projects would continue to use the default icon and would not be affected.

The same strategy can be used for overriding Xtens as well. If a particular Xten does not meet the project requirement then you can create the same folder structure and file name for the Xten in your site folder. You can keep the name of the Xten the same as well. Now you can simply change the code in this Xten as per your project requirements. The core system and other projects would be unaffected and would continue to use the old Xten.

## Extending Xtens

Wohola X also uses inheritance mechanism like classes in other programming languages. So if you need a custom Xten which has some properties which are different from the default Xten then you can extend the default Xten.

The Xten can be extended by using extend=”base-xten-name” as Xten attribute. It is similar to a class inheriting from base class.

<xten name="base-xten" element="\_hidden">

    <style>position: relative</style>

    <cssNode>

        <!-- base xten cssNode -->

    </cssNode>

    <css>

        <!-- base xten css -->

    </css>

    <init>

    <%

        /\* base xten init code \*/

    %>

    </init>

    <load>

    <%

        /\* base xten init code \*/

    %>

    </load>

    <toolbar>

        <!-- let us assume a toolbar with 2 buttons -->

    </toolbar>

</xten>

<xten name="extended-xten" extend="base-xten" element="div">

    <allow>admin</allow>

    <mandatory>aspect-ratio</mandatory>

    <css enhance="append">

        <!-- additional css to be appended after the \_list <css> -->

    </css>

    <cssNode>

        <!-- replace cssNode entirely -->

    </cssNode>

    <view>

        <!-- some view -->

    </view>

    <initElement>

    <%

        /\* initElement code \*/

    %>

    </initElement>

    <init>

    <%

        /\* init code \*/

    %>

    </init>

    <load enhance="prepend">

    <%

        /\* additional code to prepend before the \_list <load> code \*/

    %>

    </load>

    <toobar enhance="append">

        <!-- add 1 more button to \_list toolbar -->

        <!-- now toolbar would have 3 buttons -->

    </toolbar>

</xten>

In this example we have defined an Xten named ‘base-xten’. Another Xten named ‘extended-xten’ extends it. base-xten has some properties while extended-xten also has some properties. This works like normal inheritance in any language that supports class inheritance.

So mainly the properties from the extended Xten are used, but if there is a property that extended Xten does not have but any properties that are there in base-xten but missing in extended-xten are also available within extended-xten.

There is one additional feature in WX. Normally when you override a property you override the whole property. In WX you can enhance the base property by either appending or prepending to it. So in the example above <toolbar enhance=”append”> does not replace the <toolbar> defined in base-xten, but instead it keeps that toolbar and any button defined in extended-xten toolbar are appended to the base-xten toolbar.

Similarly the <load> code written in extended-xten would be prepended to the <load> code of the base-xten.

The enhance can be used for CSS related properties, views, toolbars and properties that have Javascript code.

If you are not sure what effect emhance may have then you can simply override the whole property, e.g. by copying the property from base to extended and modify it. In this case any property of the base-xten that does not need to be changed can be omitted from the extended-xten so it gets included from base-xten itself.

## Summary

In this document so far we have covered how to create Xtens and how to perform the necessary operations like:

1. How to manage HTML and CSS in the Xten
2. How to manipulate HTML through <init> and <load> property
3. How to send data back and forth between client and server
4. How to save and retrieve data from database
5. How to handle user events
6. How to upload files
7. How to stream media files
8. Call external APIs
9. Use inbuilt role based security and access control
10. Roloading, refreshing and redirecting
11. Extending and overriding Xtens and system files

These cover the basic principles of Xten programming. There would further documents which would cover the advanced programming concepts as well as reference manuals for various different functions.