**Polymer concepts**

[Notify and reflect to attribute](https://stackoverflow.com/questions/43331359/polymer-2-0-notify-and-reflect-to-attribute)

**Notify:**

can be set to True|False. Let's say you have parent-element and child-element. Working example

parent-element.html:

<dom-module id="parent-element">

<template>

<child-element foo="{{test}}"></child-element>

</template>

<script>

Polymer({

is: "parent-element",

properties: {

test: {

value: "bar"

}

}

})

</script>

</dom-module>

As you can see, we have 1 property called test which is propagated to child element, where we can manipulate with it.

child-element.html:

<dom-module id="child-element">

<template>

<paper-input value="{{test}}"></paper-input>

</template>

<script>

Polymer({

is: 'child-element',

properties: {

test: {

}

},

});

</script>

</dom-module>

What is hapenning? In child element we defined test property and this property is binded to paper-input, which means, whenever we write something in paper-input, the property updates itself automatically . YEE that's awesome, we don't need to take care of updating property inside child-element, but HOW can parent know that property test has changed? Well, he can't.

And here comes notify: true. If you set it up, you don't have to care about notifying parent-element that somewhere inside of the child-element, test property has been changed.

Shortly, property test in parent-element and child-element will updates simultaneously

**Reflect-to-attribute:**

As name already says, when you set this to some property, it's value will be visible in attribute of host element. Better to show this on some example.

In Polymer we know, that setting some binding to attribute of some element needs $ sign.

<custom-elem foo-attribute$="[[someProperty]]"></custom-elem>

Well, this can't be used everywhere. Let's say, that we need foo-attribute inside custom-elem.

Becuase foo-attribute was declared as attribute and not property, it's value won't be visible inside of element. So we need something where some attribute will represent attribute and also property.

So some working example, with some real situation would be like:

<dom-module id='parent-element'>

<template>

<style>

child-elemet[foo='bar'] {background-color: green}

child-element[foo='foo'] {background-color: red}

</style>

<child-element foo="{{test}}"></child-element>

</template>

<script>

Polymer({

is: "parent-element",

properties: {

test: {

value: "bar"

}

}

})

</script>

</dom-module>

In this case, CSS won't work, because foo is property (not an attribute) and CSS is applied on attributes only.

To make it work, we have to add reflectToAttribute on foo property inside child-element.

<dom-module id='child-element'>

<template>

<div>[[foo]]</div>

</template>

<script>

Polymer({

is: "child-element",

properties: {

foo: {

reflectToAttribute: true

}

}

})

</script>

</dom-module>

After this, foo will become attribute and also property. At this moment, CSS will be applied on child-element and we will be able to work with value of foo inside child-element.

**Styling in css**

* In Polymer.js, the host element refers to the element to which shadow DOM is attached.
* In Polymer.js to style the host, use the :host selector.

**Example:**

:host {  
    text-align:center;  
    color:white;  
    font-family:'Roboto', 'Noto', sans-serif;  
    }

* we can use CSS selectors to decide when and how to style the host element

For Example,consider the following code in my-element.js :

........

static get template() {

return html`

<!-- Encapsulated, element-level stylesheet -->

<style>

:host { font-family: sans-serif; color:black }

:host(.green) {color: green;}

:host(.red) {color: red;}

:host(.blue) {color: blue;}

:host(:hover) {color: green;}

:host(.warning) p { color: yellow;}

</style>

<p> Hi, I am paragraph from component!! </p>

<p>Make this text yellow if component has class "warning", and black otherwise.</p>

`

}

}

.........

Here,

1):host  - The selector :host matches any <my-element> element  
2):host(.green)  - This selector matches <my-element> elements of green  
3):host(.red) -  This selector matches <my-element> elements of red  
4):host(.blue) - This selector matches <my-element> elements of blue  
5):host(:hover) - This selector matches <my-element> elements when they are hovered over

in index.html we can use like this:

<html>

<head>

........

</head>

<body style="text-align:center">

<my-element></my-element>

<my-element class="red"></my-element>

<my-element class="blue"></my-element>

<my-element class="green"></my-element>

</body>

</html>

* Descendant selectors after :host match elements in the shadow tree

consider following code in my-element.js :

..........

static get template() {

return html`

<!-- Encapsulated, element-level stylesheet -->

<style>

:host(.warning) p { color: yellow;}

</style>

<p>Make this text yellow if component has class "warning", and black otherwise.</p>

`

}

}

..........

Here,the CSS selector applies to any <p> element in the shadow tree if the host has class called "warning"

**in index.html:**

<html>

<head>

........

</head>

<body>

<my-element class="warning"></my-element>

<my-element></my-element>

</body>

</html>

Style slotted content

* The <slot> element is a part of the Web Components technology suite and inside a web component, we can see <slot> placeholder where you can fill with your own markup, which allows you to create separate DOM trees and show them together.
* You can create slots in an element's template that are filled at runtime.
* We can use the ::slotted() syntax for styling slotted content
* By default, if an element has shadow DOM, the shadow tree is rendered instead of the element's children. To allow children to render, you can add an <slot> element to your shadow tree.

**Syntax:**

1. <slot></slot>
2. (OR)
3. <slot name="title"></slot>

* A named slot only accepts top-level children that have a matching slot attribute. Note that only top-level children can match a slot.

Consider following Example:

**my-element.js:**

1. .........
2. static get template() {
3. return html`
4. <style>
5. /\* Selects any element placed inside a slot \*/
6. p ::slotted(\*){
7. font-family: sans-serif;
8. color:red;
9. }
10. /\* Selects any element placed inside a slot \*/
11. h1 ::slotted(\*) {
12. font-family: sans-serif;
13. color:yellow;
14. }
15. /\* Selects any <h1> placed inside a slot \*/
16. h1 ::slotted(h1) {
17. font-family: sans-serif;
18. color: green;
19. }
20. /\* Selects any <p> placed inside a slot \*/
21. p ::slotted(p) {
22. font-family: sans-serif;
23. color: blue;
24. }
25. /\* Selects any paraStyle class placed inside a slot \*/
26. p ::slotted(.paraStyle) {
27. font-family: sans-serif;
28. color: pink;
29. }
30. /\* Selects any slot whose value equal to para1 \*/
31. p ::slotted([slot=para1]) {
32. color:brown;
33. }
34. </style>
35. <h1><slot name="title"></slot></h1>
36. <p><slot name="para"></slot></p>
37. <p><slot name="para1"></slot></p>
39. `
40. }
41. }
42. ..........

**index.html:**

1. <html>
2. <head>
3. .....
4. </head>
5. <body>
6. <my-element>
7. <div slot="title"><u>Hi I am div of title in yellow color</div>
8. </my-element>
9. <br>
10. <my-element>
11. <div slot="para">Hi I am span of para</span>
12. </my-element>
14. <br>
15. <span slot='title'>Hi I am span of title</span>
16. <br>
17. <span slot='para'>Hi I am span of para</span>
18. <my-element>
19. <h1 slot="title">Heading 1. I'm green.</h1>
20. </my-element>
21. <br>
22. <my-element>
23. <p slot="para">Paragraph text. I'm blue.</p>
24. </my-element>
26. <br>
27. <my-element>
28. <div slot="para" class="paraStyle">Hi I am div of para</span>
29. </my-element>
30. <my-element>
31. <div slot="para1">Hi I am div of para1</span>
32. </my-element>
33. </body>
34. </html>

In my-element.js, we are using ::slotted(\*) to select all slotted content,::slotted(h1) and ::slotted(p) to select by element type, ::slotted(.paraStyle) to select by class and ::slotted([slot=para1]) to select by slot name.

In index.html, we can use these slots to apply style for the content.

**Css Selector**

<https://www.w3schools.com/cssref/css_selectors.asp>

## Combinators[Section](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Selectors#Combinators)

[**Adjacent sibling combinator**](https://developer.mozilla.org/en-US/docs/Web/CSS/Adjacent_sibling_combinator)

The + combinator selects adjacent siblings. This means that the second element directly follows the first, and both share the same parent.  
**Syntax:** A + B  
**Example:** h2 + p will match all [<p>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/p) elements that directly follow an [<h2>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/h2).

[**General sibling combinator**](https://developer.mozilla.org/en-US/docs/Web/CSS/General_sibling_combinator)

The ~ combinator selects siblings. This means that the second element follows the first (though not necessarily immediately), and both share the same parent.  
**Syntax:** A ~ B  
**Example:** p ~ span will match all [<span>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/span) elements that follow a [<p>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/p), immediately or not.

[**Child combinator**](https://developer.mozilla.org/en-US/docs/Web/CSS/Child_combinator)

The > combinator selects nodes that are direct children of the first element.  
**Syntax:** A > B  
**Example:** ul > li will match all [<li>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/li) elements that are nested directly inside a [<ul>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/ul)element.

[**Descendant combinator**](https://developer.mozilla.org/en-US/docs/Web/CSS/Descendant_combinator)

The   (space) combinator selects nodes that are descendants of the first element.  
**Syntax:** A B  
**Example:** div span will match all [<span>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/span) elements that are inside a [<div>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/div) element.

[**Column combinator**](https://developer.mozilla.org/en-US/docs/Web/CSS/Column_combinator)

The || combinator selects nodes which belong to a column.  
**Syntax:** A || B  
**Example:** col || td will match all [<td>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/td) elements that belong to the scope of the [<col>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/col).

**Note :- Any E > F pair is necessarily also an E F pair; the former is a proper subset of the latter.**

Just think of what the words "child" and "descendant" mean in English:

* My daughter is both my child and my descendant
* My granddaughter is not my child, but she is my descendant.

## CSS [attribute~="value"] Selector

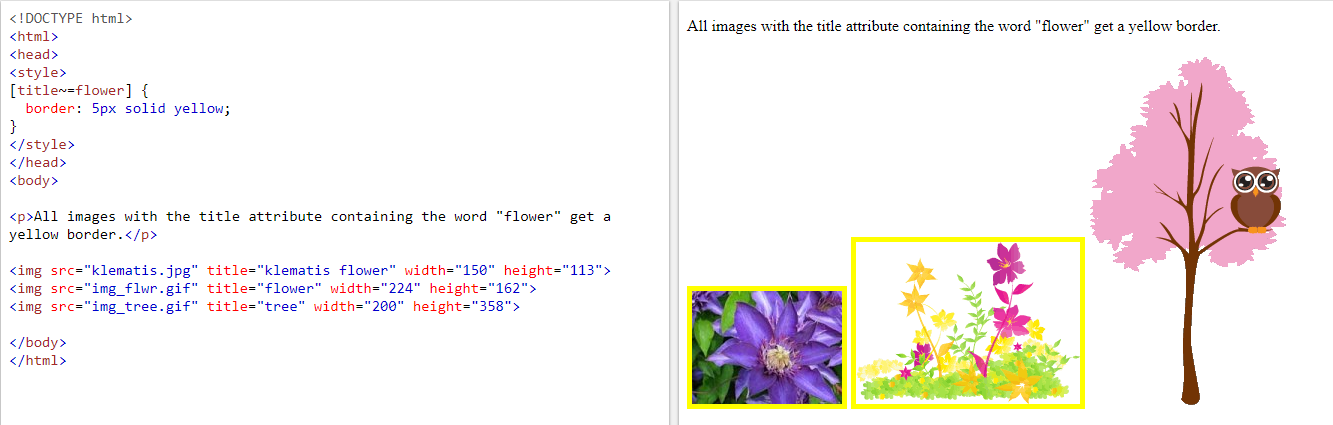
The [attribute~="value"] selector is used to select elements with an attribute value containing a specified word .Represents elements with an attribute name of *attribute* whose value is a whitespace-separated list of words, one of which is exactly value.

The following example selects all elements with a title attribute that contains a space-separated list of words, one of which is "flower":

### Example

[title~="flower"] {  
  border: 5px solid yellow;  
}

The example above will match elements with title="flower", title="summer flower", and title="flower new", but not title="my-flower" or title="flowers".



## CSS [attribute|="value"] Selector

The [attribute|="value"] selector is used to select elements with the specified attribute starting with the specified value.

The following example selects all elements with a class attribute value that begins with "top":

**Note:** The value has to be a whole word, either alone, like class="top", or followed by a hyphen( - ), like class="top-text"!

### Example

[class|="top"] {  
  background: yellow;  
}



## CSS [attribute^="value"] Selector

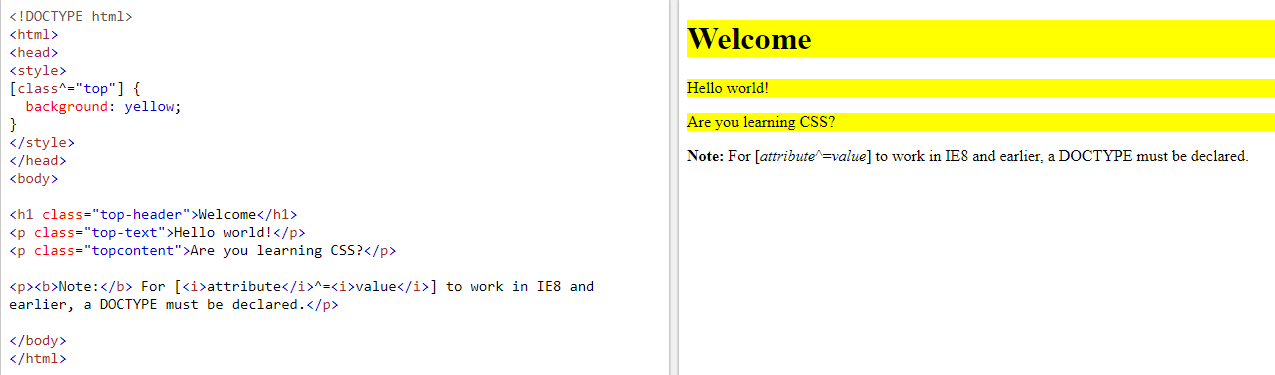
The [attribute^="value"] selector is used to select elements whose attribute value begins with a specified value.

The following example selects all elements with a class attribute value that begins with "top":

**Note:** The value does not have to be a whole word!

### Example

[class^="top"] {  
  background: yellow;  
}



## CSS [attribute$="value"] Selector

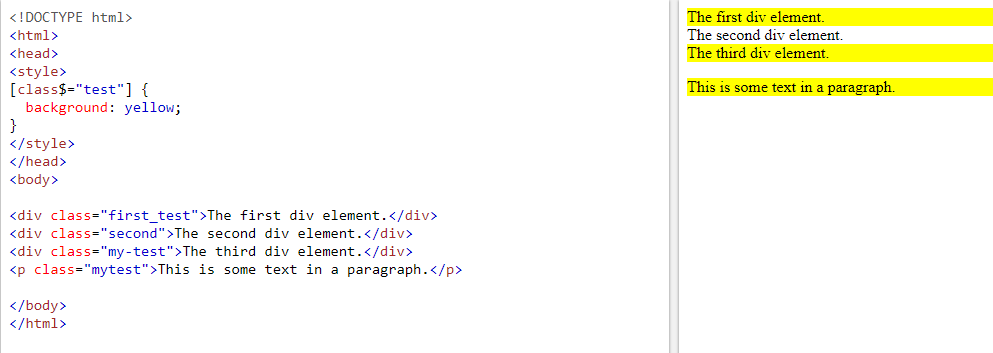
The [attribute$="value"] selector is used to select elements whose attribute value ends with a specified value.

The following example selects all elements with a class attribute value that ends with "test":

**Note:** The value does not have to be a whole word!

### Example

[class$="test"] {  
  background: yellow;  
}



## CSS [attribute\*="value"] Selector

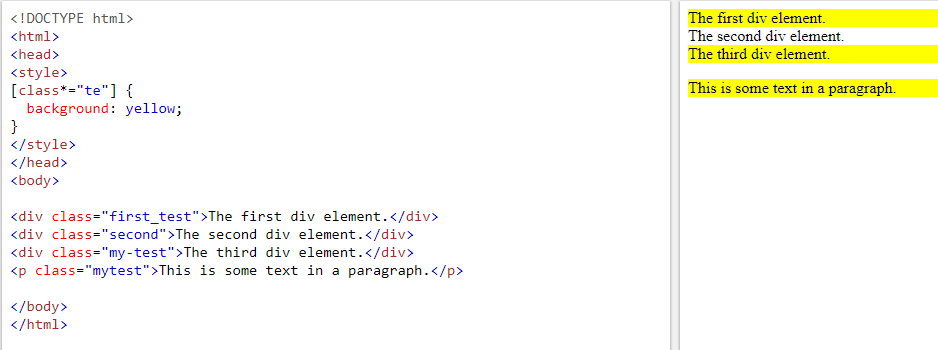
The [attribute\*="value"] selector is used to select elements whose attribute value contains a specified value.

The following example selects all elements with a class attribute value that contains "te":

**Note:** The value does not have to be a whole word!

### Example

[class\*="te"] {  
  background: yellow;  
}



The CSS **attribute selector** matches elements based on the presence or value of a given attribute.

**[*attr* *operator* *value* i]**

Adding an i (or I) before the closing bracket causes the value to be compared case-insensitively (for characters within the ASCII range).

**[*attr* *operator* *value* s]**

Adding an s (or S) before the closing bracket causes the value to be compared case-sensitively (for characters within the ASCII range).

#### CSS

a {

color: blue;

}

/\* Internal links, beginning with "#" \*/

a[href^="#"] {

background-color: gold;

}

/\* Links with "example" anywhere in the URL \*/

a[href\*="example"] {

background-color: silver;

}

/\* Links with "insensitive" anywhere in the URL,

regardless of capitalization \*/

a[href\*="insensitive" i] {

color: cyan;

}

/\* Links with "cAsE" anywhere in the URL,

with matching capitalization \*/

a[href\*="cAsE" s] {

color: pink;

}

/\* Links that end in ".org" \*/

a[href$=".org"] {

color: red;

}

Shared Styling

In polymer.js we can share the styles using style modules. We can package up styles in a style module, and share them between elements.

To create a style module we can follow these steps:

1)Create a <dom-module> element using JavaScript:

1. const styleElement = document.createElement('dom-module');

2)The <dom-module> element's innerHTML property should contain a <template> element and  wraps a <style> block inside <template> element:

1. styleElement.innerHTML =
2. `<template>
3. <style>
4. /\* Your shared styles go here \*/
5. </style>
6. </template>`;

3)Register your newly created style module as an element:

1. styleElement.register('shared-style');

4)You can package the style module in its own JavaScript file, then the element that uses the styles will need to import that file.

1. import './shared-style.js';

5)When you create the element that will use the styles, include the style module in the opening tag of the style block:

1. <style include="shared-style">

Shared styling – Demo

**Highlights:**

* To understand Shared Styling Demo

**Steps:**

1. Write the below code index.html file

1. <!doctype html>
2. <html>
3. <head>
4. <script type="module" src="src/my-element/my-element.js"></script>
5. </head>
6. <body style="text-align:center">
7. <my-element></my-element>
8. </body>
9. </html>

2. Write the below code in the my-colors.js file of the my-element folder of src folder

1. import {html} from '@polymer/polymer/polymer-element.js';
2. export const styleElement = document.createElement('dom-module');
3. styleElement.innerHTML =
4. `<template>
5. <style>
6. p.red {
7. color: red;
8. }
9. p.green {
10. color: green;
11. }
12. p.blue {
13. color: yellow;
14. }
15. </style>
16. </template>`;
17. styleElement.register('shared-style');

3. Write the below code in the my-element.js file of the my-element folder of src folder

1. import {html, PolymerElement} from '@polymer/polymer/polymer-element.js';
2. import {styleElement} from './my-colors'
3. class MyElement extends PolymerElement {
4. static get template() {
5. return html`
6. <style include="shared-style"></style>
8. <p class="red">I wanna be red</p>
9. <p class="green">I wanna be green</p>
10. <p class="blue">I wanna be blue</p>
11. `
12. }
13. }
14. customElements.define('my-element', MyElement);

4. Run the application and observe the output

5. Now modify the my-colors.js file

1. import {html} from '@polymer/polymer/polymer-element.js';
2. export const sharedStyles = html`
3. <style>
4. p.red {
5. color: red;
6. }
7. p.green {
8. color: green;
9. }
10. p.blue {
11. color: blue;
12. }
13. </style>
14. `;

6. Now modify my-element.js file also

1. import {sharedStyles} from './my-colors'
2. class MyElement extends PolymerElement {
3. static get template() {
4. return html`
5. ${sharedStyles}
6. <p class="red">I wanna be red</p>
7. <p class="green">I wanna be green</p>
8. <p class="blue">I wanna be blue</p>
9. `
10. }
11. }
12. customElements.define('my-element', MyElement);

5. Now run the application and observe the output

Same output

Styling Polymer.js elements using Bootstrap

npm install --save bootstrap

<link rel="stylesheet" href="./node\_modules/bootstrap/dist/css/bootstrap.min.css">

Install bootstrap through CDN link

<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css">

**IRON AJAX**

<iron-ajax

auto url = "https://www.googleapis.com/youtube/v3/search"

params = '{"part":"snippet", "q":"polymer", "key": "YOUTUBE\_API\_KEY", "type": "video"}'

handle-as = "json"

on-response = "handleResponse"

debounce-duration = "500">

</iron-ajax>

**params** − An attribute where you can pass the JSON with the request parameters.

debounce-duration-

Routing

* Single page application is currently favored heavily due to its advantages over traditional multi-page applications
* Polymer allows us to create a single page application(SPA) with the help of routing
* A router is used to connect a component to a path and to switch pages according to the path set in the URL
* Polymer provides additional tools to create a single page application, namely – app-location and app-route.

Now we can explore important elements that support Routing.

**app-location:**

app-location is an element that provides synchronization between the browser location bar and the state of an app. It is used to capture the URL and send it to the first app-route on the page.

**Syntax:**

1. <app-location route="{{route}}" use-hash-as-path></app-location>

where the route is the object where the URL get stored

**app-route:**

app-route is an element that enables declarative, self-describing routing for a web app. It receives the URL from app-location and is used to capture the path from it.

**Syntax:**

1. <app-route route="{{route}}"
2. pattern="/:ext"
3. data="{{routeData}}"
4. tail="{{subroute}}"></app-route>

where,

route - the object containing the URL, sent by app-location

pattern - app-route compares the URL with this pattern and  the path gets stored in the variable name given in pattern after ":"

data - the object into which the path data obtained gets stored

tail - object with path data that needs to get passed to the next object

**iron-pages:**

* <iron-pages> is an element that is used to switch between the pages of the router
* It takes in an attribute and compares it with the names gives to the components to select the appropriate page

**Syntax:**

1. <iron-pages attr-for-selected=“routeData.ext" selected="{{name}}">
2. <index-page name="index" route="{{subRoute}}"></index-page>
3. ............
4. </iron-pages>

* The iron-pages component tries to match the data present in routeData.ext and name to decide which page to show.
* It takes in an attribute and compares it with the names gives to the components to select the appropriate page

To create a Single Page Application, first, we need to build a router.

To build a router we need to follow below-given steps:

* In node.js command prompt we need to run these commands:

1. npm install @polymer/app-route
2. npm install @polymer/iron-pages

* create a fresh component to house your router
* attach this to the main index page
* import app-route, iron-pages and the individual components/pages of your application into the router component
* include app-location and immediately followed by app-route
* add iron-pages component and the components as child nodes
* make sure that the location where the path is stored and the attribute-as-selected are the same

Now, we can perform navigation between pages.

Navigating between pages can be done in two ways:

**1.Using anchor tags:**

Unlike other frameworks, you can directly use the anchor tag to navigate  to your page

Provide the appropriate path in href=" "

**Syntax:**

1. <a href=" "#/path>Component</a>

**2. Updating the route path:**

You can also navigate using functions

By updating the route object with the new path, we can programmatically navigate between pages

Provide app-location to the component, so the route object can capture the path

Use the syntax in the function to change the path

1. this.set('route.path','/path')

**Router Params:**

* Much like props, data can be passed to routes using params. However, they have certain limitations and aren't as powerful as props
* For one, we pass data through the path and thus will be exposed in the URL

**Procedure to obtain params:**

* Within your router component, pass the tail object as a route into the appropriate iron-pages component
* In the child component, import and insert yet another app-route object
* Since the route is already being passed from the router component, there is no need for an additional app-location in the child component
* Follow the same procedure as before to store and access the params