Build your first plain JavaScript app

May 26, 2022 • Aaron Collier • 13 min read • JavaScript

This tutorial guide you through building a simple blog using plain JavaScript with all of the content stored in Kontent by Kentico. You won't need to install any libraries or packages, use any frameworks, or run any servers. You just include the Kontent Delivery SDK to get all your content when you need it.

See a live example of the blog you'll be making or check out the source code on GitHub.

> **New to Kontent?**
>
> If you're starting with Kontent, we suggest you begin with the Hello World tutorial, in which you'll learn the basics of creating content.

Requirements

You'll need a text editor or an IDE like Visual Studio Code. The JavaScript used works on most modern browsers.

The blog initially pulls content from a shared Kontent project. If you want to connect the blog to your own Kontent project to edit the content, you'll need a subscription along with a Sample Project.

Create an article list

1. Create your index

The first thing to do for the blog is to create a list of articles. Start by creating the basic shell for your app in a new file called `index.html`.

The page includes a script in the `<head>` to make sure the Delivery JS SDK works. Then there's two more scripts in the `<body>` to make the list itself. You'll create these scripts later.

- The `core.js` script will contain code you'll use in multiple places in the app.
- The `articleList.js` script will be for code that is unique to this list.

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width,initial-scale=1" />
  <!-- Include the Delivery JS SDK  -->
  <script type="text/javascript"
    src="https://cdn.jsdelivr.net/npm/@kentico/kontent-
delivery@11.0.0/dist/bundles/kontent-delivery.umd.js"></script>
  <title>Kontent JavaScript sample app</title>
```

```
  <link href="style.css" rel="stylesheet" />
</head>

<body>
  <div id="app-header">
    <h1><a href="index.html">Articles</a><a class="github-link" target="_blank"
        href="https://github.com/Kentico/kontent-tutorial-plain-javascript"><img
src="img/gh.png"></a></h1>
  </div>
  <div id="app"></div>

  <!-- Include core functions -- code you'll use in multiple places in the app -->
  <script type="text/javascript" src="core.js"></script>

  <!-- Include script for the list of articles -- code that is unique to this list -->
  <script src="articleList.js"></script>
</body>
</html>
```

If you'd like your list to appear as the example, you can use the same styles. Create a file called `style.css`.

CSS

```
@import url('http://fonts.cdnfonts.com/css/gt-walsheim');

body {
  font-family: 'GT Walsheim', sans-serif;
  /* padding: 0 20%; */
  margin: 0;
  padding: 0 0 48px 0;
  background-color: #f1f5f9;
}

body a {
  color: rgb(219, 60, 0);
}

#app {
  margin-left: auto;
  margin-right: auto;
}

@media screen and (min-width: 700px) {
  #app {
    width: 90%;
  }
}

@media screen and (min-width: 1200px) {
  #app {
    width: 60%;
```

```
  }
}

#app-header {
  margin-bottom: 24px;
  background-color: rgb(219, 60, 0);
  padding: 12px 24px;
  color: #ffffff;
  line-height: 50px;
}

#app-header .github-link img {
  width: 50px;
  height: 50px;
  float: right;
}

#app-header a {
  text-decoration: none;
  color: inherit;
  border-radius: 25px;
}

#article-list {
  display: flex;
  flex-wrap: wrap;
  gap: 24px;
}

.card {
  padding: 24px 32px;
  border: solid 1px #e6e6e6;
  border-radius: 25px;
  background: white;
}

.card-no-link-style a {
  color: inherit;
  text-decoration: none;
}

.card:hover {
  box-shadow: #cfcfcf 0 0 1rem;
}

.card img {
  width: 100%;
}

@media screen and (min-width: 700px) {
```

```
  .card {
    flex: 1 1 calc((100% / 2));
  }
}

@media screen and (min-width: 1200px) {
  .card {
    flex: 1 1 calc((100% / 3));
  }
}

@media screen and (min-width: 1900px) {
  .card {
    flex: 1 1 calc((100% / 4));
  }
}

.article-teaser {
  width: 100%;
}
```

2. Create a list container

To hold your list, create a list container within the app. Start by creating a file in the same directory called `core.js` and add two constants that can be used throughout your app.

- The `app` constant finds the app container in the DOM.
- The `addToElementById` constant is a function that can be used to create a new element with a specific ID in the DOM and then attach it as the child of another element.

JavaScript

```
// Define main container
const app = document.getElementById('app');

// Function for creating and appending elements
const addToElementbyId = (elementType, id, parent) => {
  const element = document.createElement(elementType);
  element.setAttribute('id', id);
  parent.appendChild(element);
  return element;
};
```

If the function seems too abstract, you can see how it works in practice.

Create another file in the same directory called `articleList.js` and add one constant named `articleList`.

This takes the function you added in core.js and uses it to create a `<div>` element within the app container with an ID of `article-list`. If you now load index.html in a browser and inspect the DOM, you'll be able to see your new element inside the app.

JavaScript

```
// Add list container to app
const articleList = addToElementbyId('div', 'article-list', app);
```

That's a great start! Now let's get some actual content.

3. Get content from Kontent

To get your content, first you need to define in what Kontent project that content is in `core.js`.

JavaScript

```
// Set up Kontent Delivery client
const Kk = window['kontentDelivery'];
const deliveryClient = new Kk.createDeliveryClient({
  projectId: '975bf280-fd91-488c-994c-2f04416e5ee3'
});
```

That's just defining an instance of a delivery client for a specific project using the Kontent Delivery SDK that you can use anywhere in the app.

Now you can use `deliveryClient` inside `articleList.js`.

JavaScript

```
// Call for a list of all articles
deliveryClient
  .items()
  .type('article')
  .toPromise()
  .then(response => {
    console.log(response)
  });
```

You've retrieved all the articles from the project and logged them in the console. To get an idea of what kind of content you're getting, you can explore the object in the browser console (the list of content is in the `items` collection).

So now you've got a page to hold your list and the content that should be listed. It's time to put them together.

4. Add content to your list

To put the content into the list, first create a useful helper function in `core.js`.

This function creates an element in the DOM with a defined class. Then if you include an attribute and a value to look for, it adds those to the element based on the type.

JavaScript

```
// Function for adding elements to DOM with specific attributes
const createElement = (elementType, classToAdd, attribute, attributeValue) => {
    const element = document.createElement(elementType);
```

```
    element.setAttribute('class', classToAdd);

    // Set attribute value based on the attribute required
    attribute === 'href'
      ? (element.href = attributeValue)
      : attribute === 'innerHTML'
      ? (element.innerHTML = attributeValue)
      : attribute === 'innerText'
      ? (element.innerText = attributeValue)
      : attribute === 'src'
      ? (element.src = attributeValue)
      : undefined;

    return element;
  };
```

To see the function in use, change your call for the delivery client in `articleList.js` as follows.

JavaScript

```
deliveryClient
  .items()
  .type('article')
  .toPromise()
  .then(response => {
    response.data.items.forEach(item => {
      // Create nodes
      const card = createElement('div', 'card');
      card.classList.add('card-no-link-style');
      const link = createElement(
        'a',
        'link',
        'href',
        './article.html#' + item.elements.url_pattern.value
      );

      // Add nodes to DOM
      articleList.appendChild(card);
      card.appendChild(link);
    });
  });
```

You're processing each of the items. Inside the `forEach`, you can see two examples calling the function you added to `core.js`. The first creates a `<div>` element with a class of `card` to hold each article. The second creates an `<a>` element with a class of `link` and an `href` attribute set to a page with a hash you're getting from the response (read more about the URL pattern value if you're interested – you'll use the value again for building an article page). The card is then added to the `article-list` container and the link to the card.

If you open `index.html` now, you'll see six rectangles with links inside.

Let's add the rest of the content to your list.

JavaScript

```
deliveryClient
  .items()
  .type('article')
  .toPromise()
  .then(response => {
    response.data.items.forEach(item => {
      // Create nodes
      const card = createElement('div', 'card');
      card.classList.add('card-no-link-style');
      const link = createElement(
        'a',
        'link',
        'href',
        './article.html#' + item.elements.url_pattern.value
      );

      const teaser = createElement(
        'img',
        'article-teaser',
        'src',
        item.elements.teaser_image.value && item.elements.teaser_image.value.length
          ? item.elements.teaser_image.value[0].url + '?w=500&h=500'
          : undefined
      );
      const title = createElement(
        'h2',
        'article-title',
        'innerText',
        item.elements.title.value
      );
      const description = createElement(
        'div',
        'article-description',
        'innerHTML',
        item.elements.summary.value
      );

      // Add nodes to DOM
      articleList.appendChild(card);
      card.appendChild(link);
      link.append(teaser, title, description);
    });
  });
```

You've created a teaser image, title, and short description and added them all to your link. If you open `index.html`, you should see the same list as in the live example.

Now you know how to create a simple list of items from Kontent and display them with JavaScript. To see the other options available, it's time to create the articles themselves.

Create an article page

1. Create a page for articles

To display individual articles, create a file called `article.html`.

This is basically the same as `index.html` (`core.js` is still included), but instead of `articleList.js` you add `article.js`. Except that file doesn't exist yet.

HTML

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width,initial-scale=1" />

    <!-- Include the Delivery JS SDK -->
    <script
      type="text/javascript"
      src="https://cdn.jsdelivr.net/npm/@kentico/kontent-delivery@11.0.0/dist/bundles/kontent-delivery.umd.js"
    ></script>

    <title>Article</title>

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

  <body>
    <div id="app-header">
      <h1><a href="index.html">Articles</a><a class="github-link" target="_blank" href="https://github.com/Kentico/kontent-tutorial-plain-javascript"><img src="img/gh.png"></a></h1>
    </div>
    <div id="app"></div>

    <!-- Include core functions -->
    <script type="text/javascript" src="core.js"></script>

    <!-- Include script for the article -->
    <script src="article.js"></script>
  </body>
</html>
```

2. Create the basic article structure

Create another file called `article.js`. The code is very similar to what you had in `articleList.js`.

- The `articleSlug` constant defines the URL pattern of the article, that is the value you added to the link in the list.
- In `articleContainer`, you again add a container to the app.
- In the `deliveryClient` call, you add an `equalsFilter` that ensures you get only the article that matches the hash in the current URL. You should get an object with only one item in it.
- In the response, you check if the hash matches an article (to prevent an exception from an empty array). Then you use the `createElement` function from `core.js` to create a header image, title, and body for your article and lastly attach them to the article container.

JavaScript

```
// Define which article is being retrieved
const articleSlug = location.hash.slice(1);

// Create article container
const articleContainer = addToElementbyId('div', 'article', app);

// Call for article info
deliveryClient
  .items()
  .type('article')
  .equalsFilter('elements.url_pattern', articleSlug)
  .toPromise()
  .then(response => {
    // Check if article found before adding
    const article =
      response.data.items && response.data.items.length ? response.data.items[0] :
undefined;

    // Update title
    document.title = `Article | ${article.system.name}`;

    // Create nodes
    const headerImage = createElement(
      'img',
      'article-header',
      'src',
      article.elements.teaser_image.value[0].url
    );
    const title = createElement(
      'h2',
      'article-title',
      'innerText',
      article.elements.title.value
    );
    const richTextElement = article.elements.body_copy;
```

```
    const rteResolver = Kk.createRichTextHtmlResolver().resolveRichText({
      element: richTextElement,
      // Here you'll define your resolvers
    });
    const body = createElement(
      'div',
      'article-description',
      'innerHTML',
      rteResolver.html
    );

    // Add nodes to DOM
    articleContainer.append(headerImage, title, body);
    return;
  });
```

If you load `index.html`, you should now be able to click on the articles in the list and see them display the same as in the live example. You can use the blog header to go back to the list and see all of the articles.

Now you've got a basic blog with a listing and article view that really works. You can stop there if you like or continue on and add in a couple more useful features.

Resolve links to content items

One of the great features of Kontent is the ability to create links within content that don't rely on any front-end logic. So you don't have to hard code specific URLs into your content, but rather add a link to a content item.

Because these links are independent of front-end logic, if you leave them alone, they'll resolve as `<a>` elements with empty `href` attributes, as you can see by opening `article.html#coffee-beverages-explained` and inspecting the link.

HTML

```
<a data-item-id="3120ec15-a4a2-47ec-8ccd-c85ac8ac5ba5" href="">so rich as today</a>
```

1. Define a resolver function

To get these links to resolve exactly as you'd like them, you can take advantage of the SDK's resolvers. In the `rteResolver` in `article.js`, define a function to handle hypertext links.

JavaScript

```
const rteResolver = Kk.createRichTextHtmlResolver().resolveRichText({
  element: richTextElement,
  linkedItems: Kk.linkedItemsHelper.convertLinkedItemsToArray(response.data.linkedItems),
  urlResolver: (linkId, linkText, link) => {
    // Set link based on type
    const urlLocation =
      link.type === 'article'
        ? `article.html#${link.urlSlug}`
```

```
        : link.type === 'coffee'
          ? `coffee.html#${link.urlSlug}`
          : 'unsupported-link';
    return { linkUrl: urlLocation };
  },
});
```

Here, you're defining different paths based on the type of link that's being resolved. If there's a link to an article, it will use a path like the one you defined for your article list.

2. Add the function to your call

In `article.js`, add in a `queryConfig` with a `urlSlugResolver`.

JavaScript

```
deliveryClient
  .items()
  .type('article')
  .equalsFilter('elements.url_pattern', articleSlug)
  .queryConfig({
    urlSlugResolver: (link, context) => {
      return resolveUrl(link);
    },
  })
  .toPromise()
  // Continue as above
```

The code implements the URL slug resolver for this specific query, as you can see if you refresh `article.html#coffee-beverages-explained` and inspect the link again.

3. Handle routing

To make the link work when you click on it, add the following code to `article.js` to refresh the page (and get the new article) whenever the hash changes.

JavaScript

```
// Reload page on hash change
const renderHash = () => {
  window.history.go();
};
window.addEventListener('hashchange', renderHash, false);
```

Now your app should be able to handle links between different content items and always display the article you want. To get the full advantage of your structured content, you'll want to make sure your app can handle added structure.

Resolve linked items and components

The body of each article is a rich text element. One of the benefits of structuring your content with Kontent is you can have clearly predictable structure without having to know exactly what the content will be (read more in our article on dealing with structure in rich text).

For these articles, that means being able to add things like embedded tweets and videos to the page. You can see an example of tweets in the mentioned article, so here just make sure videos are embedded correctly.

Before you do anything, your embedded video will appear as an empty `<p>` element, as you can see by opening `article.html#coffee-beverages-explained`.

HTML

```
<p class="kc-linked-item-wrapper"></p>
```

1. Define a resolver function

To fill in this element with an embedded video, in `rteResolver` in `article.js` define a function to handle resolving.

This function starts by checking if the item you're trying to resolve is a video. Then it checks if a specific host has been selected. It then returns embed code based on the specified hosting provider. If no host has been selected or if the item is not a hosted video, the function returns an empty string for inside the paragraph.

JavaScript

```
const rteResolver = Kk.createRichTextHtmlResolver().resolveRichText({
  element: richTextElement,
  linkedItems: Kk.linkedItemsHelper.convertLinkedItemsToArray(response.data.linkedItems),
  urlResolver: (linkId, linkText, link) => {
    // Your link resolution logic
  },
  contentItemResolver: (itemId, item) => {
    if (item.system.type === 'hosted_video') {
      const videoID = item.elements.video_id.value;

      // Check if a video host exists
      const videoHost =
        item.elements.video_host.value && item.elements.video_host.value.length
          ? item.elements.video_host.value[0].codename
          : undefined;
      if (videoHost) {
        // Return based on hosting provider
        const htmlCode = videoHost === 'youtube'
          ? `<iframe src='https://www.youtube.com/embed/${videoID}' width='560'
height='315' frameborder='0'></iframe>`
          : `<iframe src='https://player.vimeo.com/video/${videoID}' width='560'
height='315' allowfullscreen frameborder='0'></iframe>`;
        return {
```

```
            contentItemHtml: htmlCode
          };
        }
      }
      return {
        contentItemHtml: ''
      };
    }
  });
```

2. Add the function to your call

You also need to include this function in your `queryConfig` in `article.js`.

JavaScript

```
deliveryClient
  .items()
  .type('article')
  .equalsFilter('elements.url_pattern', articleSlug)
  .queryConfig({
    urlSlugResolver: (link, context) => {
      return resolveUrl(link);
    },
    richTextResolver: (item, context) => {
      return resolveLinkedItems(item);
    }
  })
  .toPromise()
  // Continue as above
```

Now if you open `article.html#coffee-beverages-explained`, you'll see an embedded YouTube video about halfway down the page. There's still an empty `<p>` element for a tweet, if you'd like to take on an additional challenge.

Your app should have all the basic blog functionalities when everything is working well. You might want to add in some assistance for when things don't go as planned.

Handle errors

Although you can hope everything will run smoothly, from time to time we all encounter errors. So it'd be good to add some basic error handling to your app.

1. Define error messages

The first thing to do is add a report handler function to `core.js` so you can use it in both your list and individual articles.

JavaScript

```
// Error messages
const reportErrors = err => {
  console.error(err);
  app.innerHTML = `<p>An error occured ☹:</p><p><i>${err}</i></p>`;
};
```

2. Add error handling to call

For your article list, you're not calling for a specific article so you just need to include handling of general errors, which you can do at the end of the code in `articleList.js`.

JavaScript

```
deliveryClient
  .items()
  .type('article')
  .toPromise()
  .then(response => {
    // Your current code
  })
  .catch(err => {
    reportErrors(err);
  });
```

Any errors will now show up in the in the browser window and also in the console as error objects.

For `article.js`, you'll want to add the same general error handling, but also a response if a specific article isn't found (such as if someone enters a random string after the hash).

JavaScript

```
deliveryClient
  .items()
  .type('article')
  .equalsFilter('elements.url_pattern', articleSlug)
  .toPromise()
  .then(response => {
    // Check if article found before adding
    const article =
      response.data.items && response.data.items.length ? response.data.items[0] :
undefined;

    // 404 message if not found
    if (!article) {
      app.innerHTML = '<p>That article could not be found.</p>';
      return;
    }
  })
  .catch(err => {

```

```
    reportErrors(err);
});
```

And that's it. Now you have your basic app up and running.

What's next?

Explore our boilerplates and tools:

- Express.js Boilerplate with Apollo server with a built-in GraphQL support.
- Model generator to leverage strongly typed models in your code.
- Nuxt.js module for Vue.js to create statically generated apps and sites.
- React boilerplate to render data from Kontent using React components.
- Test HTTP service to mock request responses from the Delivery SDK.

We also recommend you try our sample apps for Express, React, Vue, and more.