Getting Started With Headless CMS

WRITTEN BY SERGE HUBER
CTO AT JAHIA

CONTENTS

What is a CMS?

A content management system (CMS) is a software application that facilitates the creation, management (editing versioning, workflow, etc.) and publishing of content. Historically speaking, content management systems were invented so that non-technical content creators could quickly collaborate to produce and publish content without worrying about the technological complexities of displaying it in a web page or application. Today, CMS technology is just as important to developers as it is to non-technical content managers. The line between what is and what isn't code has become more blurred than ever before, with most applications containing content today. Thus, developers are looking for solutions that allow them to craft and deploy content-enabled apps, websites, and other digital experiences quickly, without having to reinvent the wheel.

Headless CMS in a Nutshell

Headless CMS is a content management system that allows you to manage and access content from your applications using an API. Unlike traditional CMS solutions, headless CMS operates without the presentation layer (the "head," or frontend) that would dictate how the content should be displayed. Instead, you control the presentation with your own code.

This not only enables a content-first approach when engaging your audience (as content creators no longer have to wait for development teams to catch up), but also means you can use the same content across multiple channels — websites, mobile apps, digital assistants, virtual reality, smart watches, etc. — making the headless CMS the ideal solution for a fast-paced, multichannel world.

Traditional (Coupled) vs. Headless vs. Hybrid CMS (Decoupled)

When choosing a new CMS, it's important to understand the differences between various architectural approaches that different

jahia

Stack-up & Stand Out

Unite Content, Data, and Applications into
Your Stack to deliver a better customer experience.

- One hub for your content and customer data
- Modernize legacy applications to deliver personalized online experiences
- Flexible and continuous cross-channel management
- Open and modular to fit your needs
- Built to integrate with your best-in-class Martech stack

Visit us at www.jahia.com
Contact us at marketing@jahia.com

GENEVA | PARIS | LONDON | BOSTON | TORONTO

products use. While this may look like a technical detail, it has a big impact on how the CMS will support your business goals now and in the future.

COUPLED/MONOLITHIC

In a coupled system, the underlying store for your content serves both authoring and delivery, and the process of making content live is typically a matter of setting a flag in the database.

HEADLESS

Headless CMS technology provides content to the presentation tier (or content consumer) as a service, typically via a RESTful interface in JSON or XML format. This is known as Content as a Service (CaaS).

A headless CMS can either be coupled or decoupled. Further, any CMS worth its salt today (decoupled or not) must support headless/CaaS-based content delivery.

DECOUPLED/HEAD OPTIONAL

A decoupled system puts authoring and delivery in separate applications and potentially on separate infrastructure.

In a decoupled system, the process of making content live is completed through a publishing mechanism through which content is pushed from the authoring platform (and underlying content repository) to a content delivery infrastructure.

Advantages and Disadvantages of Headless CMSs

PROS

- The API makes the content available through any channel and on any device and allows you to include the CMS as part of your microservices architecture.
- You can write your websites or mobile applications using any programming language, your favorite tools, and your own development process.
- You have full control over the application lifecycle without having to interfere with any CMS code.
- It provides higher security and much easier scalability.

CONS

- A pure headless CMS doesn't provide channel-specific support (especially for the web channel), which means developers may need to develop some web-specific functionality themselves.
- Marketers may be limited in what they can do with a pure headless CMS and rely more on developers for tasks like creating a landing page with a custom layout.

The Importance of Separating Code From Content

Developers have a strong handle on how to manage and deploy code assets. Yet, at some point in our application build, we've all said, "What about this text? What about these images? Where do these belong?" That's pretty universal. Nearly every single application today has content in it. Be it a web app or a native app, it's full of strings, images, icons, media, and other classes of content.

This content doesn't really belong in our code base because it's not code. These non-code assets make us as developers pretty uneasy. We know that at some point, a business user will ask us to change one of those strings and we'll spend hours going through build and deploy cycles to handle a 30-second code change. We know that at some point, we'll need to translate that content. We know that at some point, we'll have to replace this UI with another one. We know all of these things — and we know leaving that content, even if it's abstracted into a string table or a resource bundle, will come back to haunt us. No matter the abstraction, it's part of the build. Developers must update it. Developers and systems folks must deploy it.

Smart developers separate code from content. They make sure that the content in the application is completely independent of the build and deploy cycle of the application itself. Where appropriate, they make sure non-technical business users can access and update the externalized content and publish changes at any time.

Can I Use Any Web CMS to Supply Content to My Apps?

Maybe. A lot depends on the app's needs and the CMS. If the app is web-based — or if the app is a native app and the CMS supports APIs/headless mode — then the answer is yes.

But there's a catch. In a few words, the ability to build the app against a CMS alone is not sufficient. Technology choices like programming language matter. Architecture matters. Making the right choices up front can save countless man hours and money spent on rework later.

TECHNOLOGY

Choose a CMS that aligns architecturally and ideally, one that aligns technology-wise with your existing technology investment and skills.

ARCHITECTURE

CMS technologies based on an RDBMS database and JCR repositories rely on clustering and replication. For content delivery, these systems rely on clustering and replication to handle high throughput.

For high transaction throughput applications, make sure you have a plan to properly scale the backend or consider a CMS with a "sharding" content delivery architecture.

Decoupled CMS platforms are more likely to support a shared nothing topology.

PURPOSE

Many CMS technologies were created to manage web pages. Make sure the CMS is either designed to manage the content you intend to input or that it's appropriately content agnostic enough to meet your needs.

Headless CMS Project Examples

[Video] Creating React Single Page Application with React – This is a live coding session that was presented at Jahia Days 2018. View the recording: https://www.youtube.com/watch?v=6fbkH-EIkcs

[Video] Jahia Progressive Web Application Demo – Jahia makes headless content management simpler. See how in a quick demo: https://youtu.be/3ax7P5MLqmc

Putting a Headless CMS Into Practice With Jahia

Jahia is an open-source, Java-based Digital Experience Platform, which helps companies make the digital world simpler and accessible for everyone. In this section, we'll walk you through the steps for building a single page application (SPA), progressive web app (PWA) from scratch using React and GraphQL.

Note: This tutorial assumes you have a Jahia installation up and running. If you don't, we suggest you follow the First Steps With Jahia tutorial.

This Refcard shows you how to create a JavaScript application from scratch using React and GraphQL with Jahia as a headless content backend. The application displays a news list that Jahia's headless content management UI manages: Content and Media Manager.

This Refcard also shows you how to create a standalone application that could be deployed in any website. If you're interested in building a JavaScript application hosted in a Jahia module and deployed on a Jahia server, we recommend you use a different starting point. We provide a starter module project on GitHub here: https://github.com/Jahia/dx-react-starter

After completing the tutorial, you'll know how to:

- Create a React application.
- Use Content and Media Manager to add sample content to your app.
- Use GraphiQL to execute GraphQL queries and browser schema.
- Set up a basic Apollo Client to perform GraphQL queries to Jahia.
- Set up basic authorization.
- Style the app's UI using React Material.
- Create a component that retrieves news objects from Jahia.
- Use GraphQL field aliases to improve property retrieval.

REQUIREMENTS AND TECHNOLOGIES USED

You will need the following tools for the tutorial:

- Node.js and npm
- Yarn
- DX 7.3 or Jahia 7.3.2 with the Content and Media Manager module installed

**Follow the First Steps With Jahia tutorial if you haven't done this yet.*

The tutorial uses the following technologies:

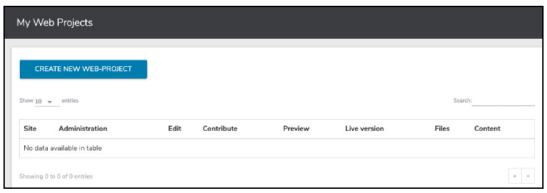
- Yarn
- Content and Media Manager
- React.js
- React Material
- GraphQL
- Apollo GraphQL client
- GraphiQL
- Jahia Security Filter

CREATING A WEB PROJECT IN JAHIA

Web projects are virtual web sites that you can edit in Jahia. One single Jahia server can host several web projects for different teams and can handle separate domain names if needed.

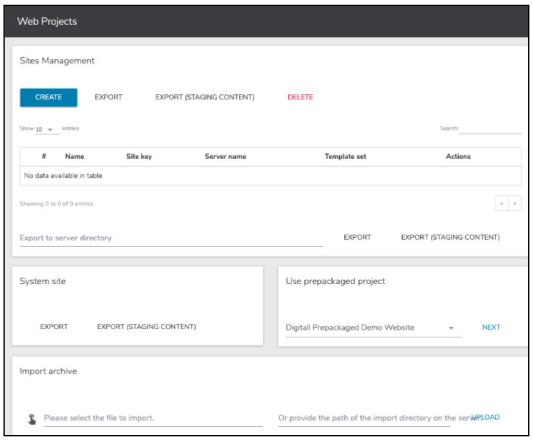
To create a web project:

1. In your dashboard, click **My Web Projects** in the left menu.



Then click **Create New Web Project** in the main pane.

2. Create a project from a prepackaged site in the **Use prepackaged project** pane. Then select a project and click **Next**.



3. Set the properties for your new web project and choose modules if needed in the dialogs that follow. This imports a prepackaged site with existing data to help you follow the tutorial. On average, an import takes 2-3 minutes. In the meantime, you can set up the front-end part of the project.

CREATING AND LAUNCHING THE STANDALONE JAVASCRIPT APP

First, use `yarn create` to generate a skeleton React application by executing the following commands on the command line:

```
yarn create react-app my-app
cd my-app
yarn start
```

Then you'll see the URL to which you can connect using your browser to view the compiled application. Then you can perform changes, which are immediately visible in your browser, offering fast compile-deploy cycles.

Note: If some updates don't work properly, just stop the server and restart it using `yarn start`*. That should fix most issues.*

REACT MATERIAL

React Material styles the application's look and feel using Google's Material design components. This library provides an out-of-the-box React component library that makes building applications with a consistent look and feel much easier and faster.

To add React Material to the project:

1. Execute the following command on the command line at the root of the project:

```
yarn add @material-ui/core
```

2. Update the media viewport property so it's compatible with mobile devices:

```
<meta
        name="viewport"
        content="minimum-scale=1, initial-scale=1,
        width=device-width, shrink-to-fit=no"
/>
```

3. To add an `AppBar` in `App.js`, first add the imports at the top of the file:

```
import AppBar from '@material-ui/core/AppBar';
import Toolbar from '@material-ui/core/Toolbar';
import Typography from '@material-ui/core/Typography';
import CssBaseline
    from '@material-ui/core/CssBaseline';
import Grid from "@material-ui/core/Grid/Grid";
import Paper from "@material-ui/core/Paper/Paper";
```

4. Then replace the App class component generated by the `create-react-app tool`:

```
class App extends Component {
    render() {
        return (
            <div className="App">
            <header className="App-header">
                <img src={logo} className="App-logo"
                    alt="logo"/>
                <p>
                    Edit <code>src/App.js</code>
                    and save to reload.
                </p>
                <a
                    className="App-link"
                    href="https://reactjs.org"
                    target="_blank"
                    rel="noopener noreferrer"
                >
                    Learn React
                </a>
            </header>
        </div>
    );
    }
}
export default App;
```

With:

```
const App = () => {
    return (
        <React.Fragment>
            <CssBaseline/>
            <AppBar position="static" color="default">
```

Code continued on next page

```
                <Toolbar>
                    <Typography variant="title"
                        color="inherit">
                        Companies
                    </Typography>
                </Toolbar>
            </AppBar>
            <Grid container>
                <Grid item xs={12}>
                    <Paper>
                        {/*CompanyList placeholder*/}
                    </Paper>
                </Grid>
            </Grid>
        </React.Fragment>
    );
};
```

*Note: This tutorial uses React **functional** components over **class** components (as demonstrated above).*

The `AppBar` displays at the top of the page with the **Companies** title to prepare the layout for the `CompanyList` component.

PROJECT STRUCTURE

Before continuing, create the following folder structure in your project:

```
|____src
| |____components
| | |____Company
| | |____CompanyList
```

COMPONENT STRUCTURE

Following React best practices, this tutorial assumes that you create each component in a folder that identifies the component. For example, the `CompanyList` component is created under `src/components/CompanyList`. By doing this, you can take advantage of using an `index.js` file to export your component as the default.

CREATING THE COMPANY COMPONENT

Next, create the `Company` component that will be rendered in the `CompanyList`.

1. Create a `Company.jsx` file under `src/components/Company` and add the following content:

```
import React from 'react';
import {withStyles} from "@material-ui/core";
import CardMedia from "@material-ui/core/CardMedia";
import CardContent from "@material-ui/core/ CardContent";
import Card from "@material-ui/core/Card";
import Typography from "@material-ui/core/Typography";
const styles = {
  card: {
    maxWidth: 300,
    maxHeight: 350,
  },
```

Code continued on next column

```
  media: {
    height: '120px',
  },
};
const Company = ({
  classes,
  title,
  description,
  image,
}) => {
  return (
    <Card className={classes.card}>
      <CardMedia
        className={classes.media}
        image={image}
        title="Company"
      />
      <CardContent>
        <Typography component="h1" variant="display1">
          Company Name
        </Typography>
        <br />
        <Typography component="p">
          {description.length > 150
            ? `${description.substr(0, 100)}...`
            : description}
        </Typography>
      </CardContent>
    </Card>
  );
};
export default withStyles(styles)(Company);
```

2. Export the 'Company' component as default by creating an 'index.js' file:

```
import Company from './Company';
export default Company;
```

CREATING THE COMPANY LIST COMPONENT

Next, create a `CompanyList` component to retrieve the list of company objects from the Jahia server.

To create the `CompanyList` component:

1. Create a `CompanyList.jsx` file in `src/components/CompanyList` with the following content:

```
import React from 'react';
import {withStyles} from '@material-ui/core';
import GridList from '@material-ui/core/GridList';
import GridListTile from '@material-ui/core/GridListTile';
import Company from "../Company";
const styles = theme => ({
  root: {
    display: 'flex',
    flexWrap: 'wrap',
    justifyContent: 'space-around',
    overflow: 'hidden',
```

Code continued on next page

```
    backgroundColor: theme.palette.background.paper,
  },
  gridList: {
    paddingTop: '12px',
    width: 1020,
    height: 660,
  },
});
const CompanyList = ({classes}) => {
  const title = 'Company name';
  const description =
    'Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Vivamus a tortor hendrerit, dapibus libero eu,
tincidunt nisl. Sed leo turpis, rutrum id condimentum quis,
consequat eget enim. Nunc a tempor dui, eget tristique
mi. Nunc ut ultrices sem, vitae posuere erat. Nulla
sollicitudin blandit nunc, vel scelerisque orci vehicula eu.
Nam sit amet sapien lectus.';
  const image = 'http://via.placeholder.com/300x120';
  return (
    <div className={classes.root}>
      <GridList
        className={classes.gridList}
        justify="center"
        cellHeight={300}
        cols={3}
        spacing={32}
      >
        <GridListTile>
          <Company
            title={title}
            description={description}
            image={image}
          />
        </GridListTile>
      </GridList>
    </div>
  );
};
export default withStyles(styles)(CompanyList);
```

Note: Ensure you save the file.

2. Create an `index.js` file in the same folder with the following contents:

```
import CompanyList from './CompanyList';
export default CompanyList;
```

3. In the `App.js` file, add the import for the new `CompanyList` component at the top of the file:

```
import CompanyList from './CompanyList';
```

4. Then replace:

```
</AppBar>
</React.Fragment>
```

With:

```
</AppBar>
    <Grid container>
        <Grid item xs={12}>
            <Paper> <CompanyList/> </Paper>
        </Grid>
    </Grid>
</React.Fragment>
```

BUILDING QUERIES WITH GRAPHIQL

Next, use GraphiQL to explore Jahia GraphQL queries before integrating them in your app. This example shows how to execute queries on the nodesByQuery query field.

To build queries with GraphiQL:

1. Open GraphiQL by navigating to the Jahia tools at http://localhost:8080/tools. Then select **GraphiQL** at the lower left.

*Note: You can use the **Documentation Explorer** to explore the Jahia GraphQL API. Open the explorer by selecting **Docs** in the upper-right corner.*

2. Execute the following query on the `nodesByQuery` query field:

```
{
  jcr(workspace: LIVE) {
    nodesByQuery(
      query: "SELECT * FROM [jdnt:company] as results
              WHERE ISDESCENDANTNODE(results,
                  '/sites/ digitall/')"
      queryLanguage: SQL2
    ) {
      nodes {
        uuid
        name
      }
    }
  }
}
```

Note: You are using an SQL2 query to retrieve the data. If you receive a login exception, make sure you are properly logged into the CMS first.

3. Next, add properties to the query:

```
{
  jcr(workspace: LIVE) {
    nodesByQuery(
      query: "SELECT * FROM [jdnt:company] as results
        WHERE ISDESCENDANTNODE(results,
            '/sites/ digitall/')"
      queryLanguage: SQL2
    ) {
      nodes {
        uuid
        name
        properties {
          name
```

Code continued on next page

```
          value
        }
      }
    }
  }
}
```

4. The internationalization (i18n) properties are not returned unless you specify a language with which to retrieve them. For example, modify the query to look like this:

```
{
   jcr(workspace: LIVE) {
      nodesByQuery(
         query: "SELECT * FROM [jdnt:company] as results
          WHERE ISDESCENDANTNODE(results,
              '/sites/ digitall/')"
         queryLanguage: SQL2
      ) {
         nodes {
            uuid
            name
            properties(language: "en") {
               name
               value
            }
         }
      }
   }
}
```

5. Notice that the query generates too much data. Modify the query to only retrieve the properties that you need:

```
{
   jcr(workspace: LIVE) {
      nodesByQuery(
         query: "SELECT * FROM [jdnt:company] as results
                WHERE ISDESCENDANTNODE(results,
                    '/sites/digitall/')"
         queryLanguage: SQL2
      ) {
         nodes {
            uuid
            name
            title: displayName(language: "en")
            description: property(name: "overview",
                    language: "en") {
                        value
                    }
                    thumbnail: property(name: "thumbnail",
                        language: "en") {
                        url: refNode {
                            path
                        }
                    }
                }
            }
        }
```

Code continued on next column

```
}
```

CONNECTING TO GRAPHQL USING APOLLO CLIENT

Next, add the Apollo GraphQL client library so you can connect to Jahia's GraphQL API.

To add the Apollo GraphQL client library:

1. From the root of the project, execute the following commands on the command line:

```
yarn add react-apollo
yarn add apollo-cache-inmemory
yarn add apollo-client
yarn add apollo-client-preset
yarn add apollo-link-rest
yarn add graphql
```

2. In `App.js` file add:

```
//...
import {ApolloProvider} from 'react-apollo';
import {ApolloClient} from 'apollo-client';
import {HttpLink} from 'apollo-link-http';
import {InMemoryCache} from 'apollo-cache-inmemory';
const JWTDXToken = 'JWT_DX_TOKEN';
const httpLink = new HttpLink({
  uri: 'http://localhost:8080/modules/graphql',
  headers: {
    Authorization: `Bearer ${JWTDXToken}`
  }
});
const client = new ApolloClient({
  link: httpLink,
  cache: new InMemoryCache()
});
```

3. Then modify the render body to the following:

```
(<React.Fragment>
  <ApolloProvider client={client}
    <CssBaseline />
    <AppBar position="static" color="default">
      <Toolbar>
        <Typography variant="title" color="inherit">
          Companies
        </Typography>
      </Toolbar>
    </AppBar>
    <Grid container>
      <Grid item xs={12}>
        <Paper>
          <CompanyList />
        </Paper>
      </Grid>
    </Grid>
  </ApolloProvider>
</React.Fragment>);
```

4. Next, create a container for the `CompanyList` component

Code continued on next page

that will fetch company data using GraphQL. Create a `ContainerList.container.jsx` in `src/components/CompanyList` and add the following imports:

```
import React from 'react';
import {Query} from "react-apollo";
import gql from 'graphql-tag';
import CompanyList from './CompanyList';
```

5. Now declare a query for retrieving all available companies:

```
const COMPANIES_QUERY = gql`
query CompaniesListQuery($language: String) {
   jcr(workspace: LIVE) {
      nodesByQuery(
         query: "SELECT * FROM [jdnt:company] as results
                WHERE ISDESCENDANTNODE(results,
                    '/sites/ digital1/')"
         queryLanguage: SQL2
      ) {
         nodes {
            uuid
            title: displayName(language: $language)
            description: property(name: "overview",
                    language: $language) {
                        value
                    }
            thumbnail: property(name: "thumbnail",
                    language: $language) {
                        url: refNode {
                            path
                        }
                    }
         }
      }
   }
}
`;
```

6. Next, define the functional `CompanyListContainer` component that uses the previously imported `Query` component to fetch and render the CompanyList:

```
const CompanyListContainer = () => {
  const variables = {
    language: 'en',
  };
  const generateURL = path => {
    return `http://localhost:8080/files/
live${path}?t=thumbnail2`;
  };
  return (
    <Query
      query={COMPANIES_QUERY}
      variables={variables}
      fetchPolicy="network-only"
    >
      {({loading, data}) => {
        let companies = [];
        if (data && data.jcr && data.jcr.nodesByQuery) {
          //Build the company data as expected by the
Company component
          data.jcr.nodesByQuery.nodes.forEach(node => {
            companies.push({
              id: node.uuid,
              title: node.title,
              description: node.description.value,
              image: generateURL(node.thumbnail.url.path)
            });
          });
        }
        return (
          <CompanyList
            loading={loading}
            companies={companies}
          />
        );
      }}
    </Query>
  );
};
export default CompanyListContainer;
```

7. Update the existing `index.js` so that `CompanyListContainer` is the default export. Then change:

```
import CompanyList from './CompanyList';
```

To:

```
import CompanyList from './CompanyList.container';
```

8. Lastly, update the `Company` component to make data display dynamically. To do so, change:

```
<CardContent>
  <Typography component="h1" variant="display1">
    Company Name
  </Typography>
  <br />
  <Typography component="p">
    {description.length > 150
      ? `${description.substr(0, 100)}...`
      : description}
  </Typography>
</CardContent>;
```

To:

```
<CardContent>
  <Typography variant="title">{title}</Typography>
  <br />
  <Typography component="div">
    <p
      dangerouslySetInnerHTML={{
        __html:
          description.length > 150
            ? `${description.substr(0, 100)}...`
            : description,
```

Code continued on next column

Code continued on next page

```
      }}
    />
  </Typography>
</CardContent>;
```

Now you have a base JavaScript application that retrieves content from Jahia using GraphQL and uses React and React Material to display the content.

SETTING UP AUTHORIZATION

By default, Jahia's REST and GraphQL API are closed even if nodes have public read permissions. The APIs are closed for security reasons. To open the API for use in your application, you must configure Jahia's security filter module to allow public access. You can find more information about the security filter module here: https://github.com/Jahia/security-filter

To make the data publicly accessible:

1. Add the `org.jahia.modules.api.permissions-myapp.cfg` Jahia configuration file to the `digital-factory-data/karaf/etc` folder with the following content:

```
permission.myapp.api=graphql
permission.myapp.scope=myapp
permission.myapp.
nodeType=jnt:news,jnt:contentFolder,rep:root
permission.myapp.pathPattern=/,/sites/[^/]+/contents/.*,/
sites/[^/]+/files/.*
```

The scope setup here requires creating a JWT token when we integrate authorization in the Apollo Client in the JavaScript code. Please be aware that the value of `permission.myapp.pathPattern` here should match the node paths you will access.

2. You must also create an `org.jahia.modules.graphql.provider-myapp.cfg` file for the CORS authorization with the following content:

```
http.cors.allow-origin=http://localhost:3000
```

3. In Jahia, navigate to tools at http://localhost:8080/tools. Then select **Jahia API security config and filter : jwtConfiguration** and create a new JWT Token using the following settings:

```
Scopes : myapp
Referrer : (empty)
IPs : (empty)
```

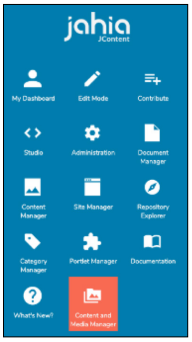
4. Click **Save** and copy the generated token. Replace the example `JWT_DX_TOKEN` value with the copied token.

CREATING CONTENT USING CONTENT AND MEDIA MANAGER

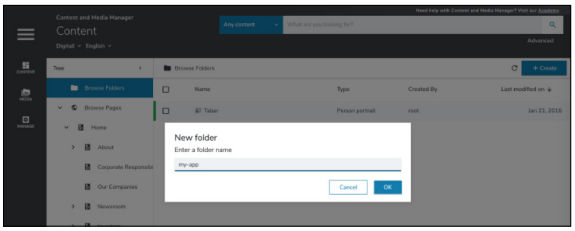
Next, create and publish sample content from **Content and Media Manager** so you have content to browse and display in your JavaScript application.

To create and publish sample content:

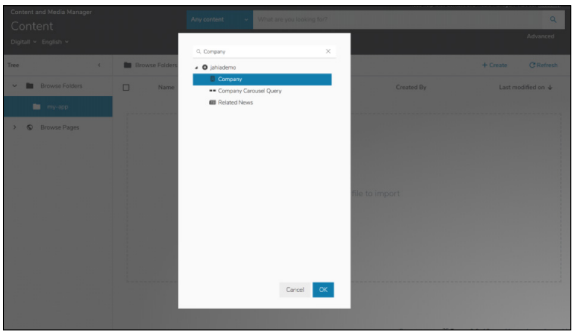
1. In Jahia, click the Jahia logo in the upper-left corner to open the Jahia menu. Then select **Content and Media Manager**.



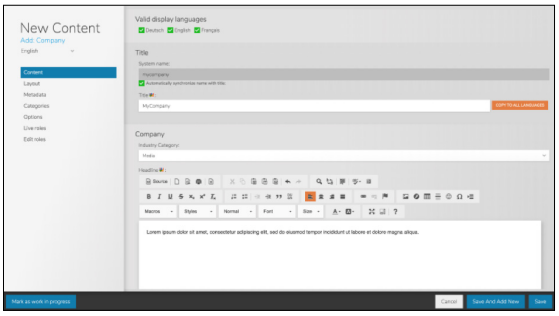
2. Select **Browser Folders**, then click **+ Create** > **New content folder**. Create a new folder named `my-app`.



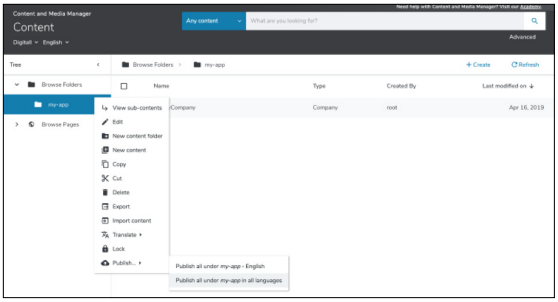
3. In the new folder, click **+ Create** > **New content**. To create a company entry, select **Content:jahiademo** > **Company** and click **OK**.



4. Enter a title, industry, headline, and overview.
The following example shows sample text entered in **Content and Media Manager**.



5. Publish the folder that you created, and your content is ready to be queried.



Wrapping Up

In this Refcard tutorial, you:

- Created a React application.
- Used Content and Media Manager to add sample content to your app.
- Used GraphiQL to execute GraphQL queries and browser schema.
- Set up a basic Apollo Client to perform GraphQL queries to Jahia.
- Set up basic authorization.
- Styled the app's UI using React Material.
- Created a component to retrieve news objects from Jahia.
- Used GraphQL field aliases to improve property retrieval.

Learn more about Content and Media Manager in the Academy.

Written by **Serge Huber, CTO**

Serge Huber is the Chief Technology Officer (CTO) and co-founder at Jahia (http://www.jahia.com). Serge has more than 30 years of experience in developing digital experience solutions in various technologies, including AR and VR, and is constantly striving to find new ways to build high-quality and high-performance software. He has experience in building high visibility, mission-critical applications for large customers (including HomeAway, BNP Paribas, Sodexo, and the European Parliament). He now oversees the future technical development of Jahia's software and manages the interaction with open-source communities such as the Apache Foundation. He is a VP and the initiator of Apache Unomi and a committer on the Apache Jackrabbit Project. He still considers himself primarily a geek and enjoys talking at events like Java One, ApacheCon, UX+Dev Summit, and more.

Devada, Inc.
600 Park Offices Drive
Suite 150
Research Triangle Park, NC 27709

888.678.0399 919.678.0300