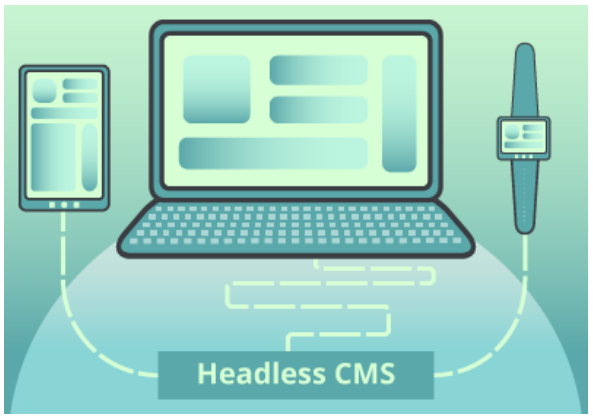
Headless CMS as a Microservice

02/01/2019

CONTENT:

As a Software House that develops a lot of B2C and B2B systems we have to deal with content management systems on the daily basis. We use the most popular CMS solutions like LifeRay or Sitecore. These products offer great functionalities and experience for content creators and marketing. But they also heavily affect our systems architecture and usually tightly couple our solution to chosen CMS. Also our customers have to deal with this kind of heavy and strong dependency.



Do we have any viable alternatives?
With the rising popularity of microservices and API-first approach comes a new and popular solution – **Headless CMS.**

In this article we will guide you through the process of integrating two open source CMS solutions with our sample Micronaut based microservice sales portal, with Single Page Application client written in VueJS.

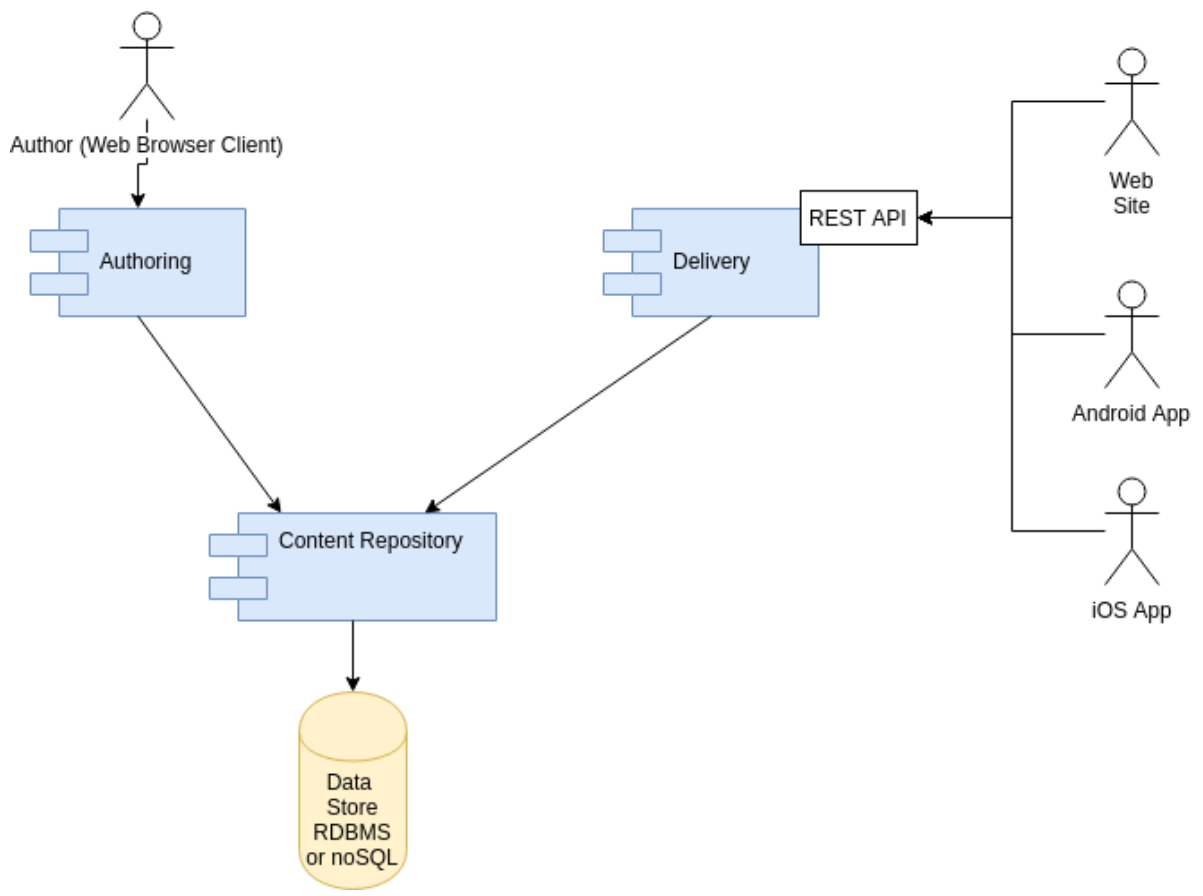
What is Headless CMS?

Headless CMS is back-end only content management system which works as a content repository and gives access to this content via REST (or GraphQL) services.
In traditional CMS you have the following core subsystems:

- content creation and management
- publication workflow
- content delivery
- analysis and monitoring.

Headless CMS focuses just on content creation and publication workflow. It is your application responsibility to get the content and display it in appropriate way based on your applications users needs, device they use and channel they operate on.
You can checkout list of popular open and closed source headless cms solutions here.

Typical Headless CMS architecture



Typical headless CMS consist of the following components:

Pros and Cons of Headless CMS

Advantages of headless CMS:

- no tight coupling between business applications and CMS resulting in flexibility that allows you to choose whatever technology and framework you like for your application
- headless CMS are usually much easier to deploy and use
- ability to easily integrate new channels, as we are not blocked by the functionalities available in CMS
- nicely fits into microservice based solution landscape
- improved scalability and security due to dividing responsibilities of authoring and delivery, delivery can be separately scaled and authoring part can be completely hidden and not accessible to the outside world behind company firewalls.

Disadvantages of headless CMS:

- content authors are not able to preview how created content will look in the applications from inside CMS
- analytical capabilities and content personalization features of full blown CMS cannot be used and must be developed somewhere else

Business Case: Integration of Insurance Sales Portal with CMS

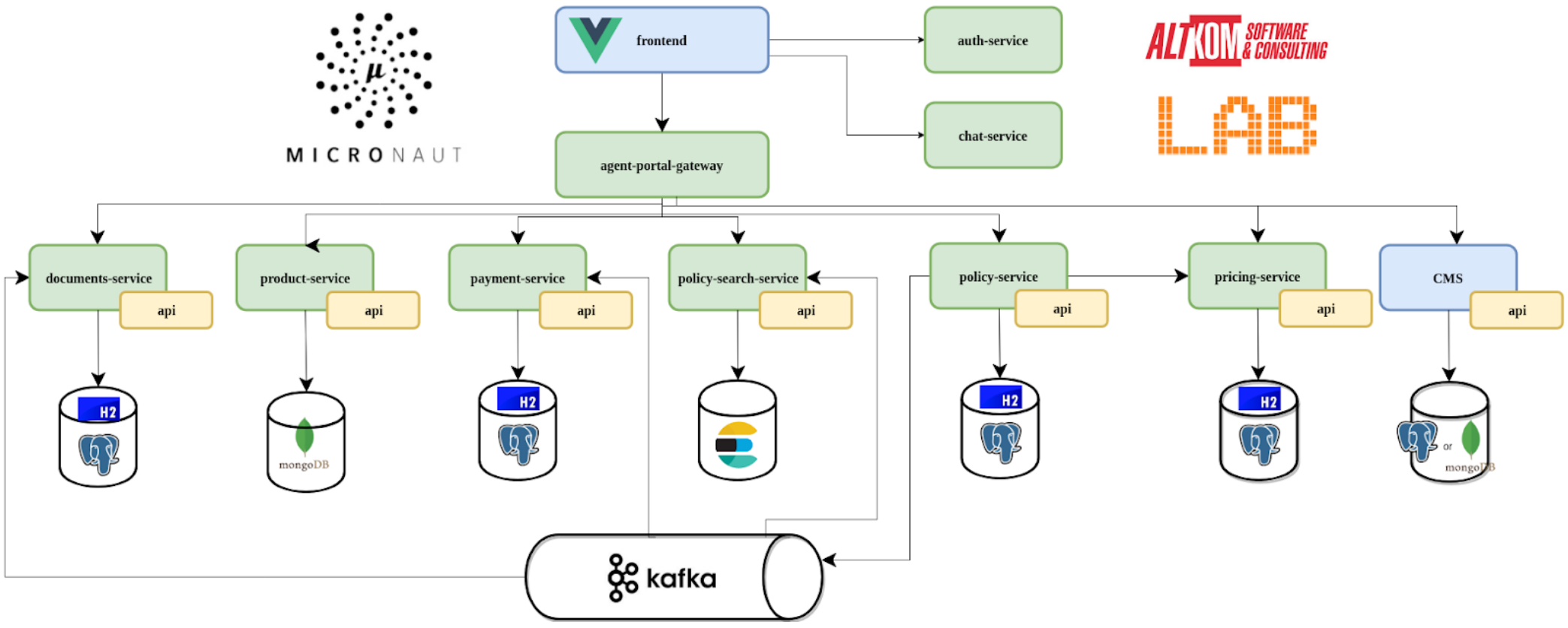
We have a simple Sales Insurance Portal which lets insurance agents select a desired product for their customers, create an offer and finally sale a policy. This system also allows searching and viewing offers and policies so that they can be managed by an agent.
We are going to extend this system functionality by adding two new features:

- blog module
- products home pages.

Blog is going to be managed by CMS. Backend CMS users will create and publish posts. Each post is going to have a title, associated categories and content. Blog posts will be accessed via REST API and displayed in the portal. Agents will also be able to search blogs.

Products home page will display marketing product information in a nice Bootstrap Carousel component. For each product a description, title and picture will be managed in the CMS and accessed via REST in order to be displayed in the portal.

Our solution architecture will look like this:



CMS will be deployed as a back-end system and will be used by content authors to create blog posts and product marketing information. Content authors will use web application that is part of CMS system to create and edit content.
CMS will expose REST APIs that will give access to: blog posts, product, related images and documents.

Our client web application won't access CMS directly. We will use API Gateway pattern for this purpose. There are many reasons for this approach. First is that we do not want to expose CMS APIs to everybody. Second is that we want to control and centralize APIs access from web client app to internal microservices and **we want to treat CMS just as another microservice.** This also gives us a chance to aggregate and adjust content to the channel given api-gateway is dedicated for. For example we can take some data from CMS like product marketing info and take prices from product-catalog microservice and combine it together.

Client application will treat CMS just as another microservice and will access it through api-gateway REST endpoints.

This example can of course be extended. Using the same approach as outlined below we can add:

- FAQ Lists with FAQ Items
- News
- Event calendars
- and many, many more ....

code. It is also a mature solution with large user base and customer base.

Setting CMS up

Let's start our project. We are going to add blog and product info pages managed by CMS to our insurance sales portal.
First thing, we need to setup Hippo CMS.
We setup Hippo CMS by creation of maven project.

Creating project

```
mvn org.apache.maven.plugins:maven-archetype-plugin:2.4:generate \
-DarchetypeRepository=https://maven.onehippo.com/maven2 \
-DarchetypeGroupId=org.onehippo.cms7 \
-DarchetypeArtifactId=hippo-project-archetype \
-DarchetypeVersion=12.5.
```

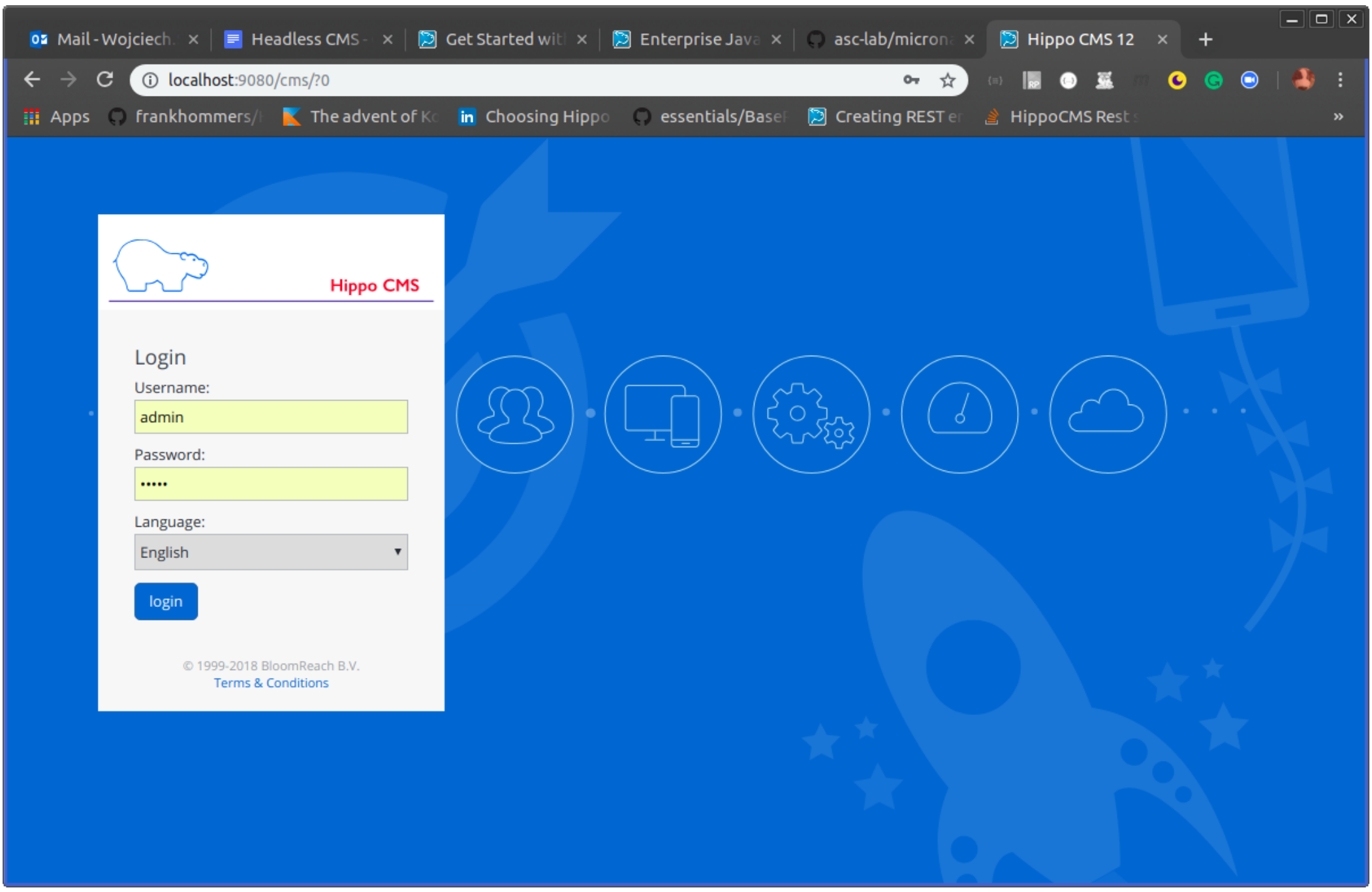
You have to specify your project name and main package name, In our case it was **minicms** and **com.asc.lab.minicms**.
Now you can switch to directory with generated project and build it using maven.

```
cd minicms
mvn clean verify
```

Now we are ready to start the whole thing.

```
mvn -Pcargo.run -Drepo.path=storage
```

After these steps you should be able to access your cms at ~~http://localhost:8080/cms~~.



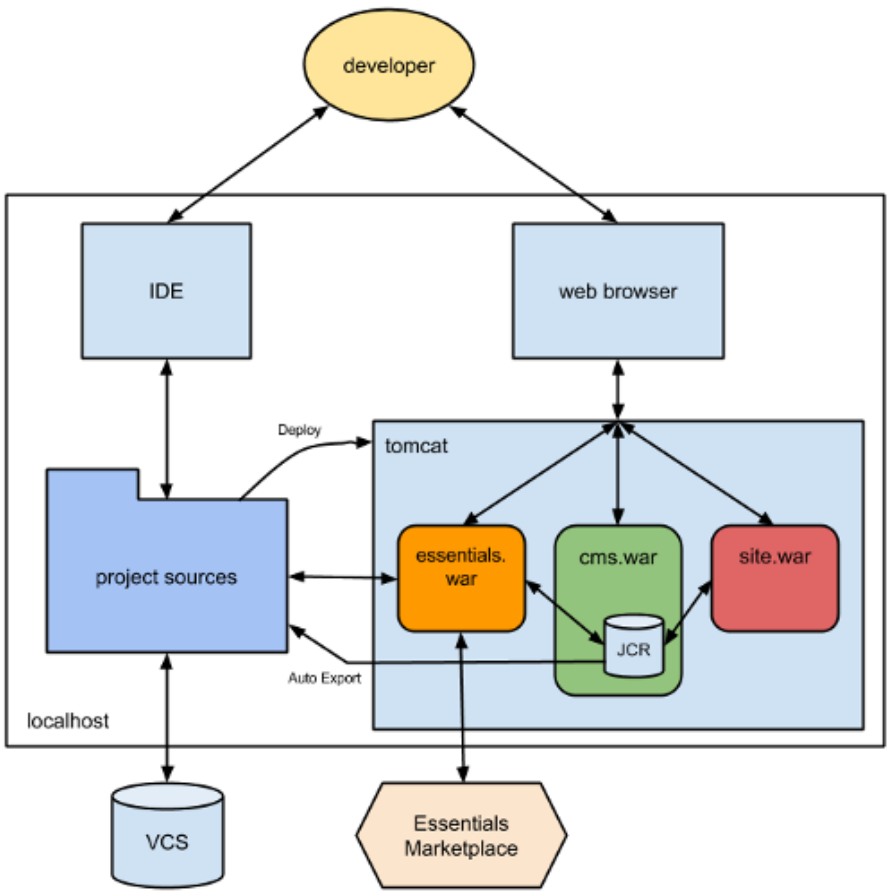
You can now login with admin user and admin password and spend some time to explore content authoring and administration application.
You can also try to follow introductory tutorial available here.

Generated solution contains three applications:

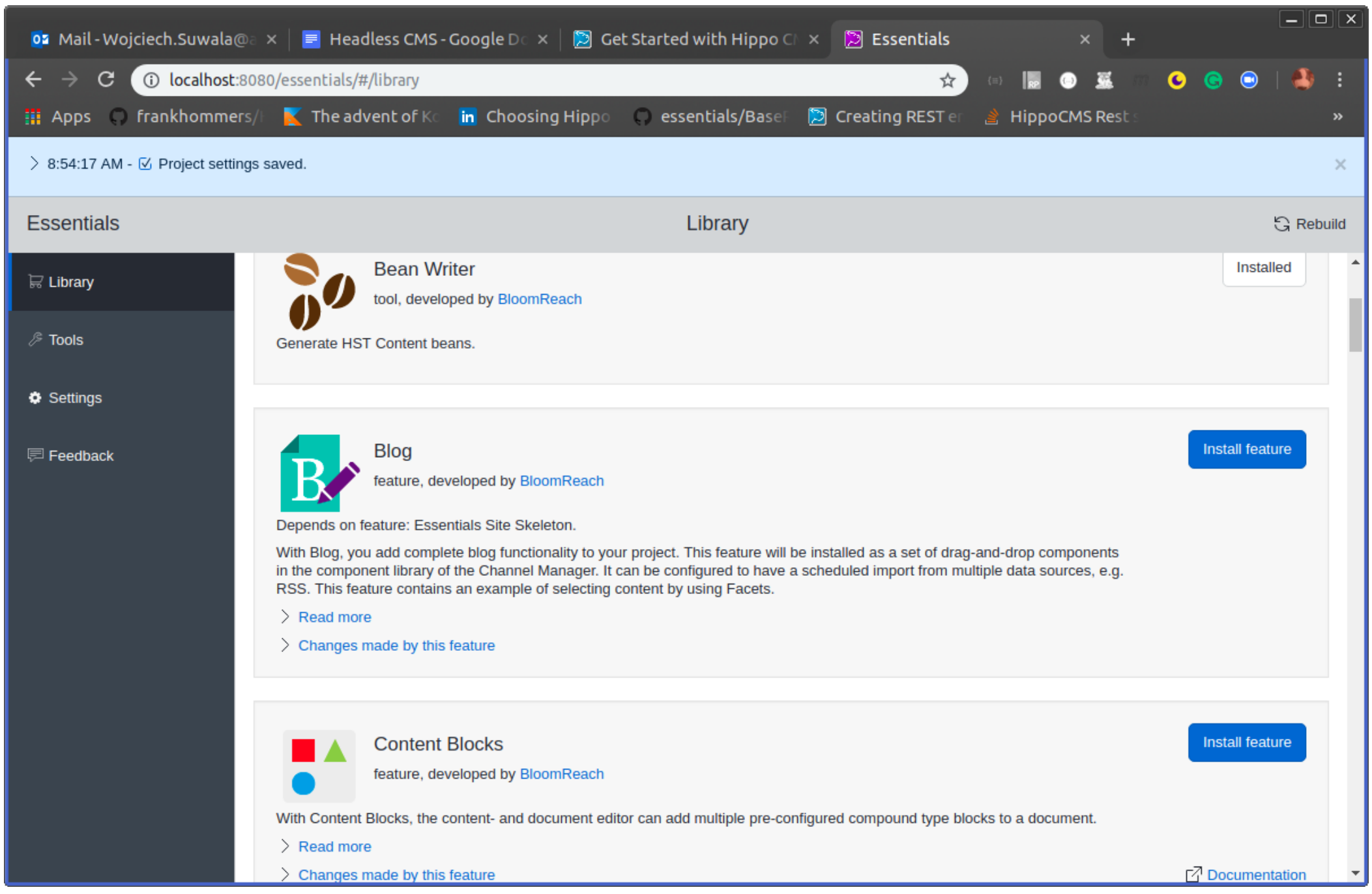
- **cms** – web application for content authors and admins. It allows users to create and manage content types, assets (images, video, pdf files), create and manage content.

- **console** – power tool for advanced repository operations for use by developers and administrators.

The diagram below presents generated project structure.

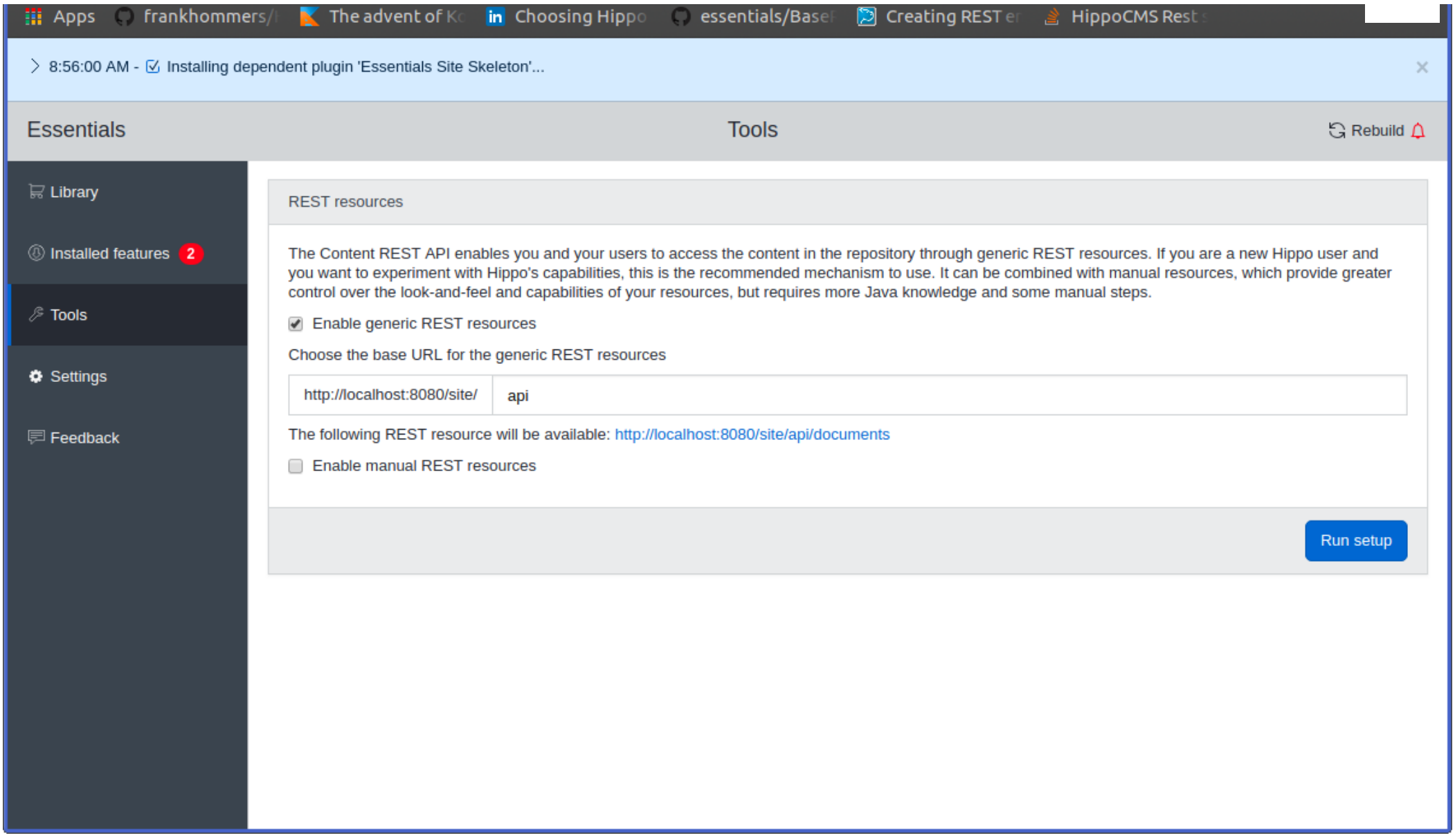


Let's use essentials application to add blog functionality and to enable REST API.



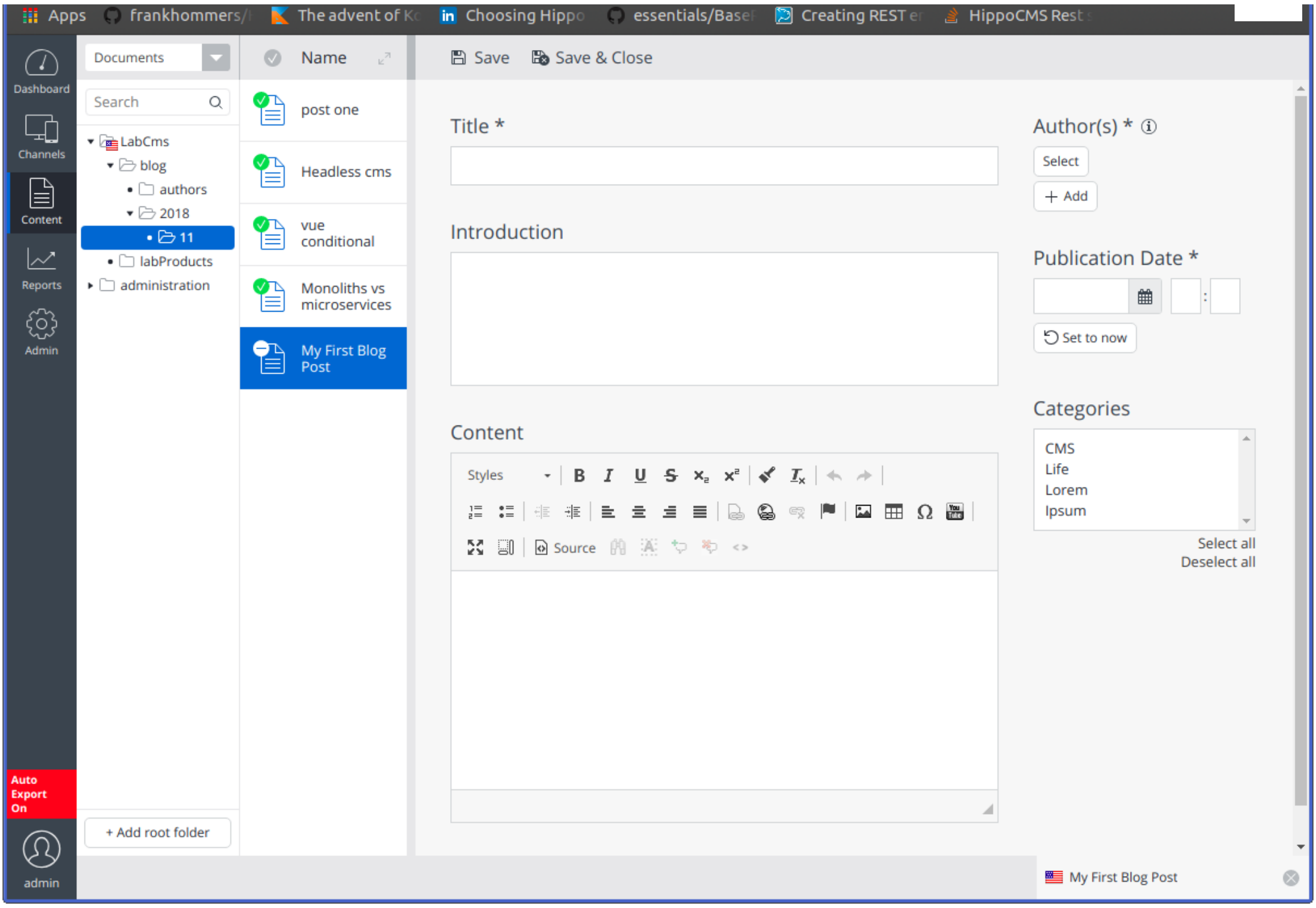
Click install feature. After successful installation you have to rebuild and start your project.

In order to add REST API you have to go to essentials application, select 'Tools' and choose 'Rest Service Setup'.



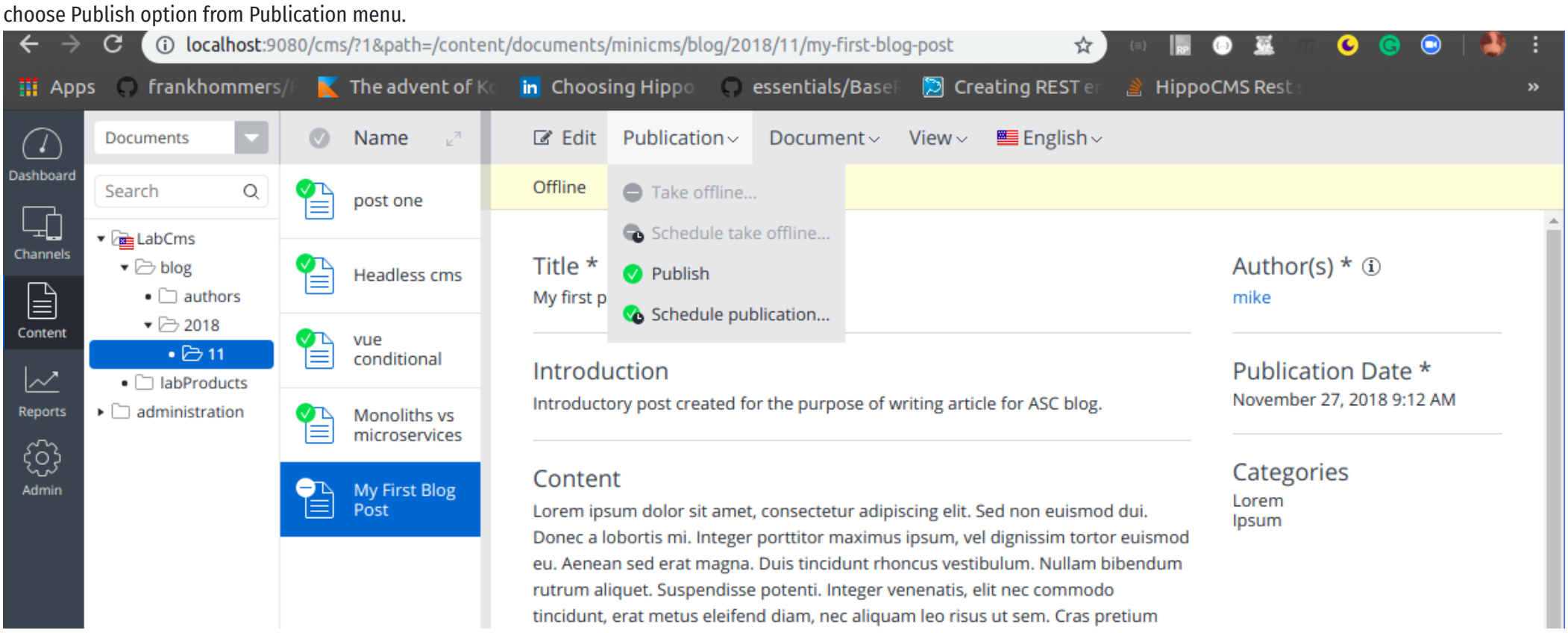
As you can see there are two options. Generic REST and manual REST. We will see manual REST in action later in this article. For now click **Run setup** and follow instructions on the screen.
When setup is finished you have to rebuild and start your project again.
In our sample installation on branch **cms-integration-hippo** we also added the following features and tools: Blog, Events, FAQ, News and Gallery Manager.

It's time to create some content. Switch now to cms application, in the left navbar select Content, expand content tree to see the whole path: LabCms/blog/2018/11. Now you can click on a down arrow next to 11 and select 'Add new blog post'. Give it a name, for example 'My first blog post' and click OK.



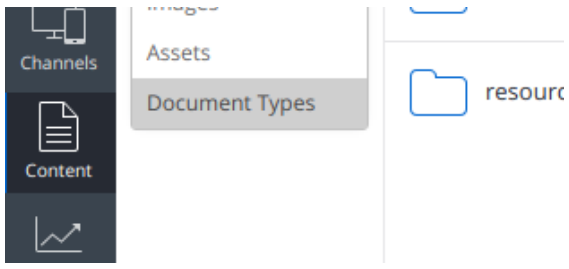
Now you can provide title, introduction information, content, select categories, publication date and authors.

When done press **Save & Close**. You have just created your first content. But we are not done yet. We have to publish our post so other users can view it. All we have to do is to choose Publish option from Publication menu.

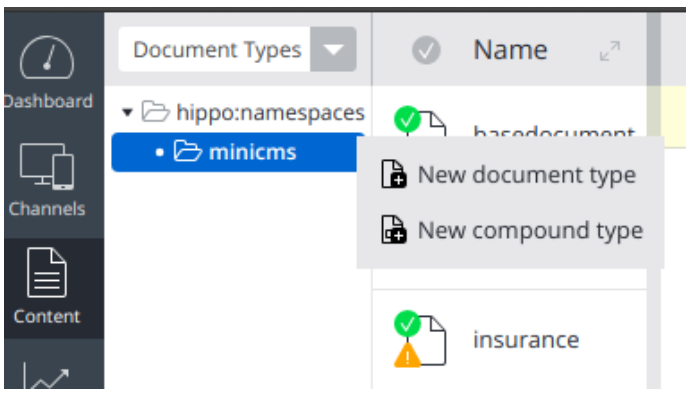


Now our blog posts are ready to be consumed. Now we need to define our insurance product marketing information. There are no templates for this available so we have to create a new content type.

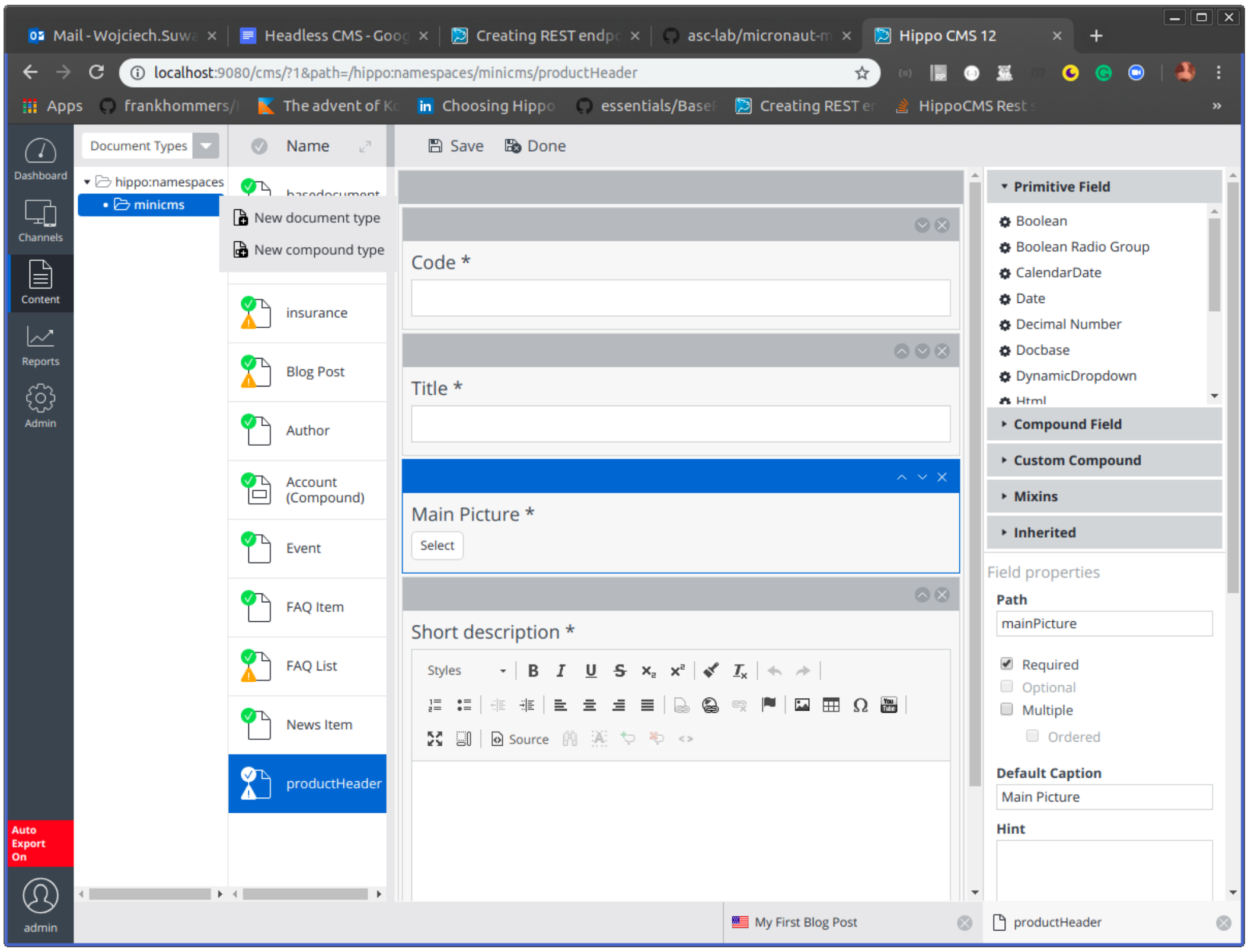
To achieve this select ‘Document Types’ in combo.



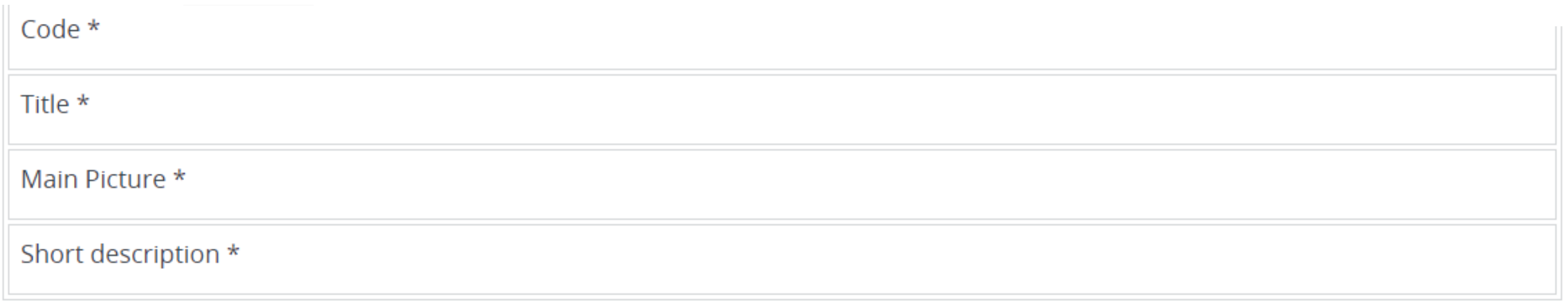
Select minicms node and choose ‘New document type’.



Give it a name (in our case – productHeader) and now you can add fields. It is important to give each file a ‘Path’. Paths will be used to generate property names for classes representing REST resources.
The process of new content type definition is similar to creation of a class and its fields (properties).
You can add text fields, boolean fields, date fields, rich text (html) fields and many others.
You can also add fields that are references to other content types (like references to other classes), fields that contain images or fields that are list of such references. Fields can be marked as required.
Below is a screenshot with **productHeader** content type defined.

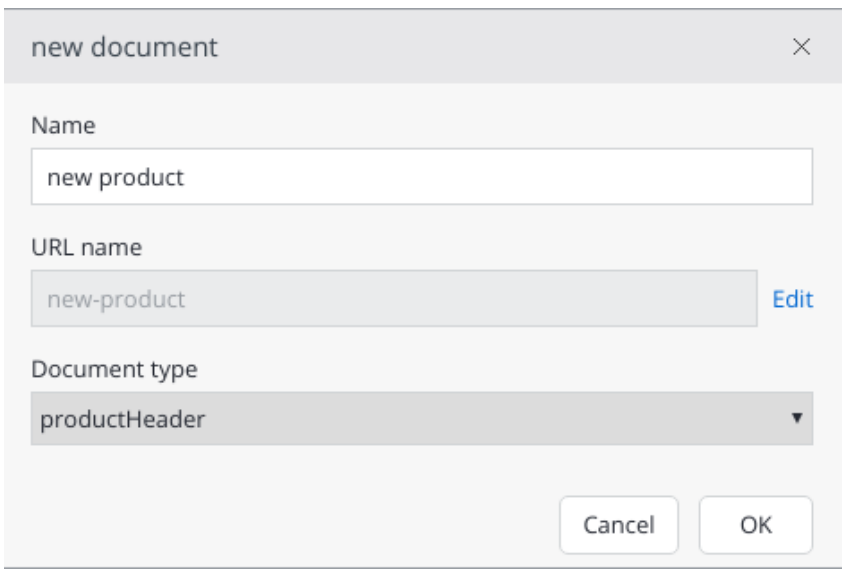


As you can see we have some String fields like code and title, we have a Image Link field main picture and Rich Text (html) field for description.
Once you have added all required fields click ‘Save’, then ‘Done’.
Then select Commit from ‘Type Actions’.

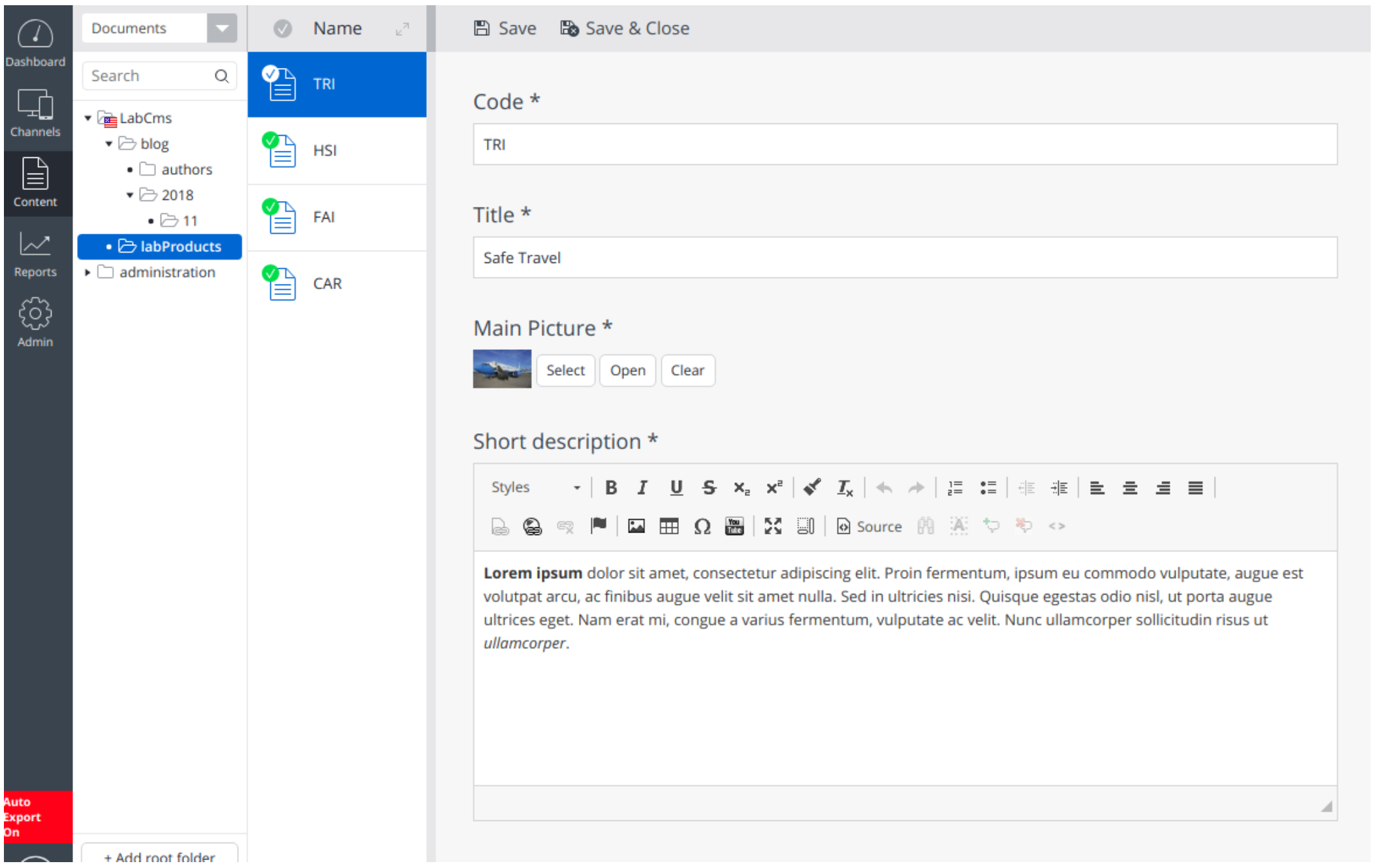


After this action our new content type should be available in the content creation context.

Choose ‘Documents’ from combo, select top level node and click on down arrow next to it. Select ‘Add new document’. In the dropdown for Document type you should be able to select our newly created type – productHeader.



Below you can see a productHeader item for product with code TRI. It is also worth noticing that we create a folder for our products.



Now you can save and publish product definition the same way as in case of blog posts.
Now it’s time to setup and check rest services.

- Content REST API – generic REST API running on the top of delivery tier and exposing all published content based on the document type,
- Custom REST API – is set JAX-RS services generated using REST Service Setup plugin from essentials application. These services can be customized in code.

Content REST API

Generic API exposes two resources: document collection and document detail.

Document collection resource gives us access to all published documents. It has the following features:

- filtering by document type
- paging
- sorting (by default items are sorted by publication date)
- filtering by text query (performs 'contains' query on all document fields)
- document attribute selection

Here is an example of query that returns two most recent blog posts:

```
GET http://localhost:8080/site/api/documents?_nodetype=minicms:blogpost&_offset=0&_max=2
```

This query returns results in the following form


```
{
   "offset": 0,
   "max": 2,
   "count": 2,
   "total": 18,
   "more": true,
   "items": [
      {
         "name": "monoliths-vs-microservices",
         "id": "646d219f-44dc-4ca5-b47b-0b42a052c9a0",
         "link": {
            "type": "local",
            "id": "646d219f-44dc-4ca5-b47b-0b42a052c9a0",
            "url": "http://localhost:9080/site/api/documents/646d219f-44dc-4ca5-b47b-0b42a052c9a0"
         },
         "type": "minicms:blogpost",
         "locale": "en"
      },
      {
         "name": "vue-conditional-rendering",
         "id": "9576b61d-fe03-42a0-82b9-71d32b9e79cf",
         "link": {
            "type": "local",
            "id": "9576b61d-fe03-42a0-82b9-71d32b9e79cf",
            "url": "http://localhost:9080/site/api/documents/9576b61d-fe03-42a0-82b9-71d32b9e79cf"
         },
         "type": "minicms:blogpost",
         "locale": "en"
      }
   ]
}
```


Items element contains list of blog post documents. As you can see id and name fields are returned. Also a link to document details is returned.
Apart from document data you get the information about total number of document, offset, page size (max) and availability of next pages with data (more).

Document detail resource gives access to details of a document with given id.

Example of a query that returns details of blog post with id equal to '8205e43f-d99c-4cc8-894d-0d7b572a795a'.

```
http://localhost:8080/site/api/documents/8205e43f-d99c-4cc8-894d-0d7b572a795a
```

This request returns details of the blog post with given id in the following form

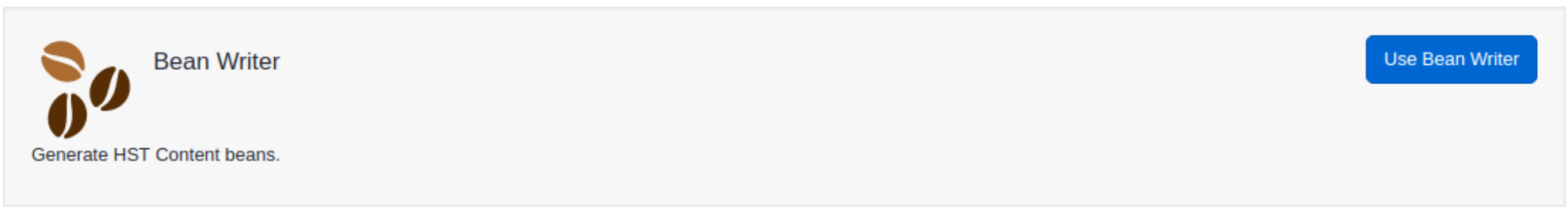
```
  "locale": "en",
  "pubState": "published",
  "pubwfCreationDate": "2018-11-10T20:15:51.579+01:00",
  "pubwfLastModificationDate": "2018-11-19T19:31:33.574+01:00",
  "pubwfPublicationDate": "2018-11-19T19:31:36.909+01:00",
  "items": {
    "minicms:title": "post one",
    "minicms:publicationdate": "2018-11-10T20:16:00.000+01:00",
    "minicms:categories": [
      "cms"
    ],
    "minicms:authornames": [
      ""
    ],
    "minicms:introduction": "this is post one",
    "minicms:content": {
      "type": "hippostd:html",
      "content": "<p>some text</p>\n\n<p><img data-hippo-link=\"atari-800.jpeg\" /></p>\n\n<p>some more</p>\n\n<p><img data-hippo-link=\"zx-s.jpeg\" />
      </p>\n\n<p> </p>",
      "links": {
        "atari-800.jpeg": {
          "type": "binary",
          "url": "http://localhost:8080/site/binaries/content/gallery/minicms/atari-800.jpeg"
        },
        "zx-s.jpeg": {
          "type": "binary",
          "url": "http://localhost:8080/site/binaries/content/gallery/minicms/zx-s.jpeg"
        }
      },
      "minicms:authors": [
        {
          "type": "hippo:mirror",
          "link": {
            "type": "local",
            "id": "e2847966-61b4-407b-8f9e-d8e0849d3cae",
            "url": "http://localhost:9080/site/api/documents/e2847966-61b4-407b-8f9e-d8e0849d3cae"
          }
        }
      ]
    }
  }
}
```


Returned JSON contains id, name, locale, publication date. Inside ‘items’ object you get all the fields defined for given document type. Note how content field is rendered. This field is defined as rich text (html) in document type schema. For this kind of field a special renderer is used which apart from raw html renders list of links inside html that lead to content managed by CMS. We can use this information to properly handle links in our applications as we do not want to expose CMS to external world and we want to have secured access to CMS resources managed by our application.

You can learn more about generic REST API in hippo documentation.

Custom REST API

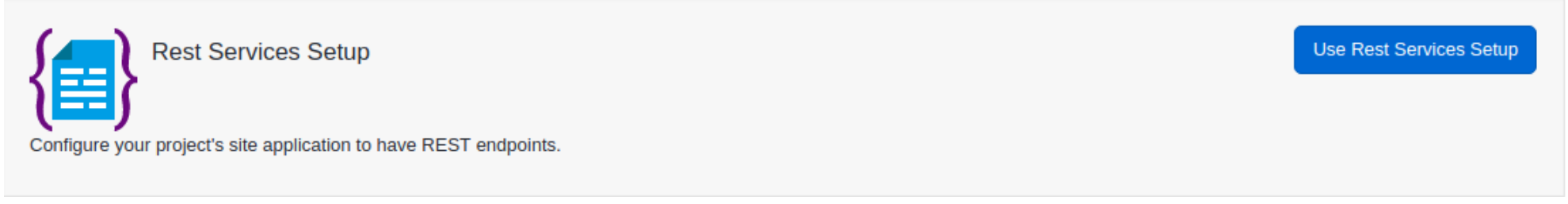
The easiest way to build custom REST API with Hippo is to start with code generation features from essentials application. Open it, select ‘Tools’ from left navbar, find Bean Writer tool and click on ‘Use Bean Writer’



Then click ‘Generate HST Content Beans’. This will generate a class for each document type in your CMS. This can be compared to process of defining entity classes in typical Java web application. The classes are added to the site project, in the beans package.
In our case you can find it in minicms/site/src/main/java/com/asc/lab/minicms/beans/ directory.
Let’s have a look at ProductHeader class which represent bean generated for the document type we created to hold product marketing information.

```
@Node(jcrType = "minicms:productHeader")
public class ProductHeader extends BaseDocument {
    @XmlElement
    @HippoEssentialsGenerated(internalName = "minicms:title")
    public String getTitle() {
        return getProperty("minicms:title");
    }

    @XmlElement
    @HippoEssentialsGenerated(internalName = "minicms:code")
    public String getCode() {
        return getProperty("minicms:code");
    }

    @XmlJavaTypeAdapter(HippoGalleryImageAdapter.class)
    @XmlElement
    @HippoEssentialsGenerated(internalName = "minicms:mainPicture")
    public HippoGalleryImageSet getMainPicture() {
        return getLinkedBean("minicms:mainPicture", HippoGalleryImageSet.class);
    }

    @XmlJavaTypeAdapter(HippoHtmlAdapter.class)
    @XmlElement
    @HippoEssentialsGenerated(internalName = "minicms:shortDescription")
    public HippoHtml getShortDescription() {
        return getHippoHtml("minicms:shortDescription");
    }
}
```

As you can see this class extends BaseDocument and has properties for each field defined in the system.
There are also annotations that provide mapping between these properties and fields in document types.
The adapter annotation (XmlJavaTypeAdapter) is very important as it specifies which class is responsible for rendering field value into desired format. In case of simple types fields it is not present. But for complex types as Image Links or Rich Text adapters are responsible for example for generating information about links to contained images or content referenced from within html field value.
You check this issue from StackOverflow to see how you can control the process of output generation for your bean classes.
Now, when our classes are generated we need to stop Hippo instance, rebuild it and start again.
Next step is to generate REST endpoint. Again we go to the essentials application, to the ‘Tools’ section and we choose ‘Use Rest Service Setup’



Here we must check Enable manual REST resources option, select a mounting point and bean for which we want to generate REST endpoint.

with Hippo's capabilities, this is the recommended mechanism to use. It can be combined with manual resources, which provide greater control over the look-and-feel and capabilities your resources, but requires more Java knowledge and some manual steps.

Enable generic REST resources

Choose the base URL for the generic REST resources

http://localhost:8080/site/	api

The following REST resource will be available: http://localhost:8080/site/api/documents

Enable manual REST resources

Choose the base URL for the manual REST resources

http://localhost:8080/site/	api-manual

Select document type(s) to expose through REST

You may want to add more document types first by installing additional features from the Library or creating custom document types in the CMS Document Type Editor.

account	
author	
basedocument	
blogpost	
eventsdocument	
faqitem	
faqlist	
insurance	
newsdocument	
product	
productHeader	http://localhost:8080/site/api-manual/ProductHeader/
story	

Now you can hit 'Run setup'. This will generate a class for each chosen bean. The classes are added to the site project, in the rest package.
In our case you can find it in minicms/site/src/main/java/com/asc/lab/minicms/rest/ directory.
Let's have a look at ProductHeaderResource class which represent REST endpoint generated for the ProductHeader bean, that we have just created in the previous step.

```
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML, MediaType.APPLICATION_FORM_URLENCODED})
@Path("/ProductHeader/")
public class ProductHeaderResource extends BaseRestResource {

    @GET
    @Path("/")
    public Pageable index(@Context HttpServletRequest request) {
        return findBeans(new DefaultRestContext(this, request), ProductHeader.class);
    }

    @GET
    @Path("/page/{page}")
    public Pageable page(@Context HttpServletRequest request, @PathParam("page") int page) {
        return findBeans(new DefaultRestContext(this, request, page, DefaultRestContext.PAGE_SIZE), ProductHeader.class);
    }

    @GET
    @Path("/page/{page}/{pageSize}")
    public Pageable pageForSize(@Context HttpServletRequest request, @PathParam("page") int page, @PathParam("pageSize") int pageSize) {
        return findBeans(new DefaultRestContext(this, request, page, pageSize), ProductHeader.class);
    }

}
```

As you can see we have a class that is able to respond to http GET requests and returns ProductHeader beans. You can analyze BaseRestResource class to see how it works and extract code from it to provide your own custom methods. Basically you can use any of available search APIs to query content repository and return results.

If you want to know more about custom API generation and development Gary Law's post Creating REST Endpoints in Hippo is a good starting point.
If you want to learn about custom query development with HstQuery in section of Hippo documentation.

Now it's time to stop our cms, rebuild it and start again. After these steps we can test our API.

Let's send a sample request:

```
GET http://localhost:9080/site/api-manual/ProductHeader
```

Now, that our APIs are up and running we are ready to start our work on integrating CMS with our microservice based portal.

Accessing CMS REST APIs

The plan for integration between Hippo CMS and our microservice based portal is presented on the diagram below.
Source code for the final version of our solution can be found at: https://github.com/asc-lab/micronaut-microservices-poc/tree/cms-integration-hippo.

As a first step we need to define operations and data structures for communication between our portal and CMS. Remember that we treat CMS just as another microservice.
For each microservice in our solution so far we created a separate project with api definition.
We do not want our solution to be tightly coupled with any particular CMS, so we are not going to use or reference any Java classes or types created inside CMS solution. We are going to define our own interfaces and types for communication between portal and CMS.
For this purpose we create new maven module called cms-service-api.

For getting, searching and displaying blog posts we define the following interface:

```
    @Get("/{postId}")
    Maybe getBlogPost(String postId);
}
```

First method returns page with blog titles and ids as a result of search or requesting next/previous page. Second method returns details of post with given id.
These methods use the following data structures:

```
@Getter
@Setter
@Builder
public class BlogPostsPageRequest {
    private Integer pageNumber;
    private Integer pageSize;
    private String searchPhrase;
}

@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class BlogPostsPage {
    private int offset;
    private int max;
    private long count;
    private long total;
    private boolean more;
    private List items;
}

@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class BlogPost {
    private String id;
    private String name;
}

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class BlogPostDetails {
    private String id;
    private String name;
    private String displayName;
    private String title;
    private String publicationDate;
    private List categories;
    private String introduction;
    private String htmlContent;
    private List links;
}
```

For image access we define an interface:

```
public interface ImageOperations {
    @Get("/{imageName}")
    InputStream getImageByName(String imageName) throws IOException;

    @Get("/imageset/{+path}")
    InputStream getImageByPath(String path) throws IOException;
}
```

First method returns image by name, second by given path.

Now we have to add blog posts functionality to our application gateway project – agent-portal-gateway.
We add new controller called CmsGatewayController with the following methods:

```
@Get("/blogposts/{postId}")
Maybe blogPost(String postId)

@Get("/imageset/{+imagePath}")
StreamedFile imageByPath(String imagePath)

@Get("/images/{imageName}")
StreamedFile imageByName(String imageName)
```

First method returns page with blog post titles, second returns details of blog post with given id, while the latter two return images.
In order to implement this methods we are going to follow the same approach as for other gateway controllers. We are going to implement HTTP clients that provide interface defined in an api project. In our case we have to implement client for BlogOperations interface and for ImageOperations.
We are going to use reactive HTTP client from Micronaut. Below is the source code for client that implements BlogOperations.

```
@Singleton
public class CmsHippoBlogGatewayClient implements BlogOperations {
    private final RxHttpClient httpClient;

    public CmsHippoBlogGatewayClient(CmsHippoConfig config) throws MalformedURLException {
        this.httpClient = RxHttpClient.create(new URL(config.getUrl()));
    }

    @Override
    public Maybe getBlogPosts(BlogPostsPageRequest pageRequest) {
        Map<String,Object> params = new HashMap<String, Object>();
        params.put("offset", pageRequest.getPageNumber() * pageRequest.getPageSize());
        params.put("max", pageRequest.getPageSize());
        if (pageRequest.getSearchPhrase()!=null) {
            params.put("query",pageRequest.getSearchPhrase());
        }
        String path = "/site/api/documents?_nodetype=minicms:blogpost&_offset={offset}&_max={max}&_query={query}";
        String uri = UriTemplate.of(path).expand(params);
        HttpRequest<?> req = HttpRequest.GET(uri);
        return httpClient.retrieve(req, Argument.of(BlogPostsPage.class)).firstElement();
    }

    @Override
    public Maybe getBlogPost(String postId) {
        Map<String,Object> params = new HashMap<String, Object>();
        params.put("uuid",postId);
        String path = "/site/api/documents/{uuid}";
        String uri = UriTemplate.of(path).expand(params);
        HttpRequest<?> req = HttpRequest.GET(uri);

        return httpClient
                .retrieve(req, Argument.of(Map.class))
                .map(doc -> DocMapper.from(doc).map())
                .firstElement();
    }
…..
}
```

Full source code can be found at https://github.com/. As you can see this is pretty simple code: we read Hippo CMS url from application configuration file and create an instance of HTTP client, then in each method we build URI, create request and execute it, we map results from JSON structures exposed by CMS into data structures defined in cms-service-api.

With HTTP client ready we can finish our controller.

```
    @Inject
    private BlogOperations cmsBlogClient;
    @Inject
    private ImageOperations cmsImageClient;

    @Get("/blogposts{?pageNumber,pageSize,searchPhrase}")
    Maybe blogPosts(@Nullable Integer pageNumber, @Nullable Integer pageSize, @Nullable String searchPhrase) {
        BlogPostsPageRequest pageRequest = BlogPostsPageRequest.builder()
                .pageNumber(pageNumber!=null ? pageNumber : 0)
                .pageSize(pageSize!= null ? pageSize : 2)
                .searchPhrase(searchPhrase!=null ? searchPhrase : "")
                .build();
        return cmsBlogClient.getBlogPosts(pageRequest);
    }

    @Get("/blogposts/{postId}")
    Maybe blogPost(String postId) {
    return cmsBlogClient.getBlogPost(postId);
    }


    @Secured(SecurityRule.IS_ANONYMOUS)
    @Get("/images/{imageName}")
    StreamedFile imageByName(String imageName) throws IOException {
        InputStream is = cmsImageClient.getImageByName(imageName);
        return new StreamedFile(is,imageName);
    }

    @Get("/imageset/{+imagePath}")
    StreamedFile imageByPath(String imagePath) throws IOException {
        InputStream is = cmsImageClient.getImageByPath(imagePath);
        return new StreamedFile(is,imagePath);
    }
}
```

We are done adding blog posts to our api gateway. We can test in using any HTTP client like Postman. We are ready to consume new api in our Vue client application.

We have to add a new option in main menu and a new route:

```
{
            path: '/blog',
            name: 'blog',
            component: loadView('Blog')
},
```

and we have to create two view components: BlogPostsList.vue and BlogPostDetails.vue.
Here is how we get blog posts list:

```
loadBlogPosts() {
                HTTP.get('cms/blogposts?pageNumber=' + this.currentPage + '&pageSize=' + this.pageSize + '&searchPhrase=' +
this.searchPhrase).then(response => {
                    this.blogPosts = response.data.items;
                    this.hasMore = response.data.more;
                });
            }
```

Here is how we get details of blog post:

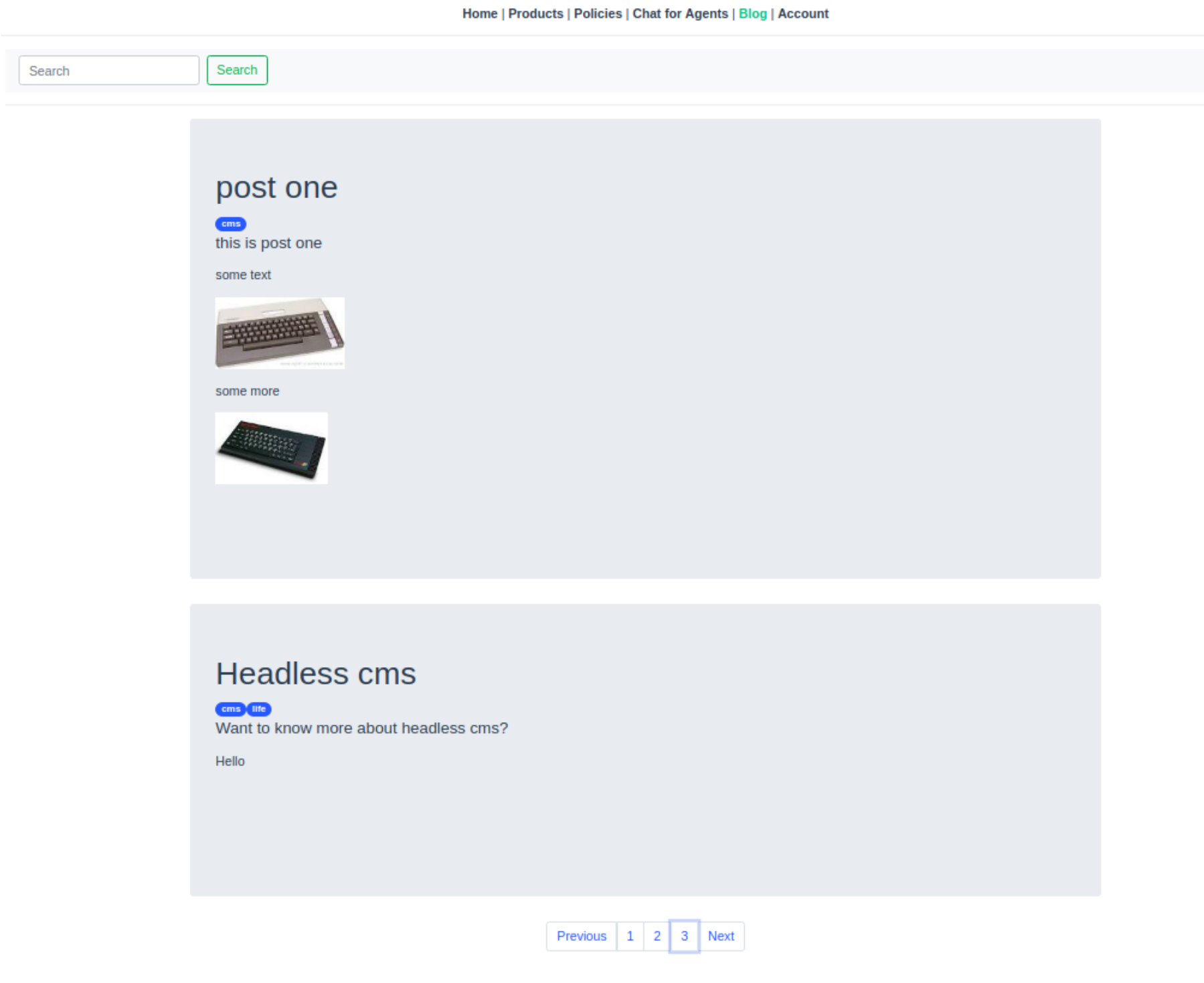
```
HTTP.get('cms/blogposts/' + this.postId).then(response => {
                this.postDetails = response.data;
                this.postDetails.htmlContent = this.resolveLinks(response);
            });
```

This method is a bit more interesting. If you remember from previous parts of this article, links to images that are referenced from inside rich text fields are not returned with fixed URL. Instead all image tags have added attribute data-hippo-link with image name.

Now we have to find all these img tags and properly set src attribute so it points to our api gateway controller and method that returns images. Here is a code that performs this operation. Note that we had to expose this method as unsecured as we have no control over how the browser sends request based on src attribute. We cannot force it to send JWT token in request header and passing it in URL is a serious security risk. In the section covering product headers we will present solution to this problem.

```
    someElement.innerHTML = response.data.htmlContent;
    var links = someElement.querySelectorAll('img[data-hippo-link]');
    for (var index = 0; index < links.length; index++) {
      //links where name = data-hippo-link
      var linkName = links[index].getAttribute('data-hippo-link');
      links[index].src =  imagesHandlerUrl + linkName;
    }
    return someElement.innerHTML;
}
```

Here is a screenshot of final results of our work.



Adding product headers functionality is very similar. Again we start with api interface and data structures.

```
public interface ProductHeaderOperations {
@Get
Maybe<List> productHeaders();
}

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class ProductHeader {
    private String code;
    private String title;
    private String shortDescription;
    private String mainPictureUrl;
}
```

Then we add method to controller:

After that we have to implement client:

```
@Singleton
public class CmsHippoProductHeaderGatewayClient implements ProductHeaderOperations {
    private final RxHttpClient httpClient;

    public CmsHippoProductHeaderGatewayClient(CmsHippoConfig config) throws MalformedURLException {
        this.httpClient = RxHttpClient.create(new URL(config.getUrl()));
    }
    @Override
    public Maybe<List<ProductHeader>> productHeaders() {
        String path = "/site/api-manual/ProductHeader";
        HttpRequest<?> req = HttpRequest.GET(path);
        return httpClient
                .retrieve(req, Argument.of(Map.class))
                .map(doc -> CmsHippoProductHeaderGatewayClient.DocMapper.mapList(doc))
                .firstElement();
    }

    static class DocMapper {
        private Map map;

        public DocMapper(Map map) {
            this.map = map;
        }

        static List<ProductHeader> mapList(Map map){
            return ((List<Map>)map.get("items"))
                    .stream()
                    .map(m -> new DocMapper(m).map())
                    .collect(Collectors.toList());
        }

        ProductHeader map() {
            return ProductHeader.builder()
                    .code((String)map.get("code"))
                    .title((String)map.get("title"))
                    .shortDescription((String)map.get("shortDescription"))
                    .mainPictureUrl((String)((Map)map.get("mainPicture")).get("path"))
                    .build();
        }
    }
}
```

And use it to implement method in our controller:

```
@Get("/productHeaders")
    Maybe<List> productHeaders() { return cmsProductHeaderOperations.productHeaders(); }
```

Now we can add a new component to our home page. Here is how home view looks like:

```
<div>
<ProductsCarousel/>
</div>
```

As with blog posts we divide product headers functionality into two components: ProductCarousel responsible for managing list of product headers and ProductHeader responsible for displaying marketing information about one product.

Here is how we load product headers:

```
created: function () {
   HTTP.get('cms/productHeaders').then(response => {
     this.productsHeaders = response.data;
   });
}
```

and here you can see a template that renders product information:

```
    <router-link :to="{name: 'product', params: { productCode: productCode }}">
      <b-button type="submit" variant="primary">Buy</b-button>
    </router-link>
</b-carousel-slide>
```

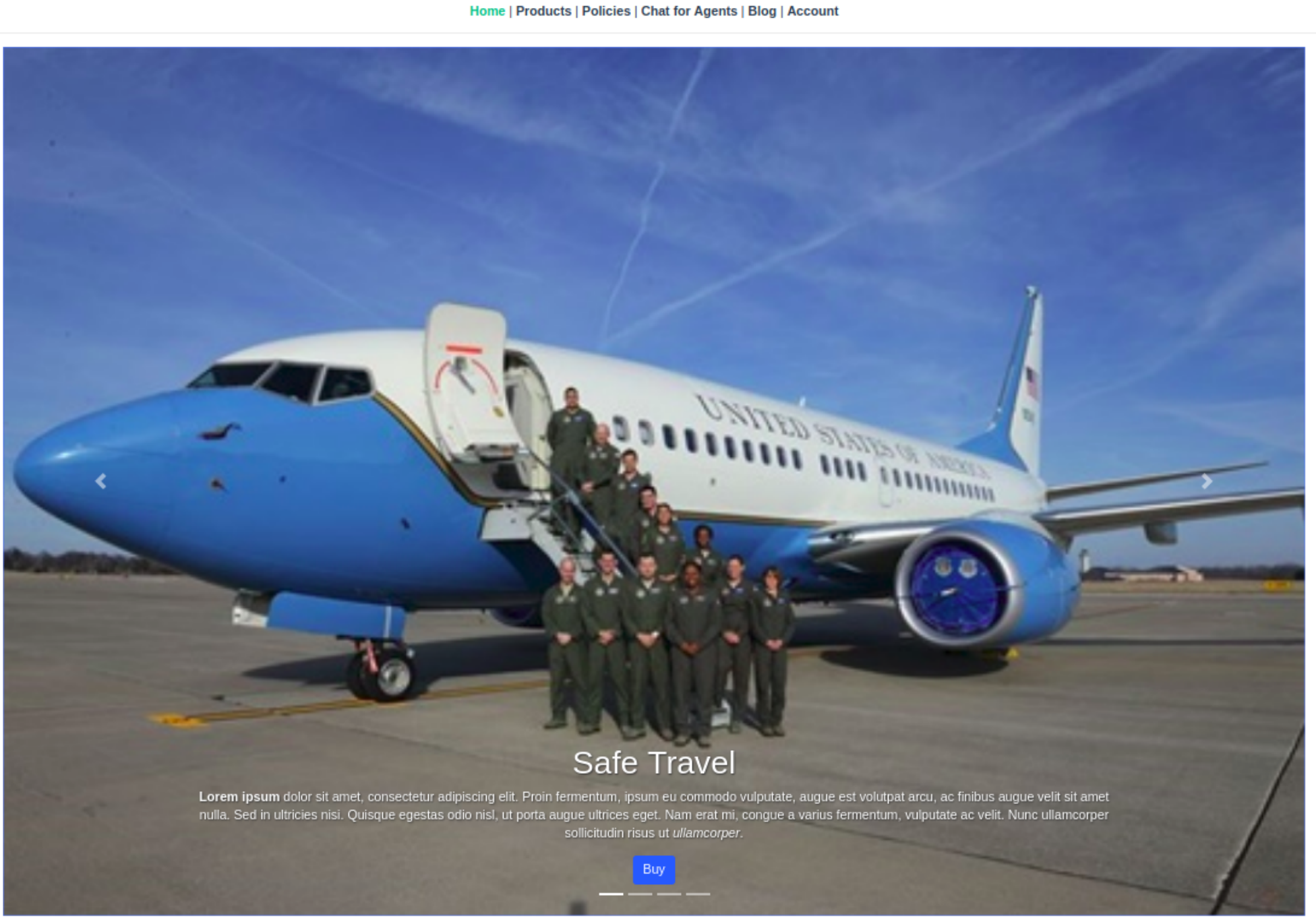
We need to take special care of images. If we want to have a secure access to images we have to get it with AJAX request. This way we can attach JWT security token to request (using an interceptor as usual).

```
HTTP.get('cms/imageset' + this.productImageUrl, { responseType: 'blob' }).then(response => {
                var url = window.URL.createObjectURL(response.data);
                this.imageUrl = url;
            });
```

We execute a GET request with responseType set to blob (this is important). Then we use JavaScript URL object createObjectURL method to create local url from blob and finally set imageUrl property value with this url. This value is bounded to src attribute of image tag (we are using Vue wrapper around Bootstrap 4 carousel component here).

```
<b-carousel-slide v-bind:img-src="imageUrl">
```

Final results look like this:



Further Steps

After all the hard work we now have our microservice based super portal integrated with CMS. Blog posts are visible and searchable, beautiful marketing descriptions and pictures of products are visible on our portal's home page.

We can now add more and more integrations with CMS – for example news or FAQ. There are also technical challenges waiting for us before going into production: we have to switch from H2 database to a real production database like PostgreSQL (more info here) and we also should secure our Hippo REST services (for example using Spring Security).

Strapi CMS integration

StrapiCMS is a typical Headless CMS for building customizable API.

Setting CMS up

We should install strapi-cli through npm (**important:** we must have NPM 6.x and Node.js 10.x, more info about requirements in docs):

```
npm install strapi@alpha -g
```

Next we use CLI to create new StrapiCMS project.

```
> strapi new asclab-cms
🚀 Start creating your Strapi application. It might take a minute, please take a coffee ☕

Lets configurate the connection to your database:
? Choose your main database: MongoDB
? Database name: asclab-cms
? Host: 127.0.0.1
? +srv connection: false
? Port (It will be ignored if you enable +srv): 27017
? Username:
? Password:
? Authentication database (Maybe "admin" or blank):
? Enable SSL connection: false

⏳ Testing database connection...
The app has been connected to the database successfully!
```

In first step 'Choose your main database', we should choose database, which we installed before. I installed MongoDB with this instruction and chose it. After testing database connection, CLI generates application and installs default plugins (content-type-builder, content-manager, users-permissions etc). This process takes a few minutes, so be patient.

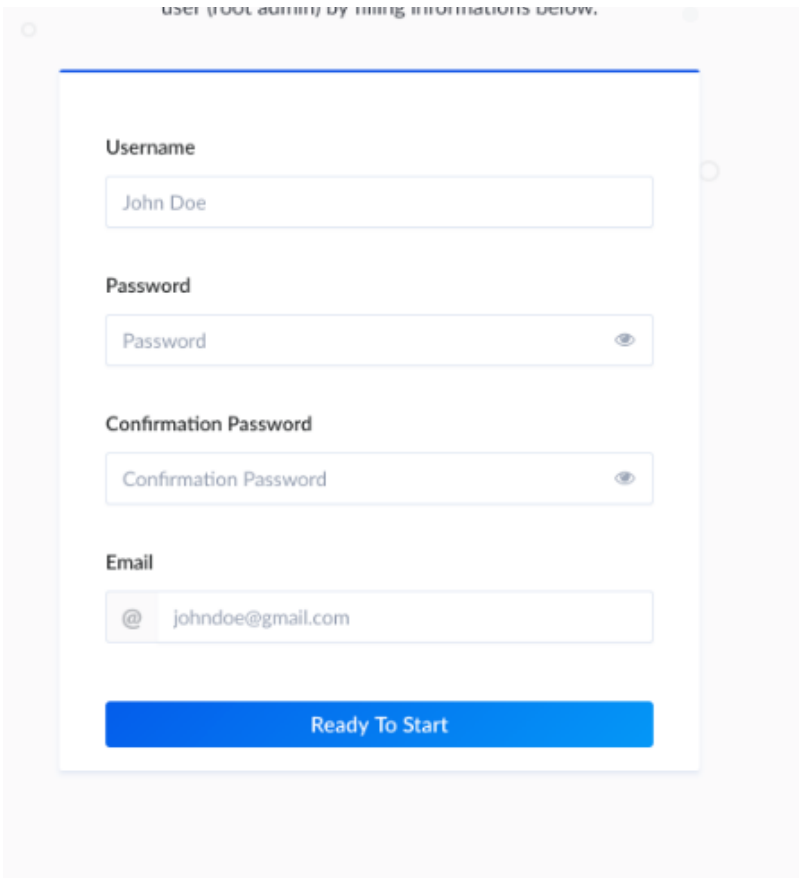
```
🚧  Application generation:
√ Copy dashboard
√ Install plugin settings-manager.
√ Install plugin content-type-builder.
√ Install plugin content-manager.
√ Install plugin users-permissions.
√ Install plugin email.
√ Install plugin upload.

🔌 Your new application asclab-cms is ready at C:\asc-example-repos\asclab-cms.

⚡ Change directory:
$ cd asclab-cms

⚡ Start application:
$ strapi start
```

Now we can change directory and start our headless CMS:

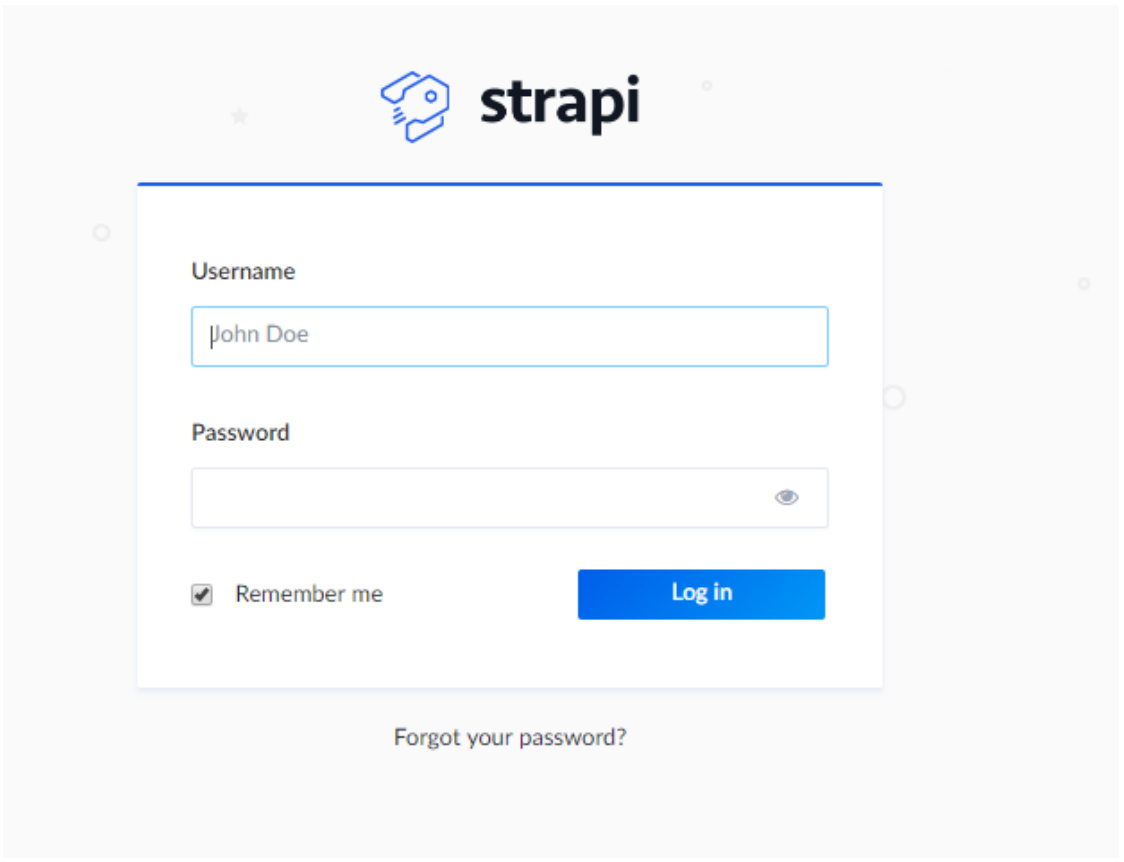
```
cd asclab-cms
strapi start
```

```
> strapi start
(node:23020) DeprecationWarning: collection.ensureIndex is deprecated. Use createIndexes instead.
(node:23020) DeprecationWarning: collection.update is deprecated. Use updateOne, updateMany, or bulkWrite instead.
(node:23020) DeprecationWarning: collection.count is deprecated, and will be removed in a future version. Use collection.countD
cuments or collection.estimatedDocumentCount instead
(node:23020) DeprecationWarning: collection.remove is deprecated. Use deleteOne, deleteMany, or bulkWrite instead.
[2018-11-28T12:51:08.050Z] info Time: Wed Nov 28 2018 13:51:08 GMT+0100 (Central European Standard Time)
[2018-11-28T12:51:08.055Z] info Launched in: 21951 ms
[2018-11-28T12:51:08.056Z] info Environment: development
[2018-11-28T12:51:08.057Z] info Process PID: 23020
[2018-11-28T12:51:08.058Z] info Version: 3.0.0-alpha.14.4.0 (node v8.11.3)
[2018-11-28T12:51:08.060Z] info To shut down your server, press <CTRL> + C at any time

[2018-11-28T12:51:08.061Z] info 💻 Admin panel: http://localhost:1337/admin
[2018-11-28T12:51:08.063Z] info ⚡ Server: http://localhost:1337

[2018-11-28T12:51:08.134Z] debug HEAD index.html (61 ms)
[2018-11-28T12:51:08.139Z] info ⏳ Opening the admin panel...
```

The browser will open automatically http://localhost:1337/admin. At the first start-up we need to create a user:

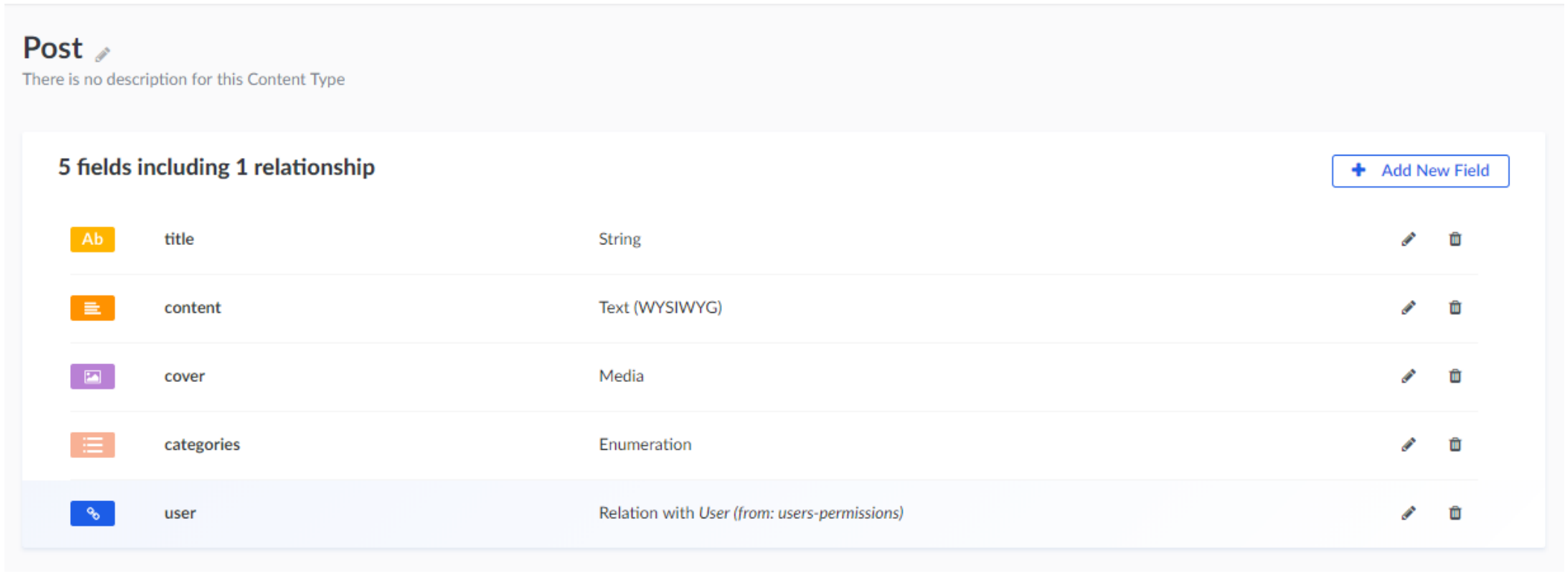


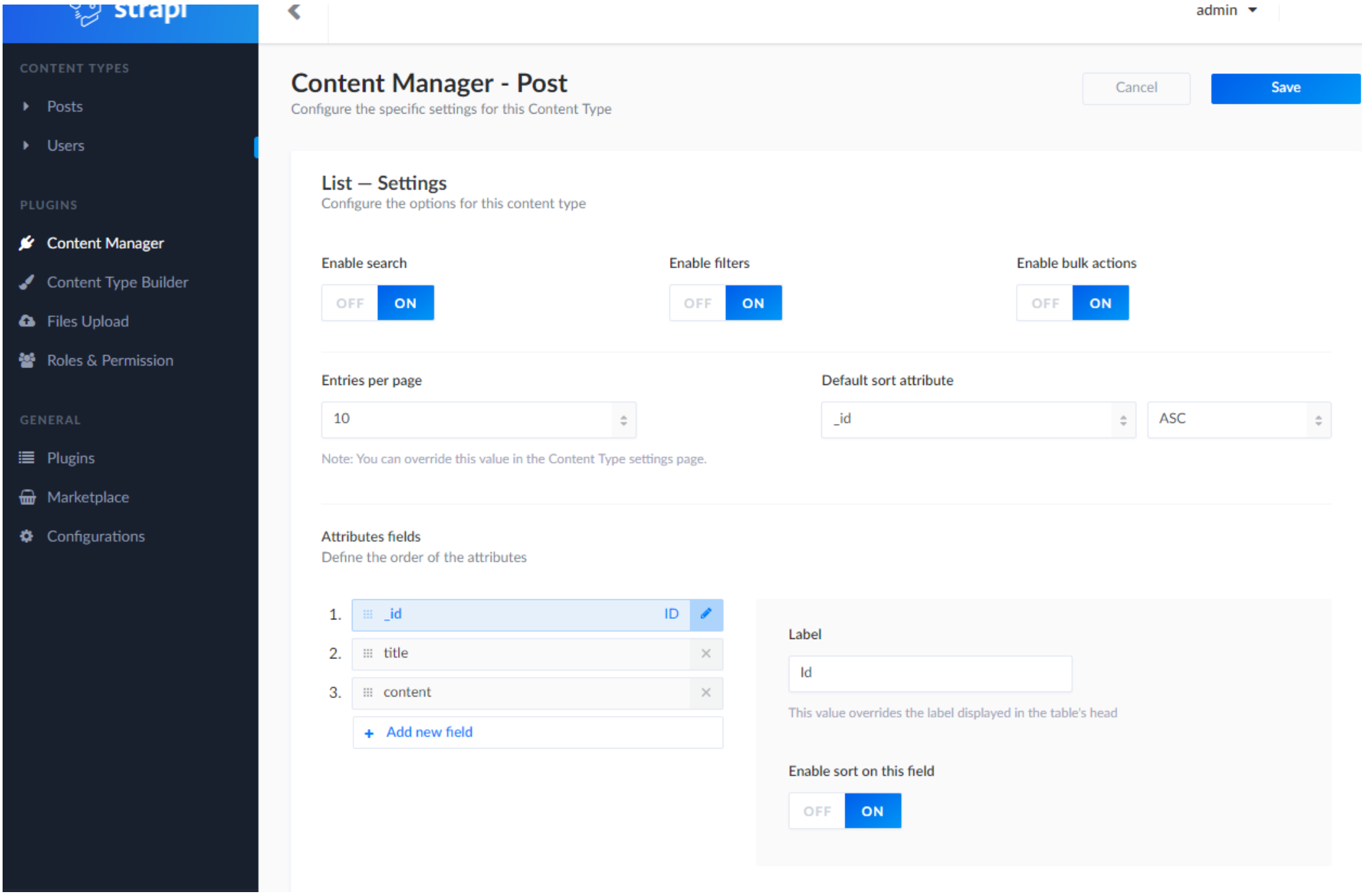
The next time we get a normal login screen:



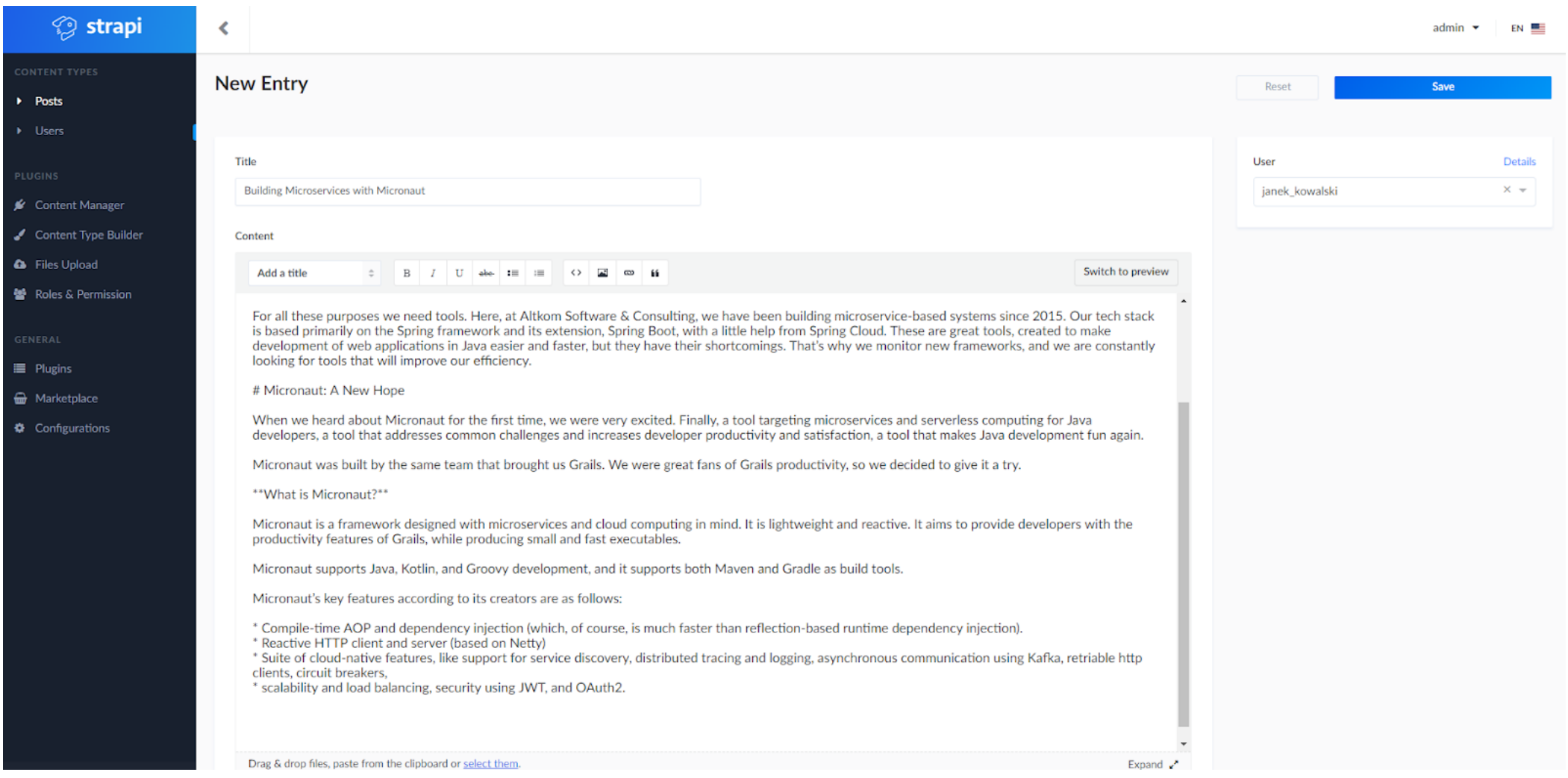
In admin panel authors can create and manage content types, upload assets, create, review, schedule and publish content. Admins can also add new features (plugins) to an instance of CMS using Marketplace.

Based on tutorial from official docs we created new Content Type – Post. We would like to have very similar model in Hippo and Strapi so I created something like this:





Now we can create the first blog post. Click on ‘Posts’ label in left sidebar and next ‘Add new post’.

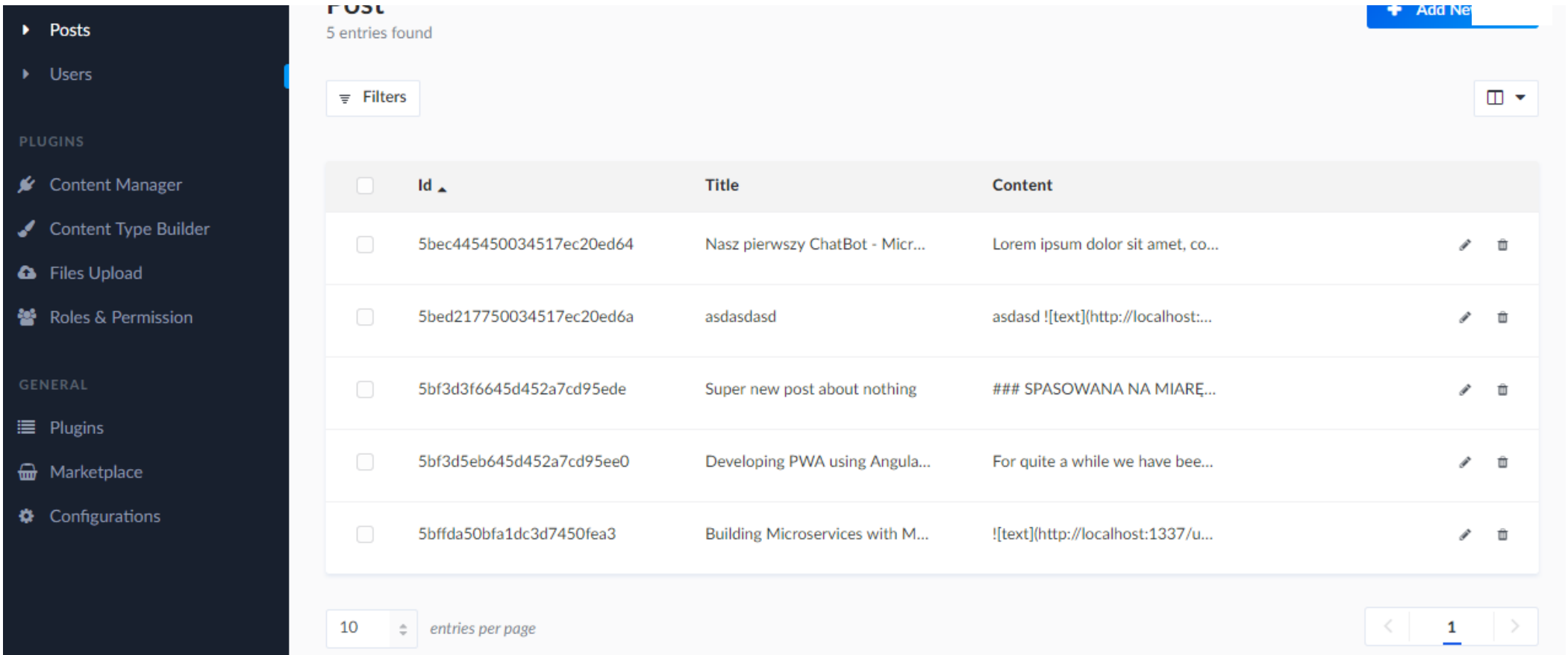


Currently, if we choose Text (WYSIWYG) as field type, Strapi only provide a markdown editor with a preview system. This editor for me is OK, but it can be difficult for non tech users, because it is not properly a WYSIWYG editor that they usually expect (for example now user cannot preview and create content at the same time).
In Public Product Roadmap Strapi’s authors added card about this feature.

We can add new images to our content by drag and drop directly to entry:

Or through File Upload:

I added a few posts to test API.



That's all. In Strapi our test not cover full business case – we test only blog feature.

Why? In our opinion, adding product catalog is just formality.

Exposing CMS REST APIs

REST API for created Content Type is available ad-hoc at address:
http://localhost:1337/[content-type], for example:
http://localhost:1337/posts and return all of entities:

```
  {
    "title": "Nasz pierwszy ChatBot - Microsoft Bot Framework w Azure",
    "content": "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut lab
Sem nulla pharetra diam sit amet nisl suscipit adipiscing. Quis lectus nulla at volutpat diam ut venenatis tellu
mattis vulputate enim. Nibh praesent tristique magna sit. Odio ut sem nulla pharetra. Sit amet aliquam id diam m
phasellus egestas tellus rutrum tellus. Tincidunt vitae semper quis lectus nulla at volutpat diam ut. Tincidunt
faucibus interdum posuere lorem ipsum dolor sit amet. Cursus in hac habitasse platea dictumst. Amet dictum sit a
Tincidunt nunc pulvinar sapien et. Lorem donec massa sapien faucibus et molestie ac feugiat. Leo duis ut diam qu
pharetra diam sit.\n\nIpsum a arcu cursus vitae. Vulputate odio ut enim blandit volutpat maecenas. Blandit cursu
aliquet lectus proin nibh. Aliquam nulla facilisi cras fermentum odio eu feugiat pretium. Sit amet venenatis urn
praesent tristique magna sit amet purus. Sed vulputate odio ut enim blandit volutpat.\n\nVitae turpis massa sed
ornare arcu odio ut. Posuere morbi leo urna molestie at elementum eu facilisis sed. In egestas erat imperdiet se
eget mauris pharetra et ultrices neque ornare. Quis lectus nulla at volutpat diam ut venenatis tellus.\n\nMattis
commodo viverra. Eget arcu dictum varius duis at consectetur lorem. Pellentesque massa placerat duis ultricies
vestibulum lorem sed risus ultricies tristique nulla. Cras pulvinar mattis nunc sed blandit libero volutpat sed
    "categories": "",
    "_id": "5bec445450034517ec20ed64",
    "createdAt": "2018-11-14T15:50:44.837Z",
    "updatedAt": "2018-11-20T10:58:57.743Z",
    "__v": 0,
    "id": "5bec445450034517ec20ed64",
    "user": {↔},
    "cover": {↔}
  },
  {
    "title": "asdasdasd",
    "content": "asdasd\n![text](http://localhost:1337/uploads/2f3e2397f7e44c5298ba5aa3df28f422.gif)",
    "categories": "",
    "_id": "5bed217750034517ec20ed6a",
    "createdAt": "2018-11-15T07:34:15.006Z",
    "updatedAt": "2018-11-20T10:29:22.420Z",
    "__v": 0,
    "id": "5bed217750034517ec20ed6a",
    "user": {
      "confirmed": true,
      "blocked": false,
      "_id": "5bf3e202e0584228a857f323",
      "username": "janek_kowalski",
      "email": "jan_kowalski@example.pl",
      "__v": 0,
      "id": "5bf3e202e0584228a857f323",
      "role": "5bec410b8155a528381b5742",
      "posts": null
    },
```


We can filtering, sorting and paging entries. Results can be filtered with some keywords (`=, _ne, _lt, _gt, _contains etc`), for example:

http://localhost:1337/posts?title=Nasz%20pierwszy%20ChatBot%20-%20Microsoft%20Bot%20Framework%20w%20Azure&content_contains=Lorem

http://localhost:1337/posts?createdAt_gte=2018-10-14

If this operators are not enough, we can install GraphQL plugin and use all GraphQL features.

Accessing CMS REST APIs

We used the same interfaces and data structures as Hippo, described above. The method of accessing endpoints is very similar to HippoCMS, so I don't describe it exactly.

Full source code is available on branch **cms-integration-strapi**.

The differences are primarily in the way of mapping response (StrapiBlogGatewayClient.java) and rendering article content (BlogPostDetails.vue), because Strapi return content as markdown syntax, not html.

Further Steps

Strapi allows you to define custom authentication providers, so we should try compose our auth-service with Strapi and use this as Central Authentication Server (CAS). Of course, we should also finish our business case so that the integration with Strapi will reach the same level as the integration with Hippo.

Summary

We hope that this article showed you that integration between CMS and your business application is pretty easy. With just few simple steps you can use and display content from CMS in any kind of application – it can be a modern web app built using Vue, React or Angular, it can be mobile app developed with Xamarin Forms or any other technology.
You don't have to couple your application with heavy-weight CMS juts to have some basic features like blogs, FAQs, about pages or display marketing information. You don't have to build into your own system content editors or publishing workflows – you can use headless CMS for this.
Comparing HippoCMS and Strapi, both solutions have their advantages. If you are already running on JVM, Hippo will be very familiar and you can use your usual toolset to customize it and extend it. Strapi is lighter and easier to setup, but requires NodeJS skills.
There is also plethora of cloud based solutions which you can explore on your own, the most promising ones are Contentful and Kentico Cloud.

Authors:

Wojciech Suwała, Head Architect, ASC LAB

Robert Witkowski, Senior Software Engineer, ASC LAB

Tagi:
Custom software • Custom Software Development • Customer service software • Dedicated IT solutions • Dedicated software • Headless CMS • HippoCMS • IT companies • Microservices • Software development • Software engineering • Software House • Software House Poland • Software producer • Strapi

Add comment