CUE Front

Developer Guide

1.4.0-3

Table of Contents

1 Introduction

CUE Front is a collection of web services that together serve content to client applications such as browsers and native mobile/tablet apps. The CUE Front web services are:

Cook
A back-end service that retrieves content from the Content Store and serves the content to clients as JSON data via an HTTP-based **content API**.

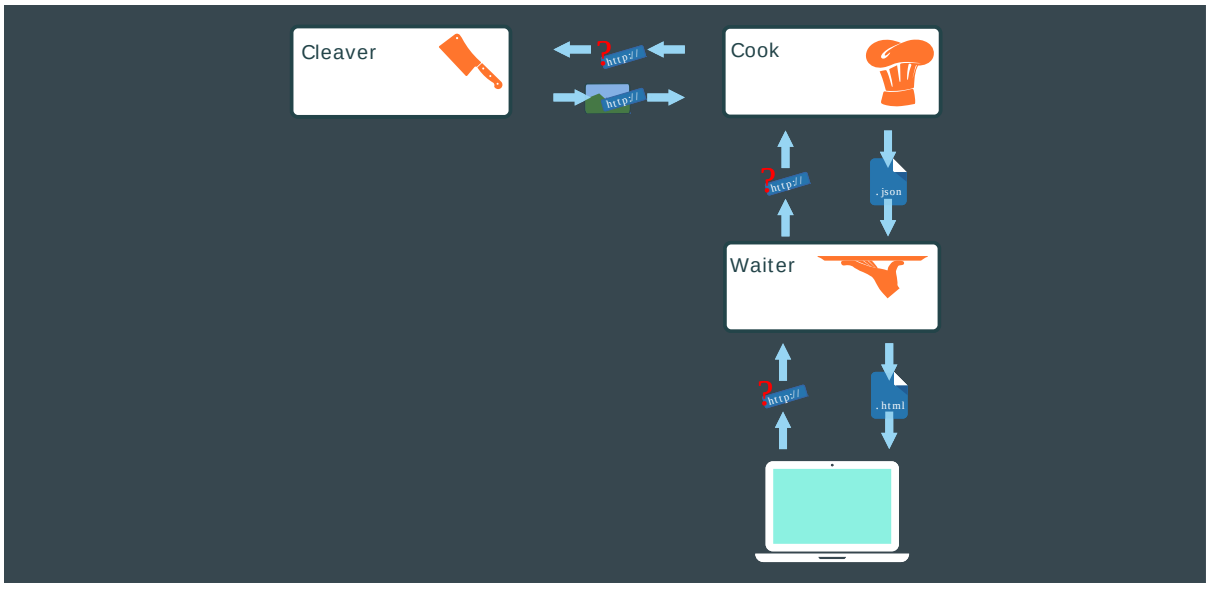
Cleaver
A back-end service that retrieves, crops and resizes images for the Cook.

Waiter
A front-end service that responds to requests from browsers and other HTTP clients. The Waiter passes on incoming requests to the Cook and is responsible for rendering the JSON data returned by the Cook as HTML.

Fridge
An optional caching service that can be used in several different ways together with the other CUE Front components.

The CUE Front services (or **microservices**) are not embedded in the Content Store. They are free-standing entities that only communicate with the Content Store and each other via HTTP. Although they will often be installed together on a single machine, they can, if required, be run on different machines in different locations, or in the cloud.

The following diagram shows how HTTP requests and responses flow between the Waiter, the Cook, and the Cleaver:



Both the Cook and the Waiter satisfy incoming requests by sending requests either to the Content Store's REST API or to a Fridge caching layer.

CUE Front is intended to serve as a more modern replacement for the Content Store's existing built-in presentation layer. It offers a number of advantages over the old presentation layer, including:

- **Technology independence:** CCI Europe's old presentation layer required the use of Java Server Pages (JSP) to build web pages. The Cook's content API, on the other hand, supplies page content as language-neutral JSON data, freeing you to use whatever language and technology you prefer for your front-end component. The Waiter that we supply with CUE Front is written in PHP, but use of this component is entirely optional. You can replace it with software written in any language you like. And in the case of mobile/tablet apps, you can dispense with a Waiter altogether, and serve JSON content directly to the app.
- **Scalability:** Scaling web sites built with the old presentation layer involved installing multiple instances of the entire Content Store, and required complicated caching strategies to avoid overloading the database. The CUE Front components are completely decoupled from the Content Store and can be scaled separately. A complete copy of all the content in the Content Store's database can be stored in one or more Fridges and all the other CUE Front components configured to get their content from a Fridge rather than directly from the Content Store. Fridge contents are kept up-to-date by pushing changes from the Content Engine when they occur. This means that you only need enough Content Store instances to support your editorial operation, and web site scaling is a completely separate issue.
- **Upgradeability:** CUE Front is designed to support blue/green deployment for frequent upgrades to the published web site. Since CUE Front is completely decoupled from the Content Store, such deployments have no effect on the back end. Conversely, upgrading the Content Store has no effect on the front end, if all web site content is being served from a Fridge. It is possible to take all Content Store instances offline simultaneously without affecting published sites in any way (other than the lack of updates to the content).

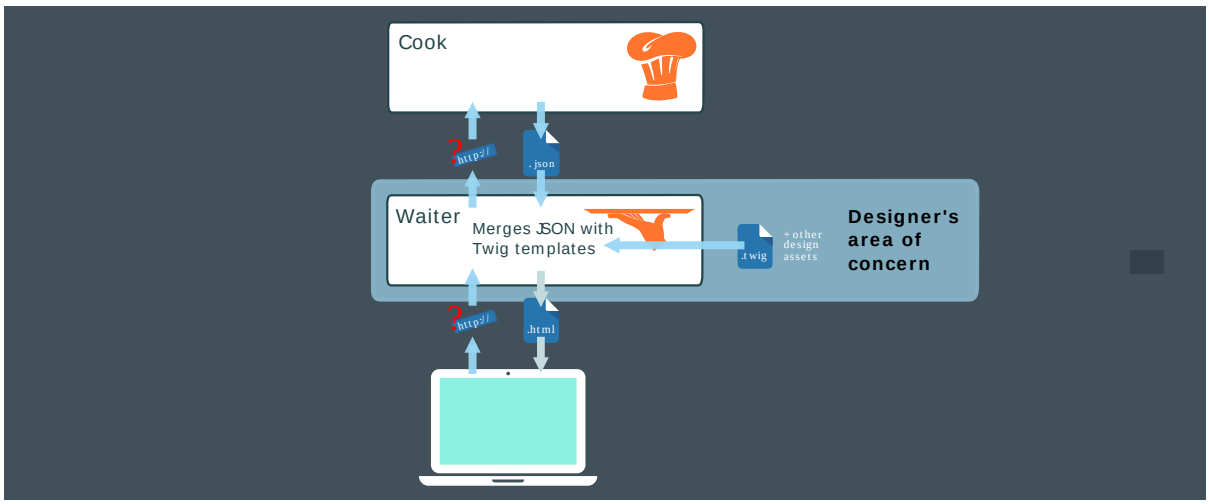
Breaking the presentation layer into separate services encourages separation of concerns: front-end developers/designers can work exclusively with the Waiter (or some other front-end service), and do not need to know anything about Cook or Cleaver. Similarly, back-end developers can concentrate on ensuring that the Cook delivers the required content to the front end, and need not concern themselves with how it is presented.

1.1 CUE Front for Designers

> This section assumes that you use the Waiter supplied with CUE Front to render your web pages. This may well not be the case, since one of CUE Front's main objectives is to give customers the freedom to choose their own front-end technologies. The Cook serves web page content as language- and technology-independent JSON data that can easily be consumed by any front-end component — both server-based web applications and client-side applications such as mobile native apps.

If you are a designer or pure front-end developer, then you will only work with the Waiter and an accompanying design tool called Patternlab. The Waiter is a PHP application that uses the Twig templating engine to serve HTML pages. When the Waiter receives a request from a client, it simply

forwards the request to the Cook. The Cook returns a JSON response. The Waiter then merges the returned JSON data with the appropriate Twig template and returns the result to the client.



As a designer, therefore, your responsibilities are to create a set of Twig templates and other design assets that generate pages from the JSON data supplied by the Cook. The supplied JSON data is your interface with the back-end developer: if it is insufficient, or badly suited to the production of the required pages, then it is up to t[redacted] back-end developer to modify the data supplied by the Cook.

The Waiter supports **styleguide-driven development** – specifically, atomic design. A **living style guide** called Patternlab is delivered with the Waiter. Patternlab is a web application that supports atomic design by presenting all of a web site's atomic design components in a browseable catalog. Using Patternlab, you can see what pages (and all the individual design components from which the pages are built) look like on different devices. Patternlab does this by merging the design's Twig templates with static JSON data fragments. This means that you can use Patternlab to work "off-line" on a web site design — that is, without any access to the Cook or the Content Store.

1.1.1 What is Patternlab?

Patternlab is a PHP web application for web designers that supports atomic design. Atomic design breaks web page designs down into re-usable components called **atoms**, **molecules** and **organisms**, and in this way helps designers to work more consistently and efficiently. Patternlab is basically a browser for these components: you can use it to browse the individual components and see what they look like, and also simultaneously examine the template source code that produces them.

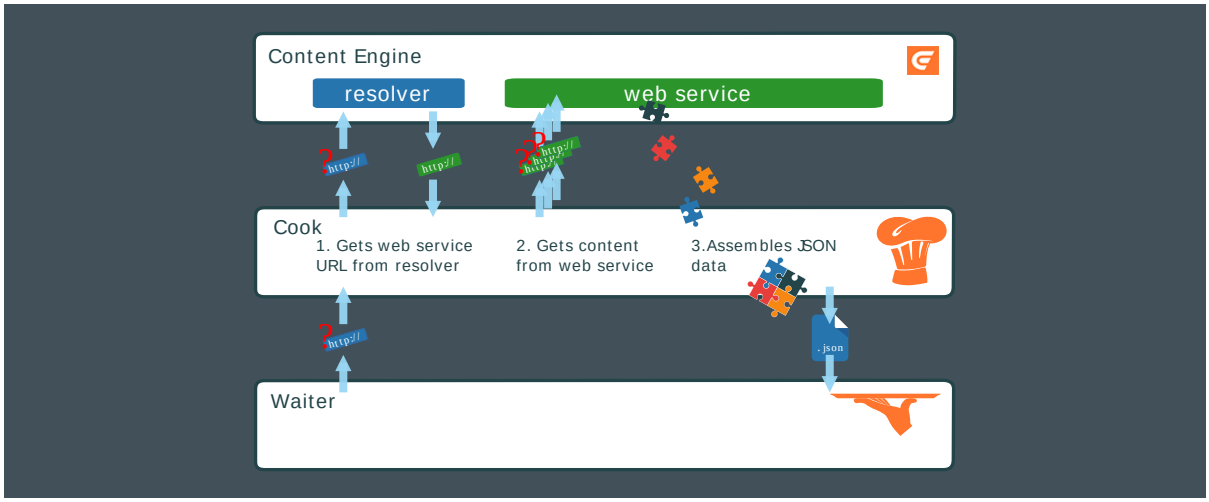
1.1.2 What is Twig?

Twig is a popular templating engine for PHP, and is fully supported by Patternlab.io.

1.2 CUE Front for Developers

If you are a back-end developer, then you will mainly be interested in the Cook. The Cook is a node.js application that supplies the content requested by the Waiter and/or other front-end components. The Waiter forwards each page request made by a client directly to the Cook. The Cook is responsible for assembling a response that contains **all** the content that the Waiter will need to render the

page. Retrieving content requires the Cook to make multiple requests to the Content Store, but this complexity is hidden from the Waiter.



When the Cook receives a request from the Waiter, it:

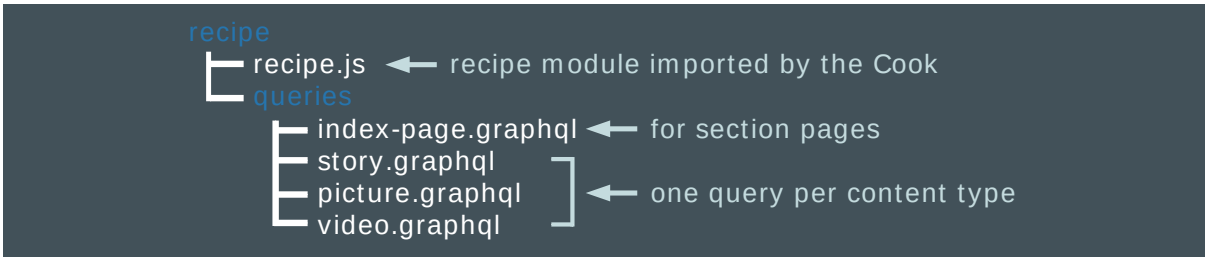
1. Sends the request URL to a Content Store web service called **`resolver`**. The resolver converts this external "pretty" URL to an internal web service URL
2. Sends a request to the returned web service URL. The Content Store web service returns data in the form of Atom XML resources. In order to obtain all the information needed to respond to the Waiter's request, the Cook will usually need to follow links embedded in the returned Atom data, and send several requests to the web service.
3. Assembles the information returned from the Content Store into a JSON structure.
4. Returns the JSON structure to the Waiter.

In order to be able to perform these steps, the Cook needs to know what data the client will need to be able to render the requested page. A content item can have many different fields - which ones is the Waiter actually going to render on the page? A content item can be related to many other content items in a variety of ways - which ones are to be included or linked to on this page, and, which of their fields is required? This information is provided in a **recipe**. A recipe defines:

- The information the Waiter needs to render specific page types
- How the Waiter would like the information for each page type to be organized (that is, the required JSON structure)

Your main responsibility as a developer, therefore, is the creation of a recipe that correctly defines the information to be supplied to the Waiter.

1.2.1 What is a Recipe?



A recipe is a Javascript code module used by the Cook to enable it to retrieve information from the Content Store and/or other sources, and make it available in a useful form to the Waiter. It consists of:

- **`recipe.js`**, a small controller for the recipe. Most of the actual recipe functionality is provided by NPM extension modules imported by **`recipe.js`**. In the delivered system, these extensions are all downloaded from CCI Europe's NPM repository, **`npm.escenic.com`**, but you can extend the recipe by creating your own extensions.
- A set of application-specific GraphQL **queries** that specify for each type of page on the site:
  - The content to be supplied to the Waiter
  - How the content supplied to the Waiter is to be organized and named

The recipe also requires access to a publication-specific **GraphQL schema** in order to provide a context for the GraphQL queries. The GraphQL schema consists of a set of publication-specific Javascript files that are called by the recipe, enabling the Cook to navigate the publication structure and retrieve data from it.

The CUE Front start pack includes a script called **`update-schema.sh`** that can automatically generate the GraphQL schema files for any CUE publication (see section 3.1). This means that creating a recipe for a new publication or family of related publications is in many cases just a matter of creating a set of suitable GraphQL queries.

In some cases it may not be possible to produce the required output using GraphQL alone. Possible reasons for this include:

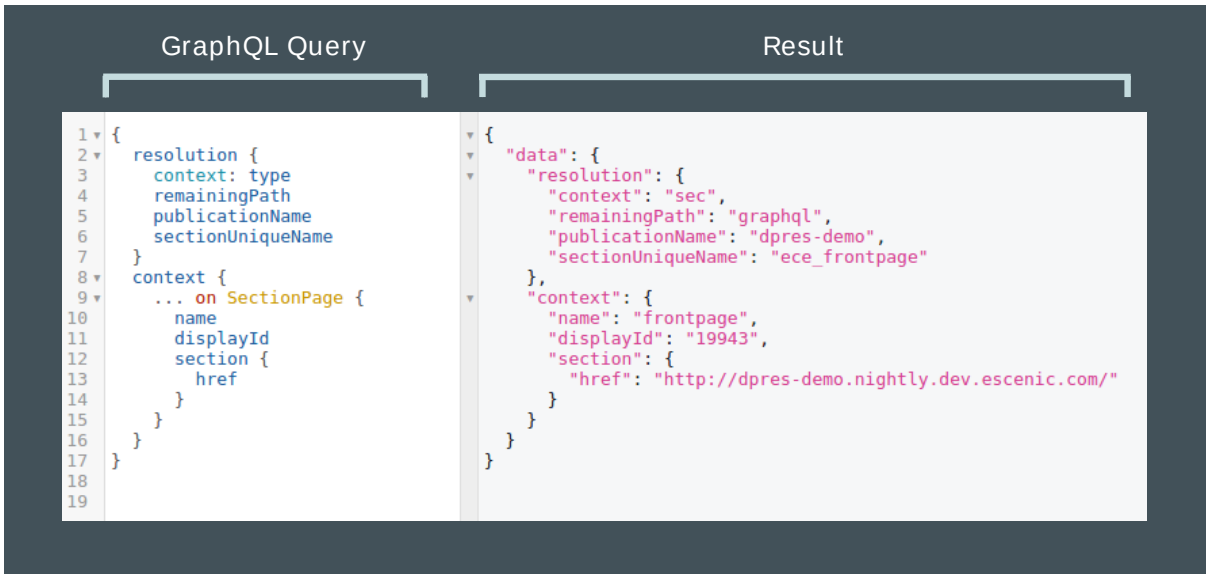
- The Waiter requires the output JSON data to be organized in a different way than the default output (which reflects the Content Store's internal structure). GraphQL allows simple modifications to the output structure, such as omitting elements and renaming, but not complex reorganization.
- The Waiter requires data from sources other than the Content Store to be incorporated into the structure, such as data from an external sports results service, or stock market data.

In such cases the default recipe supplied with the CUE Front start pack can be extended by writing an extension of your own. For more about this, see section 6.1.

1.2.2 What is GraphQL?

GraphQL is a query language that supports the definition of complex queries – sufficiently complex that a single query can be used to retrieve all the content needed to render the front page of a typical CUE publication. The result of a GraphQL query is a JSON data structure that can be passed to a templating system for rendering as HTML.

GraphQL queries are very specific about what is to be retrieved: only those items of data that are specifically requested are retrieved. This means that a GraphQL query tends to look very similar to the result it produces – it has the same "shape":



The Cook includes GraphiQL, a browser-based GraphQL interface that lets you interactively explore a dataset (in this case, your publication) by editing a GraphQL query and seeing the results in real time. The query and the results it produces are displayed side-by-side in the browser.

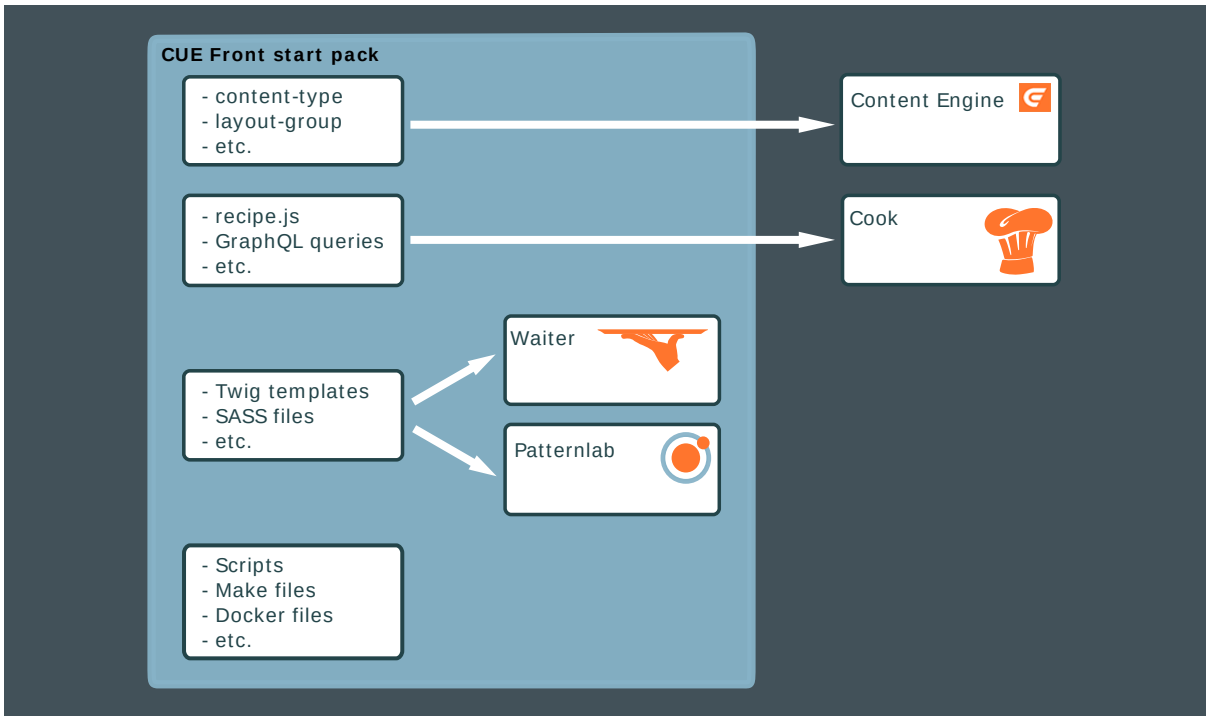
For more about this, see section 3.2.

1.2.3 What Does the Cleaver Do?

The Cleaver is an auxiliary service that handles images for the Cook. Images in CUE publications can include crop information that specifies what aspect ratio the image should have, and what part of the base image should actually be rendered in the specified location. When the Cook receives a request for an image from the Waiter, it forwards the request to the Cleaver, appending the required crop information as URL parameters. The Cleaver then retrieves the base image from the Content Store, carries out any required crop operations and returns the cropped image to the Cook. The Cook then serves this image to the Waiter. The Cleaver maintains a cache for the images it downloads from the Content Engine in order to avoid unnecessary network traffic.

This whole process is automatic and requires no intervention. Once the Cook and Cleaver are correctly configured, the Cleaver can be regarded as a "black box".

1.3 The CUE Front Start Pack



The CUE Front start pack is a combined demo system and start pack. You can use it as both a learning tool and as a starting point for your own presentation layer implementations. **`cue-front-start-pack`** consists of:

- The Waiter
- A simple demo publication that you can upload to the Content Store
- A recipe and set of GraphQL queries for retrieving page content from the demo publication
- A set of Twig templates, SASS files and other design assets for rendering the page content retrieved from the demo publication
- Patternlab, a "living style guide" that you can use to organize, view and work with Twig templates
- Scripts, make files and Docker files to simplify the process of getting started.

`cue-front-start-pack` is made available as a tarball that you can download from the CCI Europe Maven repository and modify to suit your requirements.

2 Getting Started

How much you need to do to get started with CUE Front depends on what you're going to do with it, and whether or not you have access to any existing CUE Front components. The following sections contain two "quick start" guides for Docker-based installations: one for a full-stack test/development installation and a simpler guide for designers who will be accessing an existing Cook installation.

Quick start for test/development
: This is the quickest way to install a complete CUE Front stack. All the components are installed in Docker containers and are pre-configured to work together correctly. It's the recommended starting point, since it gives you a complete, correctly configured system to explore and play around with. It also means you can install CUE Front on Mac and Windows machines, not only on Linux. Note, however, that some organizations have IT policies that disallow the use of virtualization technology on Windows machines, in which case you will not be able to install CUE Front in this way.

Quick start for designers
: If you are a designer or front-end designer working in an organization with an existing CUE Front installation, then you probably don't need to run all the CUE Front components on your computer. You will probably only want to use the Waiter and Patternlab.io, and connect the Waiter to an existing Cook installation. This guide tells you how to install and configure your Docker containers for this kind of usage.

This section does not discuss installation or configuration of the **Fridge**, since the Fridge is an optional component that is not needed in the "getting started" phase. The Fridge is a small web server/proxy that serves static content from a file system folder and can be used for two different purposes:

- Offline template development
- Caching in production systems

For information about the Fridge's different uses and how to install and configure it, see chapter 4.

For instructions on how to install the CUE Front components directly in one or more computers without the use of Docker, see chapter 8. If you use this method then you are restricted to installing the components on Linux machines. Installing CUE Front in this way is more difficult as the various components must be configured to work together correctly. You probably don't want to install CUE Front this way unless you are a system administrator installing components on production/test hosts or you are prevented from using the Docker method by corporate policies on virtualization technology. If you don't have a clear reason for installing CUE Front in this way, then we suggest you use the Docker-based method.

2.1 Quick Start for Test/Development

The general procedure is:

1. Install Docker on your machine – see section 2.1.1
2. Download the CUE Front start pack and unpack it – see section 2.1.2
3. Upload the demo publication to your Content Store if necessary – see section 2.1.3

4. Install the CUE Front components in Docker containers – see section 2.1.4
5. Run the Docker containers – see section 2.1.5

2.1.1 Installing Docker

The installation method for Docker is platform-dependent.

2.1.1.1 Installing Docker on Ubuntu

These instructions are based on the use of Ubuntu 16.04 LTS.

Before you start, make sure that your Ubuntu installation includes the **`zip`** command. If it doesn't, install it as follows:

```
sudo apt-get update
sudo apt-get install zip
```

You need to install both **`docker`** itself and an additional tool called **`docker-compose`**. There are **`docker.io`** and **`docker-engine`** packages in the Ubuntu repositories, but they contain old versions and must not be used. Instead, follow the instructions given on the following pages:

- Installing Docker CE on Ubuntu
- Installing docker-compose (make sure you select the **Linux** tab on this page)

You can now continue by following the instructions in section 2.1.2.

2.1.1.2 Installing Docker on Windows

The best way to get Docker on Windows is to install Docker Toolbox. Docker Toolbox can be installed on any 64-bit versions of Windows 7, 8 or 10. If you are using Windows 10 Pro, then you can use Docker for Windows instead. Both products work by running the Docker containers in a lightweight Linux system which itself runs inside a virtual machine. The main difference between the two products is that Docker Toolbox uses VirtualBox to host the Linux virtual machine, while Docker for Windows uses Microsoft's Hyper-V. VirtualBox and Hyper-V cannot co-exist on the same machine, so if you already use VirtualBox for other purposes, then you should stick to Docker Toolbox.

The following procedure describes how to install CUE Front using Docker Toolbox:

1. Download and install Docker Toolbox. The Docker Toolbox package **includes** VirtualBox, so you don't need to install it separately.
2. Double-click the **Docker Quickstart Terminal** icon installed on your desktop. This opens a terminal window from which you can install, start and stop Docker containers. This window is actually running a **`bash`** shell (the default command shell used in Linux), which means that from this point on, installation is very similar to installation on Ubuntu.
3. At the top of the Docker Quickstart Terminal is a line something like this, telling you the IP address of the virtual machine that the Docker containers will run in:

   ```
   docker is configured to use the default machine with IP 192.168.99.100
   ```

   Copy or make a note of the IP address, as you will need it later.

If you enter this command in the Docker Quickstart Terminal:

```
pwd
```

You will see the full Windows path of Docker's "home folder". This is useful to know so that you can find the folder in Windows Explorer, if necessary.

2.1.1.3 Installing Docker on Mac

Download and install Docker for Mac as described here. Open a command terminal and continue as described in section 2.1.2.

2.1.2 Getting the CUE Front start pack

Download and unpack the CUE Front start pack. On Ubuntu and Macs, the following commands will unpack it in your home folder, On Windows they will unpack it in the Docker home folder.

```
cd
curl -O https://user:password@maven.escenic.com/com/escenic/cook/cue-front-start-pack/1.4.0-3/cue-front-start-pack-1.4.0-3.tar.gz
tar -xzvf  cue-front-start-pack-1.4.0-3.tar.gz
rm cue-front-start-pack-1.4.0-3.tar.gz
ln -s cue-front-start-pack-1.4.0-3 cue-front
cd cue-front
```

where *username* and *password* are your CCI Europe credentials. If you don't have a username and password, please contact CCI Europe support.

If you are installing on a Mac, make sure you unpack the start pack either in your home folder or in one of its subfolders. Otherwise you may have problems later syncing the **`templates`** folder (because it cannot be mounted by the containers).

Making a git Repository

If you intend to use the CUE Front start pack as the basis for your own CUE Front project then you should commit the **`cue-front`** folder to a source control repository now before you have made any changes, and tag it. This will ensure you have a full record of everything you do, and can easily retrace your steps if necessary. You can create a **`git`** repository for your project and tag the starting point with the following commands:

```
git init
git add .
git commit -m "Starting the CUE Front journey"
git tag baseline
```

for more about this, and the development process in general, see section 3.6.

If you are just downloading the start pack for demonstration / test purposes, then you can skip this step.

2.1.3 Uploading the Demo Publication

If the CUE Front demo publication (called **Tomorrow Online**) is not already installed at your site, and you don't have access to a copy running anywhere else, then you will need to upload it to your Content Store.

Create the publication by entering the following command:

```
make dist -C publication
```

You will then find two versions of the publication here:

cue-front/publication/dist/tomorrow-online.zip
cue-front/publication/dist/tomorrow-online-with-content.zip

> If you are on Windows then you need to prefix the above path with the path of your Docker "home" folder if you want to find the demo publication from outside the Docker Quickstart Terminal (for example, using Windows Explorer). You can find the home folder path by entering the following commands in your Docker Quickstart Terminal:
>
> ```
> cd
> pwd
> ```

Upload the demo publication to your Content Store in the usual way. If you don't know how to do this, you will find instructions here. Note that:

- You only need to follow steps 1 - 7, the remaining steps are not required.
- In step 6, make a note of the publication name and administrator password you select, as you may want to enter them again when installing CUE Front.
- Don't worry that the instructions specify the use of a **.war** file – the supplied **.zip** file will work.

2.1.4 Installing the CUE Front Components

A setup tool is included with the CUE Front start pack. It displays a series of prompts asking what components you want to install, and some details about how you want to install them. From your responses it generates the configuration files needed by each component, downloads the components from the CUE Front SW repository and builds the Docker containers needed to run them.

To install a full set of CUE Front components using the setup tool:

1. Make sure you are in the **cue-front** folder:

   ```
   cd path/cue-front
   ```

2. To build the setup tool's Docker container, enter:

   ```
   docker-compose -f setup.yml build
   ```

3. To run the setup tool, enter:

   ```
   docker-compose -f setup.yml run setup add configuration-set
   ```

   where *configuration-set* is just a name for the configuration you are going to create — **myconfig**, for example. The setup tool displays a series of prompts in the terminal window. The first two prompts are:

   Enabled services
   Press **Up Arrow** a few times and then press **Space** to select **all**. Then press **Enter** to move on.

   Configuration
   You want **Quick**, which is the default choice, so just press **Enter**.

   Docker host OS
   Press **Space** to select your operating system and then press **Enter**.

The remaining prompts require you to either enter a value or just press **`Enter`** to use the default. You can accept the defaults in most cases.

Escenic Username
The user name to be used for accessing the Escenic repository.

Escenic Password
The password to be used for accessing the Escenic repository.

[fridge] Listen to port
The port number the Fridge will listen to.

[cook] Listen to port
The port number the Cook will listen to.

[cook] Escenic Content Engine domain with port
The Content Engine domain name and port number to which the Cook will send web service requests.

[cook] Access credentials username
The CUE username the Cook is to use when accessing the Content Engine. For demo/test purposes it is most convenient to use the publication's admin username, which is always *publication-name*`_admin`. For example, If you called the publication **`tomorrow-online`** when you uploaded it to the Content Store (see section 2.1.3), then the admin username is **`tomorrow-online_admin`**.

> Using the admin user for CUE Front is **not** a good idea for production systems. In a production system you should create a special user for CUE Front that has read-only access to all your publication's sections and content types. If you want to be able to support cross-publishing, then this user must also have read access to all the publications from which content might be selected.

[cook] Access credentials password
The password for the user you have specified above.

[cook] Use proxy
If you want the Cook to use the Fridge as a proxy, set this to **`true`**.

[waiter] cookBaseURL
The Cook domain name and port number to which the Waiter will forward incoming requests.

[waiter] port
The port number the Waiter will listen to.

[waiter] Publication name
The name of the CUE publication you want to publish using CUE Front (Tomorrow Online in this case). This is the name you specified when uploading Tomorrow Online to the Content Engine (see section 2.1.3).

[waiter] Publication host name
The host name the publication will be published on.

[waiter] Use proxy
If you want the Waiter to use the Fridge as a proxy, set this to **`true`**.

4. Once you have answered all the prompts, the setup tool verifies your responses, generates a set of configuration files, checks for errors, and if all is OK downloads the requested components and builds Docker containers for them.
5. To set the default configuration, enter:

```
docker-compose -f setup.yml run setup use -d configuration-set
```

where *configuration-set* is the name of the configuration set you created in step 3.

6. If your Content Store is running in a virtual machine on your PC and is accessed via a host name specified in your PC's **hosts** file, then you need to open the file **docker-compose.yml** and add an **extra_hosts** property to both the Cook and Cleaver sections of the file. The **extra_hosts** properties let you provide the Cook and Cleaver containers with a host name mapping for the Content Store, since they do not have access to your PC's **hosts** file. So if you have the following entry in your **hosts** file to set up a host name for your Content Store VM:

   ```
   192.168.56.101 engine.local
   ```

   Then you would need to add the following lines (highlighted in **bold**) to your **docker-compose.yml** file:

   ```
   services:
     cleaver:
       ... (lines omitted) ...
       extra_hosts:
         - "engine.local:192.168.56.101"

     cook:
       ... (lines omitted) ...
       extra_hosts:
         - "engine.local:192.168.56.101"
   ```

7. To build the Docker containers for the CUE Front components, enter:

   ```
   docker-compose build
   ```

CUE Front is now installed and ready to run. The setup tool has created a set of configuration files for the CUE Front components, which you will find in the **setup/configs/***configuration-set* folder. If you want to make configuration changes later, you can edit the configuration using the **setup edit** command, or create a different configuration set with a different name. The setup tool has an advanced mode which allows you to configure the components in more detail. For more information about all these options, see chapter 9.

2.1.5 Starting CUE Front

To start CUE Front, enter:

```
docker-compose up -d
```

where *configuration-set* is the name of the configuration set you created.

A sequence of output messages is displayed as the various Docker containers are created and the CUE Front services are started:

```
Creating network "cuefrontstartpack12019_default" with the default driver
Creating cuefrontstartpack12019_fridge_1 ...
Creating cuefrontstartpack12019_cleaver_1 ...
Creating cuefrontstartpack12019_rsync_1 ...
Creating cuefrontstartpack12019_cleaver_1
Creating cuefrontstartpack12019_fridge_1
Creating cuefrontstartpack12019_fridge_1 ... done
Creating cuefrontstartpack12019_cook_1 ...
Creating cuefrontstartpack12019_rsync_1 ... done
```

```
Creating cuefrontstartpack12019_browsersync_1 ...
Creating cuefrontstartpack12019_styles_1 ...
Creating cuefrontstartpack12019_styles_1
Creating cuefrontstartpack12019_styles_1 ... done
Creating cuefrontstartpack12019_styleguide_1 ...
Creating cuefrontstartpack12019_waiter_1 ...
Creating cuefrontstartpack12019_styleguide_1
Creating cuefrontstartpack12019_waiter_1 ... done
Creating cuefrontstartpack12019_loadBalancer_1 ...
Creating cuefrontstartpack12019_styleguide_1 ... done
```

If you get problems at this point, the most likely reason is that you entered the **`docker-compose up`** command in the wrong folder. You must be in the **`cue-front`** root folder when you enter any **`docker-compose`** command (the folder that contains the **`docker-compose.yml`** file). **`docker-compose`** will output an error message explaining the problem.

Assuming all went well, start a browser — you should be able to find the services listed below. When entering the URLs, you need to replace *cue-front-host* with:

- **`localhost`** on Ubuntu
- The IP address of the Docker virtual machine (**`192.168.99.100`**, for example) on Windows or Mac

The demo publication

At **`http://`***cue-front-host*`:8100/`** you should find the front page of the demo publication.

The Cook

At **`http://`***cue-front-host*`:8101/`** you should find the Cook. All you will see at this address is:

```
{}
```

If, however, you add the name of the demo publication (plus a final slash) to the URL – **`http://`***cue-front-host*`:8101/tomorrow-online/`** - then you will see the JSON data from which Waiter generates the front page:

```
{
  data: {
    resolution: {
      context: "sec",
      remainingPath: "",
      publication: {
        name: "tomorrow-online",
        features_raw: "",
        features: [ ]
      },
      section: {
        name: "Home",
        uniqueName: "ece_frontpage",
        href: "http://vagrant:8080/tomorrow-online/",
        parameters: [ ]
      }
    },
    headerMenu: [
...etc...
```

If you add **`edit`** to this URL (that is, if you enter **`http://`***cue-front-host*`:8101/tomorrow-online/edit`**), then you will see the GraphQL query that is used to retrieve the page from the Cook displayed in the Cook's GraphiQL interface. For more about this, see section 3.2.

The Cleaver

At **`http://`***cue-front-host*`:8102/`** you should find the Cleaver. All you will see is:

```
Cleaver is running...
```

Patternlab

At **`http://`***cue-front-host*`:8103/`** you should find the Patternlab style guide. You can use this to explore all the design components from which the demo publication is built. For more about this, see section 3.3.

2.1.6 Scaling CUE Front with Docker

The standard Docker start-up command:

```
docker-compose up -d
```

starts up a single instance of each CUE Front component running in its own Docker container. One of the components that is started, however, up is a load balancer, capable of distributing requests among multiple instances of the same component:

```
Creating cuefrontstartpack12019_loadBalancer_1 ...
```

The load balancer is configured to handle incoming requests directed to the Waiter, Cook, Cleaver and the Fridge. When only one instance of each component is available, the load balancer simply passes requests straight through.

You can, however, start up multiple copies of the CUE Front components by specifying **`--scale`** options with the start-up command. For example:

```
docker-compose up -d --scale cook=4 --scale cleaver=2
```

starts 4 instances of the Cook, 2 instances of the Cleaver and one of the Waiter and the Fridge. The load balancer distributes requests evenly between multiple instances of the same component. Scaling CUE Front in this way can:

- Improve performance by making more threads available for handling requests
- Improve stability since if one instance of a service fails, other instances are available to take over

2.1.7 Managing the CUE Front Containers

To stop all the CUE Front services without closing the Docker containers in which they run, enter:

```
docker-compose stop
```

You will then see a series of messages as each Docker container is stopped:

```
Stopping cuefrontstartpack1208_waiter_1 ... done
Stopping cuefrontstartpack1208_cook_1 ... done
Stopping cuefrontstartpack1208_styleguide_1 ... done
Stopping cuefrontstartpack1208_styles_1 ... done
Stopping cuefrontstartpack1208_cleaver_1 ... done
Stopping cuefrontstartpack1208_rsync_1 ... done
Stopping cuefrontstartpack1208_fridge_1 ... done
```

You can then restart the CUE Front by entering:

```
docker-compose start
```

This time, CUE Front will start faster as the containers do not need to be created first.

To stop CUE Front and remove the containers, enter:

```
docker-compose down
```

To start CUE Front again now, you will need to enter:

```
docker-compose up -d
```

To restart one of the CUE Front services while CUE Front is running, enter:

```
docker-compose restart service-name
```

To restart the Waiter, for example, enter:

```
docker-compose restart waiter
```

If you want to examine what is going on inside one of the containers (explore the file system, for example), you can start a Bash shell inside the container by entering:

```
docker-compose exec service-name bash
```

When you are finished doing what you want to do inside the container, you can return to your main shell by entering **`exit`** or pressing **`Ctrl-d`**.

If you want to be able to see the log messages output by the CUE Front services, open a second terminal window, **`cd`** to the **`cue-front`** folder and enter the following command after starting CUE Front:

```
docker-compose logs -f
```

All log messages will then be displayed in this terminal. To stop the display, just press **`Ctrl-c`**.

2.2 Quick Start for Designers

The general procedure is:

1. Install Docker on your machine as described previously in section 2.1.1
2. Download the CUE Front start pack and unpack it as described previously in section 2.1.2
3. Install the CUE Front components in Docker containers as described below in section 2.2.1.
4. Run the Docker containers as described below in section 2.2.2

2.2.1 Installing for Designers

The basic installation procedure for designers is the same as the general Docker installation procedure described in section 2.1.4. Step 3, however is different. When you get to this step you should answer the setup tool's prompts as follows:

Enabled services
Instead of selecting **all**, you should select the following individual services: **waiter**, **rsync**, **styleguide** and **styles**.

Configuration
Select **Quick**.

You can accept the defaults of all the remaining prompts, except for:

[waiter] Publication name
The name of the CUE publication you want to publish using CUE Front (Tomorrow Online in this case).

[waiter] Cook base URL
The URL of the Cook you want to use (including port number and closing slash). For example, **`http://cook.myserver.com:8101/`**.

2.2.2 Starting the CUE Front Design Tools

To start only the CUE Front components needed by designers, enter:

```
docker-compose up -d waiter styles styleguide rsync
```

A sequence of output messages is displayed as the various Docker containers are created and the CUE Front services are started:

```
Creating cuefrontstartpack1208_rsync_1
Creating cuefrontstartpack1208_styles_1
Creating cuefrontstartpack1208_styleguide_1
Creating cuefrontstartpack1208_waiter_1
```

If you get problems at this point, the most likely reason is that you entered the **`docker-compose up`** command in the wrong folder. You must be in the **`cue-front`** root folder when you enter any **`docker-compose`** command (the folder that contains the **`docker-compose.yml`** file). **`docker-compose`** will output an error message explaining the problem.

Assuming all went well, start a browser — you should be able to find the services listed below. When entering the URLs, you need to replace *cue-front-host* with:

- **`localhost`** on Ubuntu
- The IP address of the Docker virtual machine (**`192.168.99.100`**, for example) on Windows or Mac

The demo publication
At **`http://`***cue-front-host*`:8100/`** you should find the front page of the demo publication.

Patternlab
At **`http://`***cue-front-host*`:8103/`** you should find the Patternlab style guide. You can use this to explore all the design components from which the demo publication is built. For more about this, see section 3.3.

See section 2.1.7 for information on how to stop the CUE Front services you have started.

3 Using CUE Front

The default Waiter supplied with CUE Front is a PHP application that uses the Twig templating library to merge HTML templates with JSON data supplied by the Cook. It includes a set of demo templates designed to work with a demo publication (also supplied). The Waiter also includes patternlab.io, a PHP application that supports atomic design. Atomic design is a design methodology that provides a framework for breaking web site designs down into re-usable components. The supplied demo templates are structured using atomic design, and can be viewed from the Patternlab.io interface.

You can create a CUE Front presentation layer for your own publication based on the supplied demo as follows:

1. Install the CUE Front start pack as described in chapter 2.
2. Run the start pack's **`update-schema.sh`** script to replace the demo publication schema with your publication's schema. See section 3.1 for further information.
3. Modify the supplied GraphQL queries to work with the new GraphQL schema.
4. Modify the supplied Twig templates to work with the JSON structures output by your GraphQL queries (or replace them with a completely new set of templates).
5. Continue modifying the supplied Twig templates until they produce the output you require.

You don't necessarily need to perform the tasks in this order. In many organisations, steps 4 and 5 will be carried out by different people from steps 2 and 3, so it might then make sense to perform them in parallel. You could also work backwards by creating a design first, then defining the JSON structures needed to support that design, and then creating the GraphQL queries needed to produce those structures. In reality, wherever you start, the process will more than likely be an iterative one in which parallel adjustments need to be made in GraphQL queries, Twig templates and possibly also the publication definition (**`content-type`** and **`layout-group`** resources).

However, it's probably easiest to understand how CUE Front works by following the data flow from the publication structure to the rendered page.

3.1 Updating a GraphQL Schema

The Cook needs a GraphQL schema describing the structure of the content it has access to – that is, the structure of the publication. The CUE Front start pack includes a schema for the demo publication in its **`schema`** folder. If you want to create a presentation layer for your own publication, then the first step is to replace these files with files that describe your publication.

A shell script for generating new schema files based on any CUE publication is included in the start pack. In order to use the script you must have access to the publication you want to work with. In the **`cue-front`** folder, enter:

```
docker-compose exec cook bin/update-schema.sh publication-name user:password http://
my-escenic.com:8080/webservice/
```

or, if your Content Store installation includes CUE Live, enter:

```
docker-compose exec cook bin/update-schema.sh publication-name user:password http://
my-escenic.com:8080/webservice/ http://my-escenic.com:8080/live-center-editorial/
```

The script's parameters are:

- The name of the publication
- Credentials for accessing the publication
- The URL of the Content Store's webservice. The URL **must** be terminated with a **/**.
- The URL of the CUE Live presentation webservice. The URL **must** be terminated with a **/**. This parameter should only be supplied if your Content Store installation includes CUE Live

The **update-schema.sh** script sends a series of requests to the specified web service(s), and retrieves the information it needs to generate a complete description of the publication structure in the form of Javascript schema files. It writes these files to the **cue-front/schema** folder. You will see that it generates an **index.js** for section pages, one **.js** file for each content type defined in the publication's **content-type** resource and one **.js** file for each group defined in the publication's **layout-group** resource. If CUE Live is installed, then it also creates a **schema/entryTypes** folder containing a **.js** file for each CUE Live entry type.

Important notes

- If your Content Store is running in a virtual machine exposed on **localhost**, you cannot use **localhost** in the web service parameters supplied to **update-schema.sh**, as this will be interpreted to mean the Cook container's local host rather than your PC. You need to specify your computer's actual IP address instead. You should, for example specify **http://***ip-address***:8080/webservice/** instead of **http://localhost:8080/webservice/**.
- In order for your changes to take effect, you must restart the Cook after updating the schema:

  ```
  docker-compose restart cook
  ```

- The schema must be updated not only when setting up CUE Front to handle a new publication, but also any time you modify the publication's **content-type** resource or **layout-group** resource. If CUE Live is installed at your site, then you must also update the schema after modifying the **entry-type** resource. First upload the modified resources to the Content Store, and then run **update-schema.sh** using one of the commands listed above.

3.2 Working with GraphQL

The first thing you need in order to be able to display content on a page is a JSON structure that contains all the data you need. The Cook obtains this data by executing a GraphQL query that retrieves the required data from the Content Store's web service. You can see how this works by opening a browser and submitting a request directly to the Cook instead of to the demo publication URL.

If you have installed the CUE Front components as described in section 2.1 or chapter 8, then the Waiter will be listening for requests on port 8100, and the Cook will be listening for requests on port 8101. This means the URL of the demo publication's front page is **http://***cue-front-host***:8100/**. If you want to see the Cook's version of the same page, simply change the port number in the URL to 8101 and add the name of the publication: **http://***cue-front-host***:8101/tomorrow-online/** (make sure you include the final slash). The Cook will then return the JSON data from which Waiter generates the front page:

```
{
  data: {
    resolution: {
      context: "sec",
      remainingPath: "",
      publication: {
        name: "tomorrow-online",
        features_raw: "",
        features: [ ]
      },
      section: {
        name: "Home",
        uniqueName: "ece_frontpage",
        href: "http://vagrant:8080/tomorrow-online/",
        parameters: [ ]
      }
    },
    headerMenu: [
...etc...
```

A much more useful way to view the JSON data is to use the Cook's GraphiQL (section 3.2.1) interface.

> If you have installed CUE Front on a Windows machine using Docker, then replace *cue-front-host* in the above URLs with the IP address of your Docker virtual machine. Otherwise, replace it with **`localhost`**.

3.2.1 The GraphiQL Editor

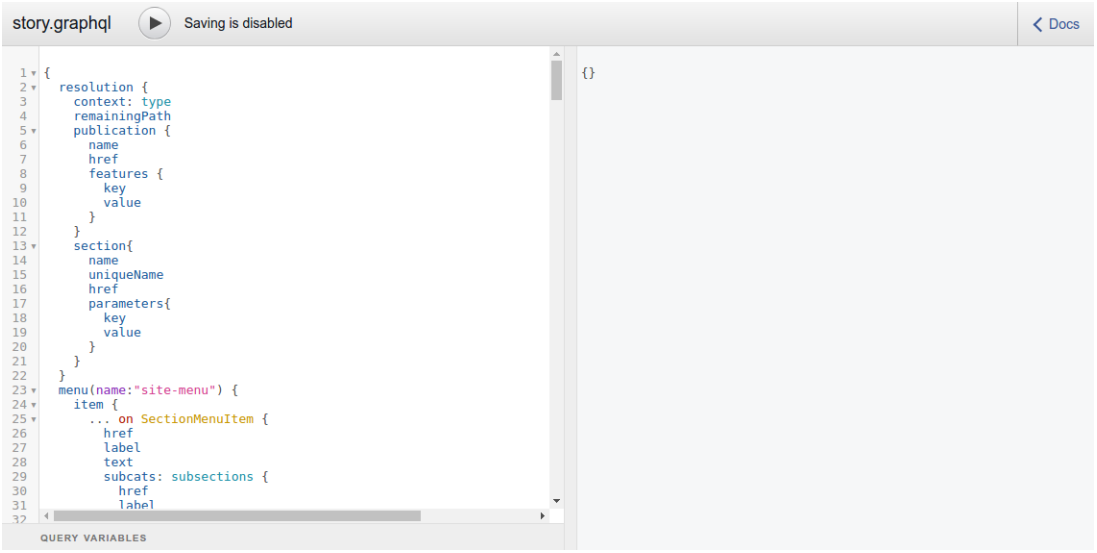
To view JSON data returned by the Cook in the GraphiQL editor, all you need to do is append **`edit`** to the URL you submit to the browser. Instead of

```
http://cue-front-host:8101/tomorrow-online/
```

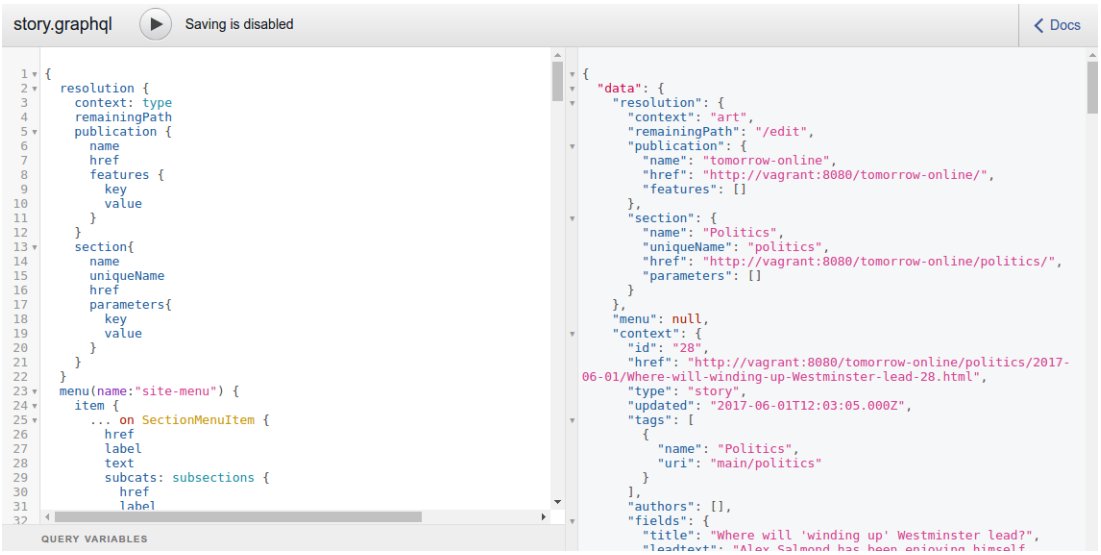
for example, enter:

```
http://cue-front-host:8101/tomorrow-online/edit
```

Now, instead of simply displaying the JSON data normally returned by the Cook, the browser displays a vertically split screen, on the left side of which is the GraphQL query that the Cook would use to retrieve the page data:



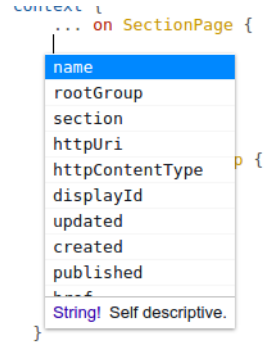
If you click on the ▶ button above the query, then the result of executing the query is displayed on the right side of the screen:



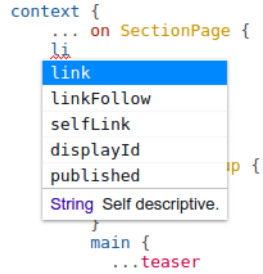
With the query and the results displayed side-by-side like this, it's relatively easy to see the relationship between them. GraphiQL is not just a viewer, it's an editor as well. If you edit the query displayed on the left and click the ▶ button again, then you will see the results of your modification on the right. Try simply deleting a field – **`uniqueName`** on line 15, for example. If you then execute the query again, you will see that the corresponding field disappears from the output on the right. Replace the field and re-execute, and you will see that the deleted field reappears in the JSON output.

The editor offers you a lot of assistance while you are editing, including code completion. The Cook knows your publication's data structure, so it can tell you what fields are available at any point in the

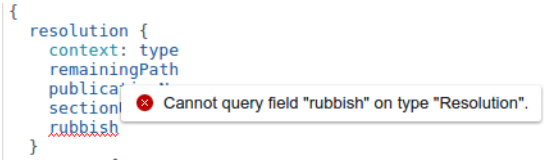
query. Try inserting a line somewhere in the query and pressing **`Ctrl-Space`**: the editor will display a context menu listing the names of all the field names you can insert at this point in the query:



If instead of pressing **`Ctrl-Space`** you start typing, then it will display a shorter list containing valid names that match what you have entered:



The editor underlines any invalid content in the query in red, and will display an error message if you hover the mouse over the invalid text:



In addition to all this, GraphiQL also provides a help function that you can use to explore your publication's data structure. To display it, click on the **Docs** link in the top right corner of the GraphiQL window. You can use this to browse the publication's data structure, find the data types of particular fields and so on. For fields that have enumeration data types, you can list all possible enumeration values.

3.2.1.1 Saving Your GraphQL changes

By default, the GraphiQL editor does not allow you to save any changes you make. You can, however, configure the editor to display a **Save** button:

Clicking on this button will save any changes you have made. The changes are saved directly into the **`cue-front/recipe/queries`** folder used by the Cook, so the saved changes will take immediate effect. If, for example you remove a field from the JSON data output by the query, then that content will disappear from any pages in which it is used on the web site. Conversely, any fields you add to the output will immediately be available for use by the front end.

To enable the GraphiQL editor's **Save** button:

1. Open **`cook-config.yaml`** for editing (see section 2.1.4 for more information about this).
2. Replace the following line:

   ```
   editor: enabled
   ```

   with:

   ```
   editor:
       allow-save: true
   ```

> You don't have to use the Cook's built-in GraphiQL editor to edit your GraphQL queries. The queries are stored in the **`cue-front/recipe/queries`** folder, and you can use whatever editor you choose to edit them. Some IDEs and programmer's editors include syntax support for GraphQL.

3.2.2 Understanding CUE Front GraphQL Queries

GraphQL is a powerful language for retrieving information from hierarchical data structures such as CUE publications. You can use it to retrieve all the information you want to display on a page in a single query. Not only can you retrieve everything you need in one query, you can also easily ignore all the information you don't need, so that only useful content is downloaded to the client. For a general introduction to GraphQL, see this tutorial.

In order to retrieve content from the Content Store, the Cook needs to be supplied with a recipe. A recipe is a Javascript module that controls the execution of a set of GraphQL queries. In the CUE Front start pack, the recipe is located in **`cue-front/recipe/recipe.js`**. The recipe in turn uses a set of GraphQL queries that specify exactly what is to be retrieved. These queries must be located in the **`cue-front/recipe/queries`** folder. The folder must contain:

- One query for each content type in the publication, called *content-type* **`.graphql`**
- one query for all section pages called **`index-page.graphql`**

Since the demo publication currently has only three content types, **`story`**, **`picture`** and **`video`** the delivered **`cue-front/recipe/queries`** folder contains the following queries:

- **`index-page.graphql`**
- **`picture.graphql`**
- **`story.graphql`**
- **`video.graphql`**

The query displayed in GraphicQL at **`http://localhost:8101/tomorrow-online/edit`** is the **`index-page.graphql`** query. Here is a brief explanation of its content:

If you click on the **Docs** link in the top right corner of the GraphiQL window, you will see that the root of the data structure that you can interrogate using GraphQL is called **`query`**, and it is an object of type **`Query`**. If you click on the **Query** link, you will see that a **`Query`** is composed of 3 fields:

`nop`
: Not used.

`resolution`
: This field contains information about the current request that has been returned from the Content Store's **resolver**. The resolver is a web service that converts public-facing "pretty" URLs like **`http://my-escenic.com/news/2016-12-02/Some-Exciting-Story.html`** to internal Content Store web service URLs like **`http://my-escenic.com/webservice/escenic/content/206246`**. **`resolution`** is a **`Resolution`** object that contains the following fields:

  `type`
  : **`art`** or **`sec`**, according to whether the requested page is a content (article) page or a section page

  `remainingPath`
  : When the resolver resolves a URL, it starts from the left hand end of the string and resolves as much as it can. If there is anything left at the end of the string, it is returned in this string. The remaining path might contain a list of URL parameters, for example, or additional URL segments that can be used by the page rendering application to modify the output in some way.

  `publicationName`
  : The name of the current publication (**`tomorrow-online`** in the case of the demo publication).

  `sectionUniqueName`
  : The unique name of the current section, or current content item's home section.

`context`
This field contains the main body of the query. It can be one of a number of different object types that correspond to the content types in the current publication. In the case of the demo publication, the possible object types are **`SectionPage`**, **`Story`**, **`Picture`** and **`Video`**. The structure of these object types is then directly related to how they are defined in the publication's **`content-type`** and **`layout-group`** resources.

A standard CUE Front **`Query`** object always has these members.

The first line in the **`context`** segment of the query contains:

```
... on SectionPage
```

`... on` is a GraphQL conditional clause. It says "if this **`context`** object is of the type **`SectionPage`**, then ...". **`index-page.graphql`** queries will always contain this clause to ensure they only operate on section pages. If you look at the other supplied queries, you will see that they contain similar clauses to select the appropriate page types: **`... on Story`** in **`story.graphql`**, **`... on Picture`** in **`picture.graphql`** and **`... on Video`** in **`video.graphql`**. **`...on`** clauses are used other places in **`index-page.graphql`** to distinguish between object types and determine how to handle them.

Another useful GraphQL construct is:

`...`*name*

for example:

```
top {
  ...teaser
```

which appears on line 52 of **`index-page.graphql`**. This is simply an inclusion mechanism. It includes a **fragment** (called **`teaser`**), defined further down in the query:

```
fragment teaser on AtomLink
```

In this case, therefore the **`...teaser`** statement is equivalent to

```
  ... on AtomLink {
    [body of teaser fragment]
  }
```

but allows the fragment to be reused in multiple places in the query, if required.

If you want to know more about GraphQL, there is a helpful tutorial here.

3.2.3 Naming GraphQL Queries

You can control what GraphQL queries are used to retrieve content in different contexts by observing the following naming conventions:

- The query called **`index-page.graphql`** is the default query used for all section pages.
- If you want to use different queries for some sections, create queries with names of the form **`index-page-`***section-name***`.graphql`**, where *section-name* is the unique name of a section. A query named like this will be used for the specified section (but not for its subsections). A query

called **`index-page-sports.graphql`**, for example, will be used for the **Sports** section but not for any of its subsections (**Football**, for example).

- There is no default query for content items. You **must** create a separate query for each content-type in your publication, with a name of the form *content-type* **`.graphql`**.
- If you want to use different queries for certain content item types when they belong to particular sections, then you can do so by creating queries with names of the form *content-type-section-name* **`.graphql`**. A query called **`story-sports.graphql`**, for example, will be used for story content items that belong to the **Sports** section.

Currently, the Cook's GraphiQL editor provides no means of renaming queries or saving them under new names. If you want to make specialized queries for particular sections, then you must do it as follows:

1. Log in on the machine where CUE Front is installed.
2. **`cd`** to your CUE Front installation.
3. Copy an existing query to a new name. For example:

   ```
   cp recipe/queries/index-page.graphql recipe/queries/index-page-sports.graphql
   ```

4. Restart the Cook:

   ```
   docker-compose restart cook
   ```

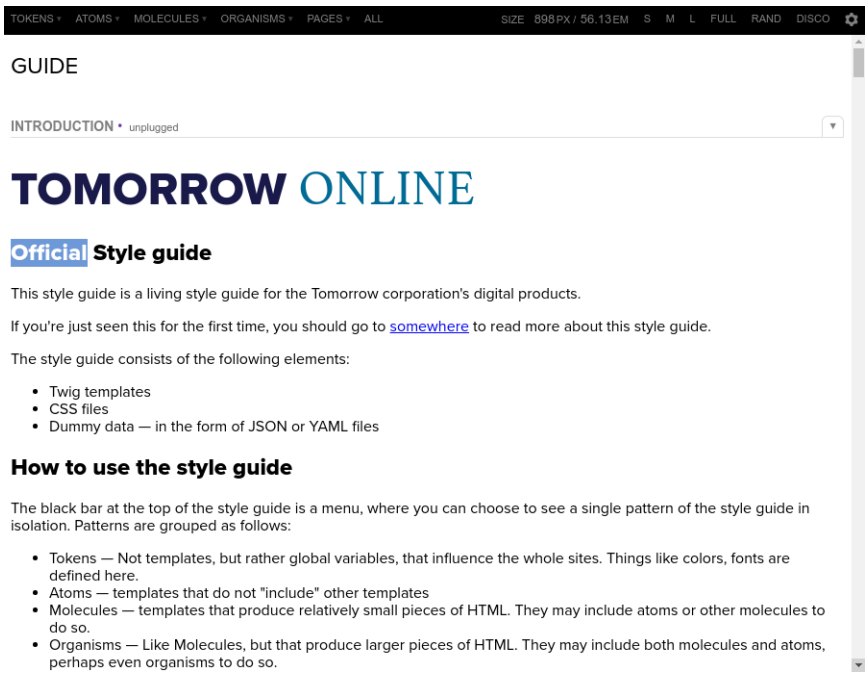
5. If you now open the GraphiQL editor in the context of the **`Sports`** section by pointing your browser to **`http://`***cue-front-host*`:8101/tomorrow-online/sports/edit`**, you will see that the editor loads the **`index-page-sports.graphql`** query rather than **`index-page.graphql`**.

You can now modify **`index-page-sports.graphql`** so that the Cook delivers a different JSON structure for the **Sports** section than it does for other section pages.

3.3 Working With Twig and Patternlab

The Waiter generates the pages of a publication by combining the JSON returned by the cook with Twig templates and CSS styles. The default location of those styles and templates is the CUE Front **`templates`** folder. The contents of this folder is monitored by a synchronization process, so any changes you make to templates or SCSS files in this folder are immediately copied into the Waiter's Docker container and result in corresponding changes to your web site. These templates, however, not only drive the actual web site - they also drive a Patternweb style guide, which you can use to view static examples of all the templates that make up your publication.

Open your browser and point it at `http://localhost:8103/`. You should see the demo publication's style guide, displayed using the Patternlab web application:



Using this application you can explore all the Atomic Design **patterns** from which the demo application is constructed – each pattern being a Twig template fragment.

You will see that Patternlab's menu bar contains menus called **TOKENS**, **ATOMS**, **MOLECULES**, **ORGANISMS** and **PAGES**. These menus represent different types of patterns. The **PAGES** menu contains the names of the page patterns used for the demo publication: **Atomic Frontpage** is the name of the pattern used for the publication's section pages, and **Article Page** is the name of the pattern used for story pages. The **ORGANISMS** menu contains re-usable patterns that may appear several places in a page pattern, or in several different page patterns, such as the **Five Story Section** component. The **MOLECULES** menu contains smaller patterns that may appear several places in different organisms or directly in page patterns, and the **ATOMS** menu contains even smaller patterns that may be re-used in molecules, organisms or pages. Finally, the **TOKENS** menu contains variables defining the colors, fonts, icons and so on that form the basis of the design.

When you select a pattern from one of the menus, the template is processed using Twig and the results are displayed in Patternlab. In order to be able to display the patterns, Patternlab has access to some sample JSON data for merging with the templates.

Besides allowing you to browse the patterns from which a design is constructed, Patternlab offers a number of other functions. The most useful are:

- You can display a pattern's template code plus a description of the pattern by selecting **Show Pattern Info** from the **Tools** menu at the right hand end of the toolbar.
- You can see what each pattern looks like on different size screens by selecting a size option from the right hand end of the menu bar: **S**mall, **M**edium, **L**arge or **Full** (the default).

Patternlab requires the templates that make up a pattern library to be stored in a known location, in accordance with specific naming conventions. The demo publication's patterns are stored in

cue-front/templates/_patterns. In **cue-front/templates/_patterns/10-pages**, for example, you will find all the templates that appear in Patternlab's **PAGES** menu.

Patternlab is a very useful review tool for designers: you can work directly on the patterns in the library, and use Patternlab to review the results of the changes in a variety of contexts. If you make a change to an atom template, for example, then you can use Patternlab to see what the change looks like in a variety of contexts:

- The atom in isolation
- The various molecules, organisms and pages in which the atom appears
- At different screen sizes

In addition, since Patternlab uses locally stored static data files for display purposes, you are not dependent on access to a working site for the design work. If you want to export the Patternlab style guide to work with it on a different machine, you can do so by entering:

```
make dist-style-guide
```

in the **cue-front** folder. This will create a zip file containing the style guide in the **cue-front/dist** folder.

Patternlab supports the concept of pattern **states** such as in **progress**, **in review**, **unplugged** and **complete** to help you organize your workflow. Pattern states are represented by coloured dots displayed before the pattern names in Patternlab menus, and the states are "inherited". That is, if an atom is in progress, then all other patterns that include that atom will also be displayed as in progress by Patternlab.

Pattern states are implemented by means of a naming convention. To put a pattern in the **unplugged** state, you simply append **@unplugged** to the end of its file name: rename **00-header.twig** to **00-header.twig@unplugged**, for example.

A good deal of Patternlab functionality is governed by naming conventions. For a brief introduction to these conventions, see section 3.3.1. For more detailed information about Patternlab, see the Patternlab documentation.

3.3.1 Patternlab Conventions

This section describes Patternlab conventions as they are used in CUE Front. For more detailed information about Patternlab conventions, see the Patternlab documentation.

Templates are stored in the **cue-front/templates/_patterns/** folder. Each subfolder within this folder defines a **top level pattern group** that appears as a menu in the Patternlab menu bar: The folders are:

01-tokens
02-atoms
03-molecules
04-organisms
10-pages

The numeric prefixes are used to control the order in which the menus appear in the menu bar. These top level pattern group names may **not** include hyphens.

You can either place Twig templates directly in the top level pattern group folder, or you can create subfolders that will be displayed as submenus in Patternlab and then place your Twig menus in the sub folders. You can use numeric prefixes to control the order of both subfolders and Twig menus, just as for the top level folders.

Twig files may be given state suffixes such as **`@inprogress`** and **`@unplugged`** to indicate their current state.

Patternlab also enforces conventions with regard to the naming of patterns within Twig templates. In order to include a template within another template, you construct the template name as follows:

```
topLevelPatternGroup-pattern
```

where:

topLevelPatternGroup
: is the name of the top level pattern group to which the pattern belongs (excluding any numeric prefix)

pattern
: is the name of the pattern (excluding any numeric prefix, any state suffix and the **`.twig`** file extension)

In other words, the Twig template **`cue-front/templates/_patterns/04-organisms/02-articles/richtextfield.twig`** must be referenced as follows when included in another template:

```
{% include "organisms-richtextfield" %}
```

The important things to note here are that:

- The name is composed only of the top level pattern group name and the pattern name: the subfolder name **`articles`** is not used
- Since subfolder names are not used, you must ensure that your pattern names are unique within each top level pattern group
- No relative addressing is used (so that templates can easily be moved around in the folder structure)

3.4 Managing Multiple Publications

The Waiter can be configured to serve multiple publications from the same Cook and Content Store. Depending on how similar the publications are they can either be rendered using shared templates and styles or a completely separate template tree. Whichever method you choose, separate Patternlab style guides are generated for each publication, so you can use Patternlab to review layouts for all your publications.

3.4.1 Shared Templates and Styles

> Currently, support for multiple publications based on shared templates and styles is limited to Docker-based installations. If you have a bare-metal installation then you will need to maintain separate template folders for each publication.

If the publications to be served by the Waiter are similar in structure (that is, they are based on similar Content Store **`content-type`** and **`layout-group`** definitions), then they can in many cases be rendered using a single set of templates and styles. Any layout differences that are required can be provided using overlay templates and style definitions.

Suppose, for example, that you have two publications called **`mypub`** and **`myotherpub`**. They have identical **`content-type`** and **`layout-group`** definitions, and can therefore be served using the same Twig templates and styles. You would, however like **`myotherpub`** to differ from **`mypub`** in certain respects. You can do this by creating alternative versions of some files in your CUE Front **`templates`** folder:

- Alternative **`.scss`** files (called **style overlays**) where you want to make changes to the publication CSS
- Alternative **`.twig`** files (called **template overlays**) where you want to make changes to publication HTML
- Alternative **`.json`** files (called **data overlays**) where you want to make changes to the static data displayed in the publication styleguide
- Alternative **`.md`** files (called **description overlays**) where you want to make changes to the descriptions displayed in the publication styleguide
- An alternative **`favicon.ico`** file where you want to make changes to the publication's favicon

Any overlays you create must be named according to the following convention:

```
base-name+overlay-name.extension
```

where:

base-name
: Is the name of the original Twig or SCSS file you are making a modified version of. The base name must be **exactly** the same as the name of the original file it is based on, including all prefixes and suffixes.

overlay-name
: Is the name of the overlay the modification is to belong to. You are recommended to use the name of the overlay's target publication as the overlay name, since the relationship between overlays and publications is 1:1.

extension
: Is the file type extension (**`twig`** or **`scss`**)

To make a **`myotherpub`** overlay for the template **`02-medium-teaser.twig`**, for example, you would copy it to a file called **`02-medium-teaser+myotherpub.twig`** and then make whatever changes you want in the copied file. Similarly, to make a **`myotherpub`** overlay for **`_colors.scss`**, you would copy it to a file called **`_colors+myotherpub.scss`** and then make your required changes.

> You **must** use a **`+`** sign to separate *overlay-name* from *base-name*, no other character may be used. If you make use of Patternlab variants, then the overlay name must still come last, after the variant name. In other words, the overlay for a file called **`02-medium-teaser~special.json`** must be called **`02-medium-teaser~special+myotherpub.json`**, not **`02-medium-teaser+myotherpub~special.json`**.

The CUE Front synchronization process copies the contents of the **`templates`** folder to one or more folders created in a Docker volume that is mounted in the Waiter and Patternlab containers. In an

installation with no overlays this volume has only one templates folder, called **templates/_base**, which contains an unmodified copy of the source **templates** folder. If there are overlays, then the volume's **templates** folder contains one subfolder for each overlay name, plus a **_base** subfolder. If you include **+mypub** and **+myotherpub** overlay names in your source templates folder, for example, then the Waiter and Patternlab containers will have access to the following generated templates folders:

templates/_base
: Contains a copy of the source **templates** folder where all **+mypub** and **+myotherpub** files are removed.

templates/mypub
: Contains a copy of the source **templates** folder where:

- all files with **+mypub** overlays are replaced by their overlays (for example, **02-medium-teaser.twig** is deleted, and **02-medium-teaser+mypub.twig** is copied to **02-medium-teaser.twig**).
- all **+myotherpub** files are removed.

templates/myotherpub
: Contains a copy of the source **templates** folder where:

- all files with **+myotherpub** overlays are replaced by their overlays. (for example, **02-medium-teaser.twig** is deleted, and **02-medium-teaser+myotherpub.twig** is copied to **02-medium-teaser.twig**).
- all **+mypub** files are removed.

The generated template folders in this Docker volume are kept in sync with the master **templates** folder you work on. Any changes you make to files in the master **templates** folder are immediately copied to the appropriate generated templates folders, and will therefore be reflected in both your publications and the publication style guides.

As part of the synchronization process, the SCSS files in the generated template folders are compiled to a single CSS output file for each publication. The SCSS files in the base templates folder are compiled to a file called **layout.css**. The SCSS files in the overlay templates folders are compiled to files called *overlay*.**css**. The SCSS files in a **templates/mypub** folder would be compiled to a **mypub.css** file, and the SCSS files in a **templates/myotherpub** folder would be compiled to a **myotherpub.css** file. All the generated CSS files are written to a separate Docker volume which is also mounted in the Waiter and Patternlab containers. The Waiter and Patternlab will therefore in this case have access to three different CSS files called **layout.css**, **mypub.css** and **myotherpub.css**.

> A new overlay template folder is automatically created as soon as you rename any file using an overlay name that you haven't used before. In other words, renaming **_colors.scss** to **_colors+newpub.scss** will cause a **templates/newpub** folder to be created if it doesn't already exist. This is not the case for CSS files, however: a **newpub.css** file will not be automatically created. In order to trigger the creation of a new CSS file, you must restart the **styles** Docker container:
>
> ```
> docker-compose restart styles
> ```
>
> This is only required to create a new CSS output file. If you subsequently make further CSS changes, **newpub.css** will be regenerated automatically.

3.4.1.1 Creating a Publication Overlay

The overall process of creating a publication overlay is as follows:

1. Find the SCSS files in the **`templates`** folder (in the **`tomorrow-online`** demo publication, they are located in **`templates/theme/css`**).
2. Make copies of any SCSS files you want to modify and rename them by adding a ***+**overlay-name* suffix as described above.
3. Modify the renamed SCSS files as required.
4. Find any Twig templates that contain references to the CSS file **`layout.css`**. In most cases there should only be one such template (in the case of the **`tomorrow-online`** demo publication it is **`templates/_patterns/01-globals/05-nonvisual/_stylesheet.twig`**).
5. Make copies of these templates and rename them by adding a ***+**overlay-name* suffix as described above.
6. Open the renamed templates for editing and replace **`layout.css`** with *overlay-name* **`.css`**.
7. Find any other files that you want to modify (**`.twig`**, **`.json`**, **`.md`**, **`.ico`**). Copy them, rename them as described above and modify them to meet your requirements.
8. Reconfigure the Waiter to recognise and handle the publication you have created the overlays for as described in section 9.5.
9. Reconfigure **`nginx`** to recognise and handle the publication you have created the overlays for (also described in section 9.5).
10. Finally, restart the Waiter and Styles containers:

```
docker-compose restart waiter styles
```

3.4.2 Separate Templates and Styles

If the publications to be served by the Waiter are very different in structure (that is, they are based on different Content Store **`content-type`** and **`layout-group`** definitions),or if the target layouts are totally different, then it is probably not practical to serve them using a shared set of templates and style definitions. If you have a bare metal CUE Front installation then shared templates and style definitions are not supported at present.

In both these cases you will need to copy the CUE Front **`templates`** folder as many times as required and maintain separate templates folders for each publication (**`templates`** and **`templates_myotherpub`**, for example).

You specify which templates folder a publication is to use in the **`waiter-config.yaml`** configuration file. (see section 9.5 if you are using the setup tool to create Docker based configurations, or section 8.3.1 if you have a bare metal installation).

3.5 Extending CUE Front

Most CUE Front-based sites have some requirements that cannot be satisfied by CUE Front and the CUE system alone – either because CUE does not provide a particular service or because the customer needs or wishes to make use of a specific third-party service. Typical examples include access to external data sources such as sports results services or stock market information, user login, paywall systems, cookie management and so on.

It is not always obvious what approach users should take when implementing such extensions: what technology to use, where to extend CUE Front and so on. This section is intended to provide a few guidelines to follow when making decisions of this kind.

First of all, here is a list of some of the possible ways in which you can add functionality to a CUE Front installation:

- Extend the Cook by creating a recipe extension that retrieves content from an external service such as a stock ticker service.
- Extend the Cook by creating a recipe extension that restructures or extends content retrieved from the Content Store in some way.
- Create your own service that generates interesting information of some kind, and then create a recipe extension that retrieves content from it.
- Create an extension to the Waiter that generates or retrieves content of some kind, or provides some kind of service to site users.
- Create a front-end component that sits alongside the Waiter and provides some kind of content or service directly to clients. In this case, the **`nginx`** web server that fields incoming requests must act as a router, forwarding requests to the Waiter or to your service depending on the request URL.

The question of how and where to implement an extension should be determined by the following considerations:

What kind of extension is it?

One of the objectives underlying CUE Front's design is to keep business logic separate from presentation and layout. The general idea is that the Cook should handle all business logic, and provide the Waiter with all the information needed to construct web pages. So the general recommendation is to implement extensions in the back end as recipe extensions. This is particularly the case for extensions that are primarily concerned with content (such as sports results services or stock market information services). Implementing content-rich extensions as recipe extensions means that the retrieved or generated content can be merged into the JSON data structure returned by the Cook, allowing all layout to be handled by the Waiter (or your alternative front-end solution) in a uniform way.

However, extensions that are primarily concerned with user access are probably best handled in the front end (whether that is CUE's Waiter, or your own alternative front-end). For example, user login systems, paywall systems, cookie management systems and so on. Cookies in particular are best dealt with in the front end. By default, the Waiter does not pass cookies on to the Cook in order to ensure the "cacheability" of the Cook's responses.

What kind of front end do you have?

If you are using CUE's Waiter as your front end and you want to extend the front end, then you might want to create a separate component that sits alongside the Waiter rather than to directly extend the Waiter itself. This will:

- Give you complete control over how you implement the service: you can make use of the languages, tools and libraries of your choice without having to conform to any limitations imposed by the Waiter.
- Give you the freedom to upgrade the Waiter without needing to merge updates from Escenic with your own changes. The Waiter is a very small, simple application that is unlikely to change much in the future, and CCI Europe does not officially maintain it. It is provided "as is" as a starting point for development projects. Nevertheless, if you want to ensure that you

will be able to easily replace it with Waiter versions included in later CUE Front start packs, you can do so by implementing any front-end extensions as separate components.

What are your preferences, strengths and weaknesses?
The answers to the two previous questions is general advice that may not necessarily apply in all situations or for all organizations. For example, implementing an extension as a recipe extension because it is "content-rich" may make sense in a general way, but not if your development department is a PHP shop with very little experience of node.js development. In such circumstances, sticking to what you know may well be the best strategy.

Routing

When implementing front end extensions, you need to ensure that routing is correctly handled and that there is no possibility of your components duplicating URLs. If you implement your extensions as standalone applications running alongside the Waiter, then this should not be a problem: you just configure nginx to route requests to the correct component based on the first part of the URL, and then each component is thereafter responsible for its own URL space.

If, on the other hand, you directly extend the Waiter or alternative front end, then the recommended approach is to let the Cook do the first part of the routing for you by passing all incoming requests to the Cook in the usual way. Any URLs that are not recognized by the Cook are returned in the JSON output's `remainingPath` field, from which point your extension can continue the routing process as required, and deal with the requests.

3.6 CUE Front Development Environment

To be supplied.

4 Using the Fridge

The Fridge is a small proxy web server that can be used to serve static content from a file system folder. You can use the Fridge in two different ways:

Fridge as Cook Proxy
You can configure the Waiter to retrieve content from the Fridge instead of retrieving it from the Cook. In this case, the Fridge needs to contain JSON documents of the kind returned by the Cook.

Fridge as Content Store Proxy
You can configure the Cook to retrieve content from the Fridge instead of retrieving it from the Content Engine. In this case the Fridge needs to contain Atom documents of the kind returned by Content Store web services, binary files and so on.

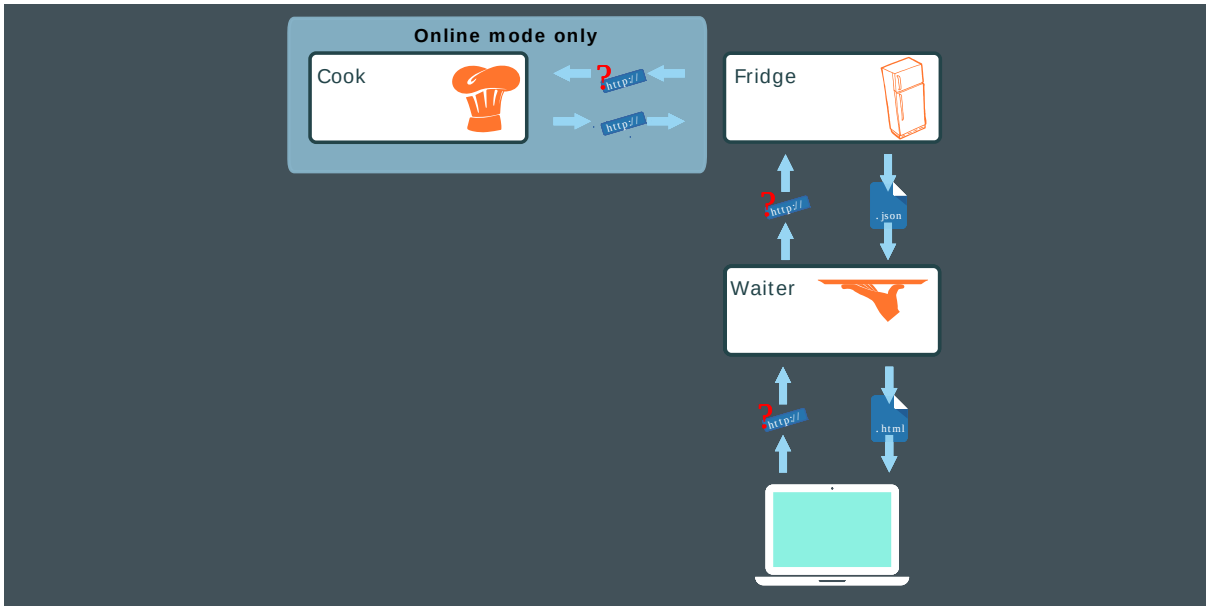
Internally, the Fridge can be configured to o

- **Offline mode**, in which case it only ever
resource cannot be found there, then the
- **Online mode**, in which case the Fridge
found in the cache folder, then the Fridge forwards the request to the original server (the Content Engine or Cook). When the original server responds, it does two things: it forwards the response to the caller **and** it stores the response in its cache folder.

The Fridge can be used for two quite different purposes:

- Offline template development
- Caching

4.1 Fridge as Cook Proxy



The Waiter can be configured to use the Fridge as its data source instead of the Cook. If you first run the Fridge in online mode and use the web site for a while, then the Fridge's cache will slowly fill up with JSON data representing all the visited pages. Once enough data has been assembled in this way, you can switch the Fridge into offline mode and continue to use the web site. It will work as before so long as you do not attempt to visit any new pages — if you do visit an uncached page, then the Fridge will return an HTTP "Page not found" error.

This means you can, for example, use the Fridge to enable template development in offline locations where you do not have access to the Cook. You can also copy the content of the Fridge's cache to Fridge instances on other machines, enabling other developers who have no access to the Cook themselves to work on template development using a realistic data set from the actual site. Given a set of JSON files to work with, all a designer needs is a Fridge to serve the JSON files and a Waiter to render the JSON files as HTML. The designer can then work on the Waiter's templates without any need for a Cook or Cleaver, or access to a Content Store.

4.1.1 Configuring the Fridge as a Cook Proxy

To set up the Fridge as a Cook proxy in a Docker-based development installation, open the Waiter's configuration file (**`docker/waiter-config.yaml`**) in an editor and uncomment the following line:

```
proxy: "fridge:8104"
```

Then restart the Waiter's Docker container:

```
docker-compose restart waiter
```

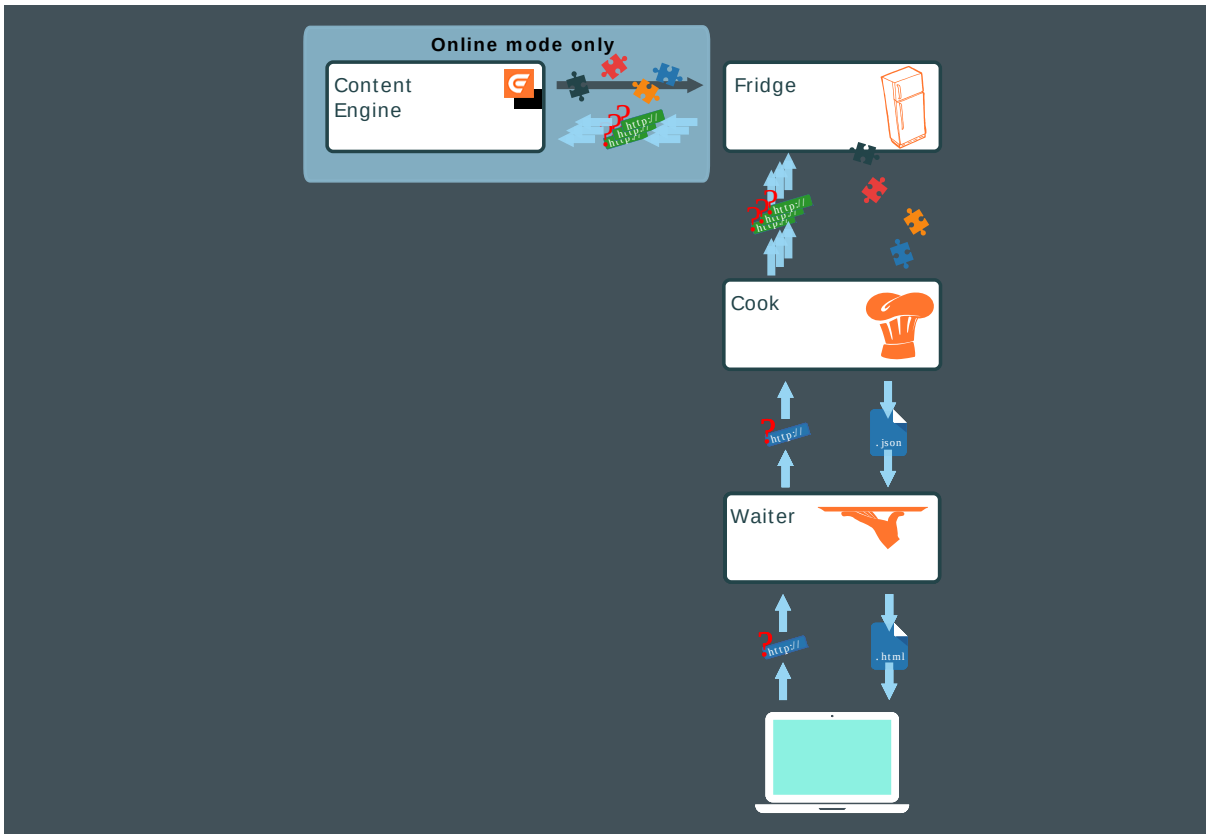
The Waiter will now direct all its requests to the Fridge instead of to the Cook. The Fridge will look for all requested resources in its cache and return them to the Waiter if found. If it cannot find the resources there, then it will forward the request to the Cook. When the Cook responds it will then both add the returned resources to the cache and pass them back to the Waiter. In this way, the Fridge will eventually be filled with more and more of the web site content.

Once the Fridge contains sufficient content for your purposes, you can switch it into offline mode as follows:

1. Open the Fridge's configuration file (**`docker/fridge-config.yaml`**) in an editor.
2. Set the **`proxy`** property to **`false`**:
   ```
   proxy: false
   ```
3. Save your changes.
4. Restart the Fridge's Docker container:
   ```
   docker-compose restart fridge
   ```

You will see that you can now revisit any page you have already visited, but if you try to visit a new page, the browser will respond with an HTTP 400 error (Page not found).

4.2 Fridge as Content Store Proxy



The Cook can be configured to use the Fridge as its data source instead of the Content Store. If you first run the Fridge in online mode and use the web site for a while, then the Fridge's cache will slowly fill up with Atom documents and binary files representing all the visited pages. Once enough data has been assembled in this way, you can switch the Fridge into offline mode and continue to use the web site. It will work as before so long as you do not attempt to visit any new pages — if you do visit an uncached page, then the Fridge will return an HTTP "Page not found" error.

This means you can, for example, use the Fridge to enable both back-end recipe development and template development in offline locations where you do not have access to the Content Store. You can also copy the content of the Fridge's cache to Fridge instances on other machines, enabling other back-end developers to work with the Cook in locations where they do not have access to the Content Store.

4.2.1 Configuring the Fridge as a Content Store Proxy

To configure the Fridge as a cache in a Docker-based development installation, open the Cook's configuration file (**`docker/cook-config.yaml`**) in an editor and uncomment the following line:

```
    proxy: "fridge:8104"
```

Note that this line is indented two levels — make sure you maintain the correct indentation after removing the comment character.

Then restart the Cook's Docker container:

```
docker-compose restart cook
```

The Cook will now direct all its requests to the Fridge instead of to the Content Store. The Fridge will look for all requested resources in its cache and return them to the Cook if found. If it cannot find the resources there, then it will forward the request to the Content Store. When the Content Store responds it will then both add the returned resources to the cache and pass them back to the Cook. In this way, the Fridge will eventually be filled with more and more of the web site content.

If you are intending to use the Fridge for offline development, then once it contains sufficient content for your purposes, you can switch it into offline mode as follows:

1. Open the Fridge's configuration file (**`docker/fridge-config.yaml`**) in an editor.
2. Set the **`proxy`** property to **`false`**:

   ```
   proxy: false
   ```

3. Save your changes.
4. Restart the Fridge's Docker container:

   ```
   docker-compose restart fridge
   ```

You will see that you can now revisit any page you have already visited, but if you try to visit a new page, the browser will respond with an HTTP 500 error (Page not found).

4.2.2 Using the Fridge as a Cache

The Fridge can play an important role in production environments as a cache. The Cook is configured to use the Fridge and the Fridge is configured to run in online mode. As the Fridge's cache fills up with data, the Fridge is able to respond to more and more requests by simply returning files from its cache, thereby minimizing the load on the Content Engine. In the most extreme case, all of a web site's content can be duplicated in the Fridge's cache so that no requests ever reach the Content Engine.

For such a solution to work, the content of the Fridge's cache must be kept up to date. The traditional mechanism for doing this is **expiration**: each piece of content in the cache is marked as expired after some arbitrary length of time. When content is retrieved from the cache, it is checked to see if it has expired: if it has expired, then it is discarded and a new copy is retrieved from the back end. This mechanism is obviously not very efficient for content that changes infrequently, since it means that content will often be refreshed even though it has not changed.

For this reason, the Fridge does not use an expiration mechanism. Instead, an Escenic component called the Change Log Daemon is used to monitor and keep a record of all changes made to the content in the Content Store. Every time a change is made to any content, that change is pushed to the Fridge, ensuring that the Fridge's contents are always fresh.

A script supplied with the Fridge can be used to start up a Change Log Daemon instance that watches the Content Store for changes and keeps the Fridge contents fresh.

Using the Fridge in this way offers several advantages in production environments:

- It improves the scalability of the system by completely decoupling the presentation layer from the Content Store and the editorial system. Increases in audience can be met by simply duplicating CUE Front components, without any need to scale the Content Store or its database.
- It improves the reliability of the system: the Content Store can be taken off line without affecting the presentation layer in any way.

- It can enable improved performance by allowing the Fridge's cache to be stored in a content delivery network, for example.

For information on how keep the Fridge's content fresh using a Change Log Daemon, see section 8.4.2.

5 Using Data Sources

A standard Cook GraphQL query (called a **content retrieval query**) allows you to request information about specific resources (sections or content items) stored in the Content Store. It allows you to determine what items of information about a given resource you want to retrieve. You can, for example, specify which fields of a content item you want to retrieve. You can also specify which of the content item's relations you want to follow, and how much information you want to retrieve about each of it's related items. Similarly, for a section page, you can specify which layout groups you are interested in, and how much information you want to retrieve about the content items desked in those groups.

A content retrieval query, however, only lets you request information directly related to the **context object** — that is, the section page or content item pointed to by the request URL. Sometimes you want to be able to include other information on a page. You might, for example, want to include links to content items with a particular tag, content items belonging to a different section or even a different publication, content items that are tagged with the same tag as the current content item and so on.

Data sources provide a means of including this kind of information in the JSON data returned by the Cook. A data source is a kind of saved search, written in graphQL. You can use the Cook's graphQL editor to write **data source queries** that are not limited to traversing the Content Store's graph. Data source queries are in fact mostly executed by Solr and offer a great deal of power and flexibility.

You can make the following kinds of data source queries:

- Get content items related to the current content item
- Get content items of a specified type
- Get content items belonging to a specified publication
- Get content items belonging to a specified sections
- Get content items with a specified value in a specified field
- Get content items tagged with a specified tag
- Get content items that share one or more tags with the current content item
- Get content items related to the current content item

You can also make more complex queries by combining queries of most of the above types using **`AND`** and **`OR`** operators. So, for example, the following data source query will get all **`story`** content items that have an "Elections" tag:

```
query {
  and {
    tag(tag: "tag:tomorrowonline@escenic.com,2017:elections")
    type(name: "story")
  }
}
```

This slightly more complex query will get all **`story`** or **`picture`** content items that have an "Elections" tag:

```
query {
  and {
    tag (tag: "tag:tomorrowonline@escenic.com,2017:elections")
```

```
      or {
       type (name: "story")
       type (name: "picture")
      }
    }
  }
```

Executing a data source query returns an intermediate JSON structure containing the results of the query.

You can save data source queries and then execute them from within your content retrieval queries. GraphQL can then be used to pick out the exact items of information required from the results returned by a data source. This effectively makes it possible to construct extremely sophisticated queries that leverage the strengths of both GraphQL and Solr.

5.1 Configuration

In order for data source queries to work, the Cook must be correctly configured to access Solr. If you used the setup tool to create a Docker-based configuration for you, then everything should be OK already and you can skip this section.

If your cook is running on bare metal or if data sources are not working then open your **`cook-config.yaml`** for editing and make sure the following section is filled out correctly:

```
recipedata
  extensions
    - name: '@escenic/cue-front-extension-datasources'
      config:
        queries: /srv/recipe/datasources/
        endpoint: 'http://my-escenic.com:8080/webservice/'
        solr: 'http://my-escenic.com:8983/editorial'
```

The entries you need to fill out are:

`endpoint`
: The URL of your Content Store web service, including a final **`/`** character.

`solr`
: The URL of your Solr core. Solr is often installed on the same host as the Content Store, but it does not have to be. By default Solr listens on port 8983, but it can have been set up to use a different port. The final part of the URL is the name of the Solr core that is to be used. In a production environment, the data source extension should always be configured to use Solr's **`presentation`** core. In a development environment, however, it is often the case that no **`presentation`** core is available since a default Content Store installation does not include one. So for development purposes, use the **`editorial`** core if no **`presentation`** core is available.

5.2 Creating a Data Source

To create a data source, start up your browser and navigate to the "Cook view" of any page in your publication. For example: **`http://`***cue-front-host*`:8101/tomorrow-online/`**. You should see the JSON data for the page you have chosen. Now add the **`/edit`** suffix to the URL to display the

GraphiQL editor. Make sure that the editor is displayed and that it has a **Save** button. If it doesn't, then you need to enable saving (see section 3.2.1.1).

If saving is enabled, then replace the **/edit** suffix with **/_datasource/politicalContent**. This URL means "show me the data source query called **politicalContent**, in the context of **http://***cue-front-host***:8101/tomorrow-online/**. Assuming you haven't already created a data source query called **politicalContent**, what you will see is the following message:

```
{
  message: "No query found with name 'politicalContent'"
}
```

Adding **/edit** to the end of this URL (so the whole URL is **http://***cue-front-host***:8101/tomorrow-online/_datasource/politicalContent/edit**) should give you a new, empty GraphiQL editor with the title **politicalContent.graphql**. To start your query, enter:

```
query {

}
```

and with the cursor inside the braces, press **Ctrl-Space**. You will see that the editor works in the same way as when editing a content retrieval query, but that the options available to you are different. If you explore the help in the **Docs** section on the right side of the editor, you will see that it too now contains completely different information, aimed at helping you build a data source query rather than a content retrieval query.

A data source query is made by combining special functions called **filters**. The root **query** field may contain only one top-level filter: an **and**, an **or** or a **related()**. All the other filter types can only be used as children of an **and** or **or** filter. For example:

```
query {
  and {
    tag(tag: "tag:tomorrowonline@escenic.com,2017:politics")
    type(name: "story")
  }
}
```

This will generate a Solr query in which the child filters are combined with **AND** operators:

```
(classification:"tag:tomorrowonline@escenic.com,2017:politics" AND
 contenttype:"story")
```

If the top level filter was an **or** instead, then the filters would be combined with **OR** operators:

```
(classification:"tag:tomorrowonline@escenic.com,2017:politics" OR contenttype:"story")
```

Even if your query only has one such filter, the Cook requires you to have an **and** or **or** at the top level (although in this case, of course, it doesn't matter which you choose). This data source query:

```
query {
  and {
    type(name: "story")
  }
}
```

will generate this Solr query:

```
(contenttype:"story")
```

The child filters in an **`and`** or **`or`** filter may themselves be **`and`** or **`or`** filters. **`not`** filters are also allowed. This allows you to construct more sophisticated queries. This data source query, for example:

```
query {
  and {
    tag (tag: "tag:tomorrowonline@escenic.com,2017:politics")
    not {
      tag (tag: "tag:tomorrowonline@escenic.com,2017:elections")
    }
    or {
     type (name: "story")
     type (name: "picture")
    }
  }
}
```

will generate this Solr query:

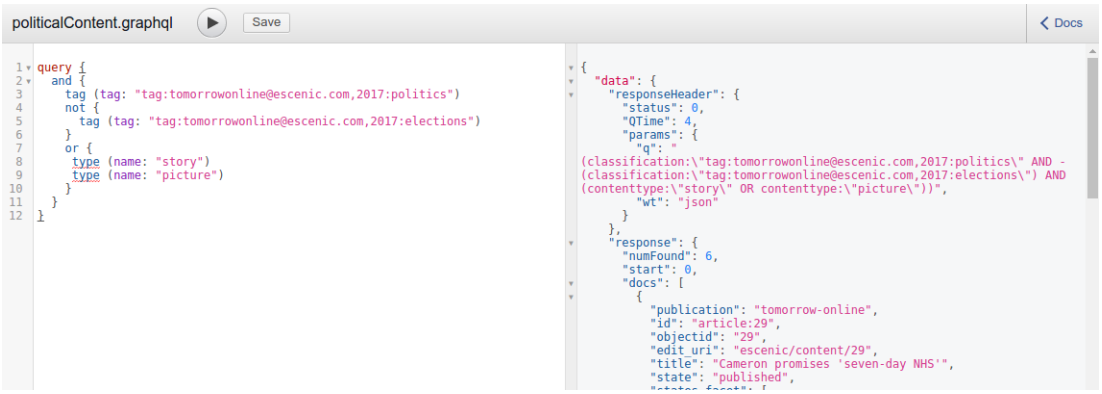
```
(classification:"tag:tomorrowonline@escenic.com,2017:politics"
AND -(classification:"tag:tomorrowonline@escenic.com,2017:elections")
AND (contenttype:"story" OR contenttype:"picture"))

(contenttype:"story")
```

When you execute this kind of data source query by clicking the editor's play button ( ), the Cook:

- Generates a Solr query
- Submits the query to Solr
- Displays a response in the editor containing both the query submitted to Solr and the response. The Solr response is a JSON structure containing information about the matching content items found.

For example:



You can use this output to verify that your query is working correctly and returning the content you are interested in. Once you are satisfied, you can save the query by clicking the **Save** button. The query is saved in your CUE Front's **`recipe/datasources`** folder. You can modify it at any time either by opening **`recipe/datasources/politicalContent.graphql`** in an editor of your choice or by returning to **`http://`***cue-front-host*`:8101/tomorrow-online/_datasource/politicalContent/edit`** in the browser.

The top level filter **`related()`** is different from all the other data source filters in that it is not implemented using Solr, but works by directly accessing the Content Store web service. For further information, see section 5.4.11.

5.2.1 Data Source Context

When you edit the **`politicalContent.graphql`** data source query at the URI **`http://`***cue-front-host*`:8101/tomorrow-online/_datasource/politicalContent/edit`** in a browser, then you are editing it in the context of the publication's front page, **`http://`***cue-front-host*`:8101/tomorrow-online/`**. You can view and edit the same data source query in the context of any page in the publication: for example by going to **`http://`***cue-front-host*`:8101/tomorrow-online/ politics/2017-07-05/The-challenges-of-election-polling-34.html/_datasource/politicalContent/edit`**. You will still be editing exactly the same data source, stored in **`recipe/datasources/politicalContent.graphql`** on your disk. And in the case of **`politicalContent.graphql`**, executing the data source will return the same results irrespective of where you execute it.

This is not the case for all data source queries. Some data source filters are context-dependent: they make use of the current context in the query submitted to Solr, so any data source that contains a context-dependent filter will return different results according to the context in which it is executed. Currently, the only context-dependent data source filters are the **`related()`** and **`sharedTags`** filters. The **`sharedTags`** filter, for example, returns a list of all content items that are tagged with the same tags as the context content item. So if you create a **`tagRelatedStories.graphql`** data source that looks like this:

```
query {
  and {
    sharedTags
  }
}
```

and execute it at **`http://`***cue-front-host*`:8101/tomorrow-online/politics/2017-07-05/The-challenges-of-election-polling-34.html/_datasource/tagRelatedStories/edit`**, you will see that it returns some results because the context is a content item that has some tags. If, however, you execute it at **`http://`***cue-front-host*`:8101/tomorrow-online/_datasource/tagRelatedStories/edit`**, then it will return no results because the context is a section, and sections have no tags.

5.2.2 Using Filter Aliases

You can improve the legibility of your data source queries by making use of aliases. You can use the **`name`** parameter of any filter as an alias for the filter. If you do this then you can drop the **`name`** parameter from the filter's parameter list, resulting in a cleaner, more legible query.

Here is a small query that does not make use of aliases:

```
{
  and{
    or{
      section(name: "politics" includeSubsections: true)
      section(name: "sport" includeSubsections: true)
    }
    type(name: "story")
  }
```

```
}
```

and here is the same query where the name parameters have been converted to aliases:

```
{
  and{
    or{
      politics: section(includeSubsections: true)
      sport: section(includeSubsections: true)
    }
    story: type
  }
}
```

The use of aliases in this way has no purpose other than improving legibility.

> GraphQL aliases may not contain hyphens, so you can't use them if the filter's **`name`** parameter contains a hyphen.

5.3 Using a Data Source

Once you have created some data sources, you can use them to enrich the data structures returned by the Cook's content retrieval queries. To do this you use a content retrieval function called **`datasource()`**. If you go back to editing the section page content retrieval query (**`index-page.graphql`**) at **`http://`***cue-front-host*`:8101/tomorrow-online/edit`**, place your cursor immediately above the **`headerMenu`** entry and press **`Ctrl-Space`**, then you will see that the displayed list includes a **`datasource`** option. Select it and add a **`name`** parameter, specifying the name of the data source you created:

```
  datasource(name: "politicalContent")
```

(You don't actually need to place the data source call before the **`headerMenu`** entry, it just has to be a top-level entry in the query, at the same level as **`resolution`**, **`context`**, **`menu`** and so on.)

The **`datasource`** function returns **`AtomEntry`** objects that contain information about each content item found by the data source query. One of the **`AtomEntry`** object's fields is **`__typename`**, which means that you can test the returned content items' type using the same **`... on`** *content-type* technique used to test the context object:

```
  datasource(name: "politicalContent") {
    ... on Story {
      displayId
      fields {
        title
      }
    }
  }
```

Once you have determined the types of the returned content items in this way, you have access to all of their content and relations in exactly the same way as for content items retrieved directly from the Content Store.

You can optionally prefix the **`datasource`** function with a descriptive field name to make the output structure easier to navigate:

```
  politicalContent: datasource(name: "politicalContent") {
    ... on Story {
      displayId
      fields {
        title
      }
    }
  }
```

Executing the query now will produce the same output as before, but with an additional **`politicalContent`** field containing the selected information about the content items returned by the datasource:

```
...
    "politicalContent": [
      {
        "displayId": "29",
        "fields": {
          "title": "Cameron promises 'seven-day NHS'"
        }
      },
      {
        "displayId": "26",
        "fields": {
          "title": "'£260m cost' if line not electrified"
        }
      },
      ..etc...
    ]
...
```

5.3.1 Datasource Function Parameters

The **`datasource()`** function has a number of parameters that can be used to control, limit and organize the returned content items:

```
datasource(
  name: String!
  deduplicate: Boolean
  sort: DataSourceOrder
  offset: Int
  count: Int
  params: [Param]
)
```

Only the **`name`** parameter is required.

`name`: ***String!***
The name of the data source to execute.

`deduplicate`: ***Boolean***
If set to **`true`**, then any duplicates in the result set are removed. The default is **`false`**. This parameter only has any effect when the called data source contains a **`related()`** filter, since this is the only data source filter that can return duplicates.

`sort`: **DataSourceOrder**
The order in which the returned elements are to be sorted. The possible options are:

`CREATED`: Sort by creation date, most recent first

OLDEST_CREATED: Sort by creation date, oldest first.
PUBLISHED: Sort by publishing date, most recent first.
OLDEST_PUBLISHED: Sort by publishing date, oldest first.
UPDATED: Sort by last update, most recent first.
OLDEST_UPDATED: Sort by last update, oldest first.

This parameter does not work for data sources that contain a **related()** filter.

offset: ***Int***
An offset to be applied to the result set (by omitting the first *n* results). If an offset of 2 is specified, then the first two results are omitted. The offset is applied after any deduplication and sorting.

This parameter does not work for data sources that contain a **related()** filter.

count: ***Int***
The maximum number of results to be returned. Once this number is reached, any remaining results are dropped. The **count** limit is applied after any deduplication, sorting and offset have been applied.

This parameter does not work for data sources that contain a **related()** filter.

params: **[*Param*]**
An array of key/value pairs to be passed in to the data source as parameters. Each element in the array consists of:

key: ***String!***
The name of the parameter to be passed to the data source.

value: ***String!***
The parameter value.

To pass two parameters called **pub** and **sec** to a data source called **mydatasource**, for example:

```
datasource(
  name: "mydatasource"
  params:
    [
      {key: "pub", value: "tomorrow-online"},
      {key: "sec", value: "sports"}
    ])
```

5.4 Data Source Reference

This section contains reference descriptions of the various components of a data source.

The primary components of a graphql query are normally called **fields**, since they reference the fields of a JSON data structure. A data source definition is not an ordinary graphql query in this respect: the "fields" do not represent the fields of a JSON structure, but data filtering functions. For this reason, the "fields" are referred to as **filters** throughout this section.

5.4.1 Query

```
query(
  [$param-name:param-type]*
)
```

```
{
  [ and | or | related() ]
}
```

or (omitting the **query** keyword):

```
{
  [ and | or | related() ]
}
```

The root of a data source query. The results of all the **query**'s child filters are concatenated in a single result data structure. No attempt is made to sort results or remove duplicate entries. This can, however, be done by the content retrieval **datasource()** function that calls a data source query (see chapter 5).

The **query** may only contain one child filter. If you add more than one filter (for example, an **and** filter and a **related()** filter), then when you execute it in the data source editor an error message is returned instead of a result set:

```
{
  "message": "Multiple queries are not allowed, please update your datasource to
 include a single query."
}
```

You can define any number of parameters for the query, specified using the format:

```
$param-name:param-type
```

where:

param-name
is the name of the parameter (preceded by a **$** sign)

param-type
is the type of the parameter (for example **String** or **Int**)

The parameters can be referenced in the body of the query using the same names, also preceded by a **$** sign. The parameters are passed in to the query from the **datasource()** function's **params** parameter (see section 5.3.1).

You must use the **query** keyword if you want to define parameters for a query.

5.4.2 And

```
and {
  [ and | or | not | publication | section | type | tag | sharedTags | field ]*
}
```

Combines all child filters with an **and** operator.

5.4.3 Or

```
or {
  [ and | or | not | publication | section | type | tag | sharedTags | field ]*
}
```

Combines all child filters with an **or** operator.

5.4.4 Not

```
or {
  [ and | or | not | publication | section | type | tag | sharedTags | field ]*
}
```

Combines all child filters with an **or** operator and returns all content items that do **not** satisfy the resulting conditions.

5.4.5 Publication

```
publication(
  name: String
)
```

Returns all content items belonging to a publication.

Parameters

If no parameters are specified, then **publication()** returns all content items in the current publication.

name: ***String***
The name of a publication. If specified, **publication()** returns all content items in the specified publication.

5.4.6 Section

```
section(
  name: String
  includeSubsections: Boolean
  publication: String
)
```

Returns all content items belonging to a specified section.

> The **section** filter requires both Solr and the CUE Content Store to be correctly configured, or else it will not work. The specific requirements are:
>
> - The Solr schema (**/etc/escenic/solr/***solr-core***/schema.xml** on your Content Store host) must contain the following field definitions:
>
>   ```
>   <field name="objectid_int" type="integer" indexed="true" stored="false"/>
>   <copyField source="objectid" dest="objectid_int"/>
>   ```
>
>   These definitions are **not** included in the default **editorial** and **presentation** schemas delivered with the Content Store. After making the change you will need to regenerate the Solr index. For general information about modifying the Content Store's Solr schemas and re-indexing, see Modifying The Standard Configuration.
> - The **types** property in the Content Store's index configuration must include the value **section**. If the Cook is configured to use the **editorial** Solr core, then this should be the case by default, but if it is using the **presentation** Solr core, then you will need to explicitly add it. To do so, open **/etc/escenic/engine/common/com/escenic/search/index/**

`PresentationIndexConfiguration.properties` for editing on your Content Store host, and make sure any **`types`** entry contains the value **`section`**. If there is no **`types`** entry in the file (or if there was no **`PresentationIndexConfiguration.properties`** file and you have created one), then add the following setting to the file:

```
types=section,article
```

The Content Store must then be restarted.

For general information about editing Content Store configuration files, see Configuring The Content Engine.

Parameters

`name:` ***String***
The name of the section from which content items are to be returned. You must specify the section's **unique name**, not the display name. If no section name is specified, then the context section is used.

`includeSubsections:` ***Boolean***
If set to **`true`**, then the result set includes content items belonging to the subtree of the specified section as well as the section itself. The default is **`false`**.

`publication:` ***String***
The name of the publication to search for the specified section. The default is the current publication.

5.4.7 Type

```
type(
  name: String
  names: [String!]
)
```

Returns all content items of a specified type, or of specified types.

Parameters

You must supply either the **`name`** parameter or the **`names`** parameter, but not both.

`name:` ***String***
The name of a content type. Only content items of the specified type are returned.

`names:` **`[`*****`String!`***`]`**
An array containing the names of content types. Only content items of the specified types are returned.

5.4.8 Tag

```
tag(
  tag: String!
)
```

Returns all content items tagged with a specified tag.

Parameters

The **tag** parameter is required.

tag: ***String!***
The **scheme** (internal identifier) of the tag to search for. You must specify the tag's scheme, not it's display name. The scheme must be preceded by a **tag:** prefix.

Examples

```
tag(tag: "tag:tomorrowonline@escenic.com,2017:football")
```

Returns all content items tagged with a **Football** tag, identified by the scheme **tomorrowonline@escenic.com,2017:football**.

5.4.9 Shared Tags

```
sharedTags
```

Returns all content items that share one or more tags with the current content item. This filter can only return data when the context object is a content item, and only if it has tags. If the context item is a section or if it is a content item with no tags then **sharedTags** will always return 0 results.

5.4.10 Field

```
field(
  name: String!
  value: String!
)
```

Returns all content items where the specified field contains the specified value. **Field** in this context has a very specific meaning: it means an **indexed Solr field**. In order to use this filter effectively you need to know what is indexed in your Solr schema. The list of available indexed fields may not include all the fields defined in your content types, and may include fields manufactured by Solr that do not exist in the content types. In fact, the filter is primarily intended for searching by fields that only exist in the Solr schema.

Parameters

The **name** and **value** parameters are both required.

name: ***String!***
The name of the indexed field to search for.

value: ***String!***
The value to be compared with the contents of the specified field.

Examples

```
field(name: "video_relation_in_main_b", value: "true")
```

Returns all content items with a field called **video_relation_in_main_b** in the Solr index, which is set to **true**. In order to use such a filter, you have to know that Solr indexes a field called **video_relation_in_main_b**.

5.4.11 Related

```
related(
  relation: String
  relations: [String!]
  type: String
  types: [String!]
  offset: Int
  count: Int
)
```

Returns content items related to the current content item. In other words, this filter can only return data when the context object is a content item. If the context item is a section then **`related()`** will always return 0 results.

Unlike most other data source filters, **`related()`** cannot be executed inside an **`and`** or **`or`** filter. This is because it is not implemented using Solr like the other filters. **`related()`** queries the Content Store directly.

> Note the following:
>
> - The **`datasource()`** function's **`sort`**, **`offset`** and **`count`** parameters do not work on results returned by the related() filter.
> - When working in the data source query editor, you will probably notice that the **`related()`** filter returns much less information about each result than the other Solr-based filters. This has no practical consequences. When a saved data source is executed by the **`datasource()`** function in a data retrieval query, the results are returned in the same format irrespective of how they were displayed in the data source query editor.

Parameters

If no parameters are specified, then **`related()`** returns **all** content items related to the current content item. The scope of the filter can be narrowed by specifying parameters. The **`relation`** and **`relations`** parameters are mutually exclusive: you are only allowed to specify one of them. The **`type`** and **`types`** parameters are also mutually exclusive.

`relation: String`
The name of a relation. Only related items belonging to the specified relation will be returned.

`relations: [String!]`
An array containing the names of relations. Only related items belonging to the specified relations will be returned. The sequence in which relations are specified is reflected in the order of the results.

`type: String`
The name of a content type. Only related items of the specified type will be returned.

`types: [String!]`
An array containing the names of content types. Only related items of the specified types will be returned. The sequence in which types are specified is reflected in the order of the results.

`offset: Int`
An offset to be applied to the initial result set. If the initial result set contains 4 results and you specify an **`offset`** of 1, then the first result will be omitted, leaving a final result set of 3.

`count`: ***Int***
A size limit to be applied to the initial result set. If the initial result set contains 4 results and you specify a **`count`** of 2, then only the first two results will be included.

Examples

All these examples assume that the current content item has a **`stories`** relation containing 3 items (a, b and c) and a **`media`** relation containing 2 items (d and e)

```
related(relations: ["stories", "media"])
```

Returns a, b, c, d and e.

```
related(relations: ["media", "stories"])
```

Returns d, e, a, b and c.

```
related(relations: ["stories", "media"], offset: 1)
```

Returns b, c, d and e.

```
related(relations: ["stories", "media"], count: 2)
```

Returns a and b.

```
related(relations: ["stories", "media"], offset: 1, count: 2)
```

Returns b and c.

```
related(relation: "stories", offset: 1)
related(relation: "media", offset: 1)
```

Returns b, c and e.

The purpose of the related() filter

You may wonder why the **`related()`** filter has been implemented as part of data sources: it doesn't work in the same way as other data source filters (using Solr), and it can't be combined with them using **`and`**, **`or`** or **`not`**. Moreover, you can easily access a content item's relations from a content retrieval query, so why is it needed?

The **`related()`** filter allows you to access a content item's relations in a more flexible way than is possible by following the relations in a content retrieval query. You can use it, for example, to construct a list of related content items drawn from a sequence of relations: if the first relation is empty or doesn't contain enough content items, then the next relation in the list is used, and so on.

6 Working with the Recipe

The Cook is responsible for retrieving all the data required to build a publication web page. In order to carry out this job, the Cook follows a **recipe**. The default recipe delivered with the start pack is a very small Javascript file, **`cue-front/recipe/recipe.js`**.

The reason the recipe is so small is that all the recipe functionality is actually provided by recipe **extensions**. The extensions are Javascript modules included by the recipe, most of which are downloaded from CCI Europe's NPM server, **`npm.escenic.com`**. The extensions that need to be downloaded from **`npm.escenic.com`** are listed in the file **`cue-front/recipe/package.json`** along with various other Javascript dependencies.

The recipe is responsible for:

- Retrieving data from all back-end systems: the Content Store, the Solr server plus any other external systems used by your application
- Filtering and organizing the retrieved data to produce the final JSON data structure delivered to the Waiter.

Much of the recipe's work is done by GraphQL queries, and therefore a great deal of customization work can be done using GraphQL (see section 3.2) and data sources (see chapter 5). At many installations, the recipe itself will never need to be modified.

Some customizations, however, require changes to the recipe, for example:

- **Retrieving data from external systems:** you might need to get sports results or weather data from an external web service.
- **Restructuring the output JSON data:** your Waiter might be an existing front end system that requires the JSON data to be supplied in a predefined format. GraphQL supports simple modifications to the output structure, such as omitting elements and renaming, but not complex reorganization.

Such changes can usually be made by writing your own extension.

6.1 Making a Recipe Extension

A recipe extension is a node module that can be called by the Cook's recipe, and provides some specific functionality to the recipe. The extensions are loaded by the **`ExtensionLoader`** object. A recipe extension must export an **extension object** that exposes the following methods and properties to the recipe:

`constructor(config, context)`
The constructor creates and initializes the extension object. **`ExtensionLoader`** passes in two parameters to the constructor:

`config`
Configuration values (if any) for this extension.

`context`
The **context object**, an object containing information about the current request. For details see section 6.1.2.

This method is **required**.

`extendSchema(assembler, model)`
This method can be used to add extensions to the GraphQL schema (for retrieving data from an external web service, for example). **`ExtensionLoader`** passes in two parameters to this method:

`assembler`
The SchemaAssembler object.

`model`
The CUE Front GraphQL model.

This method is **optional**. Use it if you want your extension to extend or modify the GraphQL schema.

`priority: number`
Determines when this extension is executed by the recipe (that is, when its **`run()`** method is called). All extensions **may** be assigned a priority, which determines their position in the execution order. Extensions are executed in the following order:

1. From the highest priority (that is, the **lowest** number) to the lowest priority (that is, the **highest** number). In other words, **`priority: 10`** is higher than **`priority: 20`** and will be executed first.
2. All unprioritized extensions are executed in alphabetical order.

Should more than one extension be assigned the same priority, then they are executed in alphabetical order.

> Priority can also be specified in the configuration file. A priority specified in the configuration file will take precedence over the value specified via this property.

This property is **optional**. Use it if you want the recipe to be able to control when your extension is executed.

`pattern: string`
A regular expression for testing **`remainingPath`**. **`remainingPath`** is a property of the context object, and is the last part of the original request URL that has not yet been resolved by any other recipe extension. If **`pattern`** is specified, then it is used to test the **`remainingPath`**, and the extension's **`run()`** method is only executed if **`remainingPath`** matches the regular expression. If **`run()`** is successfully executed, then the matched part of the **`remainingPath`** is regarded as resolved and removed from the **`remainingPath`**. If **`pattern`** is not specified then no such test is performed before executing the extension's **`run()`** method.

This property is **optional**. Use it if you want your extension to be triggered by a component in the request path.

`constraint(context): boolean`
This method can be used to enforce additional constraints on the execution of the extension. You can use it, for example, to limit the extension so that it is only used for a specific content type. If a **`constraint()`** method is specified and returns **`false`**, then the extension's **`run()`** method will not be executed. **`ExtensionLoader`** passes in one parameter to this method:

`context`
The **context object**, an object containing information about the current request. For details see section 6.1.2.

This method is **optional**. Use it if you want to constrain the circumstances in which your extension is used.

`run(recipe, context, done)`
This method actually executes the extension's functionality. It must return its results via the **`done`** callback function to enable asynchronous communication. **`ExtensionLoader`** passes in three parameters to this method:

`recipe`
The recipe object itself, created in **`recipe.js`**. For a detailed description of the recipe object, see section 6.1.1.

`context`
The **context object**, an object containing information about the current request. For details see section 6.1.2.

`done`
The callback method to be used for returning the extension's results. If the method does not need to return any actual results (if, for example, the extension's purpose is just to modify the **`context`** object in some way) then it must return **`done(true)`** to indicate successful execution, or **`done(false)`** in the case of failure. If a result set or **`done(true)`** is returned, then **`ExtensionLoader`** will call the next extension.

This method is **required**.

6.1.1 The Recipe Object

The supplied **`recipe.js`** creates a valid recipe object for you. If you are not using the supplied **`recipe.js`** and are creating the recipe object in some other way, it needs to have the following properties:

`log`
A Bunyan logger object.

`config`
An object containing the configuration parameters needed by the extension. By default, this object is created by loading the entries for this extension from the **`recipedata/extensions`** section of the **`cook-config.yaml`** file.

`assembler`
A new **`@escenic/cue-front-graphql/SchemaAssembler`** object, that has been initialized and populated with a base schema model.

`assemblerModel`
An **`@escenic/cue-front-graphql/model`** object.

`queryLoader`
An **`@escenic/cue-front-query-loader`** object, initialized with the base queries for the recipe.

6.1.2 The Context Object

The context object is constructed by the **`ExtensionLoader`**. It has the following properties:

request
A request object containing information about the current HTTP request.

log
A Bunyan logger object.

config
An object containing the configuration parameters needed by the extension. By default, this object is created by loading the entries for this extension from the **recipedata/extensions** section of the **cook-config.yaml** file.

cook
The CUE Front Cook.

schema
The assembled GraphQL schema object.

model
The model type for the current request.

query
An object with the name, GraphQL query string and path of the request.

resolvedRemainingPath
The remaining part of the request URL that has not yet been resolved.

Only the **request** needs to be set by **recipe.js**. All the other properties of the context object can be derived from the **recipe** object.

6.1.3 Extension Configuration

The extension and its location must be declared in the **recipedata/extensions** section of the **cook-config.yaml** file. For example:

```
recipedata:
...
  extensions:
    ...
    - name: 'path-to/my-cue-front-extension'
...
```

The declaration must include either a **name** or a **path** parameter(or both), and one of them must specify the path of the extension (either a single file or a folder containing the extension files).If you specify both parameters, then **path** should contain the path, and **name** should contain just the name. For example:

```
recipedata:
...
  extensions:
    ...
    - name: 'my-cue-front-extension'
      path: 'path-to/my-cue-front-extension'
...
```

If you register your extension as an NPM module, then you will not need to specify a path – the name (including an optional scope prefix) is sufficient information for NPM to locate your extension. For more about this, see the NPM documentation.

If you need to control the order in which your extension is executed, then you can do so by specifying a numeric **`priority`** parameter:

```
recipedata:
...
  extensions:
    ...
    - name: 'my-cue-front-extension'
      path: 'path-to/my-cue-front-extension'
      priority: 10
...
```

Extensions with a **`priority`** setting are executed first, in priority order (low number to high number), followed by all unprioritized extensions (in alphabetic order). None of the default extensions are prioritized by default.

In addition to **`name`**, **`path`** and **`priority`** you can optionally add a **`config`** property. You can include any configuration properties you like as children of the config property. For an extension intended to retrieve data from an external web service, for example, you would probably want to add a property for specifying the web service URL:

```
recipedata:
...
  extensions:
    ...
    - name: 'my-cue-front-extension'
      path: 'path-to/my-cue-front-extension'
      priority: 10
      config:
        result-service-url: 'http://my-result-service.com/'
...
```

7 Cleaver Image Filters

The Cleaver can apply filters to the images it handles. By default, the Cleaver does the following:

1. Retrieves a requested image from the Content Store
2. Crops and scales the retrieved image as specified in the URL parameters supplied with the request.
3. Saves the prepared image in its cache
4. Returns the the prepared image to the client that requested it.

It can, however, optionally apply filters to the prepared image before it is cached (between steps 2 and 3).

The Cleaver does not, however, have any built-in image filtering functionality: all it does is provide a convenient mechanism for running external image processors such as ImageMagick. The image processor must already be installed in the Cleaver's local environment. If, for example you want to use the Cleaver to apply ImageMagick filters to cropped images, then you must first of all make sure that ImageMagick is installed in the same environment as the Cleaver. If you are running the Cleaver on bare metal, then ImageMagick must be installed in the same machine. If you are running the Cleaver in a Docker container, then ImageMagick must be installed in the same container.

7.1 Filter Configuration

In order for the Cleaver to be able to perform any filtering, you must configure it correctly, by adding configurations to your **`cleaver-config.yaml`** file. For Docker-based installations, you will find this file in your installation's **`docker`** folder (**`docker/cleaver-config.yaml`**). For bare metal installations you will find it in the **`/etc/escenic`** folder (**`/etc/escenic/cleaver-config.yaml`**).

All filtering configurations are grouped under a **`filters`** entry:

```
filters:
  - name: clean-metadata
    execute: auto
    description: "Removes metadata exif information"
    command: "convert -strip {input} -strip {output}"
  - name: watermark
    execute: auto
    description: "Adds a Escenic watermark to the picture"
    command: "composite -dissolve 20% -gravity center /path/to/watermark.png {input}
 {output}"
    extensions: [jpg,  JPG, jpeg, JPEG]
   ...etc.
```

and may contain the following settings:

`name`
: **Required.** The name you want to give to the filter. The name setting must be preceded by a **`-`** (hyphen) to indicate the start of a new filter item.

`execute`
Optional. If you specify **`execute: auto`**, then this filter will be automatically applied to all images handled by the Cleaver (unless excluded by the **`extensions`** setting – see below). If you omit this setting, then the filter will only be applied if explicitly requested.

`description`
Optional. A brief description of what the filter does.

`command`
Required. The operating system command required to execute the filter operation. In the example shown above, the command:

```
convert -strip {input} -strip {output}
```

uses the ImageMagick **`convert`** utility to remove EXIF metadata from images. Whatever command you use, you must include the placeholders **`{input}`** and **`{output}`** in the correct positions in the command.

`extensions`
Optional. A comma-separated list of filename extensions, enclosed in square brackets (**`[`** and **`]`**). If specified, then the filter will only be applied to images with one of the specified extensions. If **`extensions`** is not specified, then the filter is applied to all images.

If you configure more than one filter, then they will be executed in the order they are specified in the configuration file. In some cases, the order in which filters are executed may be significant, so you should think about this when editing your filter configurations.

The default **`cleaver-config.yaml`** configuration file contains a number of predefined filter configurations, most of which make use of ImageMagick utilities, so in order to use them, you need to make sure ImageMagick is installed together with the Cleaver. Two of the predefined configurations, however (**`base64`** and **`guetzli`**) make use of the Gueztli compression tool, so in order to use them you need to make sure that this tool is installed together with the Cleaver. For information about this tool, see https://github.com/google/guetzli.

You can, of course create filters that make use of other image processing tools. Any tool with a command line interface can be used to implement a Cleaver filter.

8 Bare Metal Installation

This chapter describes how to install each of the CUE Front components on a clean Ubuntu 16.04 system. It is only of interest to:

- Developers who want to install all the CUE Front components on their machine, but don't want to use Docker.
- System administrators responsible for installing CUE Front in their test and/or production environments.

These instructions do not assume that you are necessarily installing all the components in the same machine: they will work even if you install each component on a different machine.

> You are strongly recommended to use one of the other installation methods if you are installing CUE Front for development purposes / personal use.

8.1 Install Cook

To install the Cook on Ubuntu 16.04:

1. Install the following prerequisites:

```
sudo apt-get update
sudo apt-get install -y apt-transport-https curl make unzip nodejs nodejs-legacy
```

2. Install the Cook itself:

```
curl --silent https://apt.escenic.com/repo.key | sudo apt-key add -
sudo echo "deb http://user:pass@apt.escenic.com stable main non-free" > /etc/apt/
sources.list.d/escenic.list
sudo apt-get update
sudo apt-get install -y escenic-cook
```

   where *user* and *pass* are your credentials for accessing the CCI Europe APT repository. If you do not have any such credentials, please contact Escenic Support.

3. Configure the Cook as described in section 8.1.1.
4. Start the Cook by entering:

```
cook -c /etc/escenic/cook-config.yaml
```

   or start it as a service by entering:

```
service cook start
```

You should now be abe to find the Cook by starting a browser and visiting **`http://localhost:8101`** (assuming you have specified 8101 as the Cook's **`listen`** port – see section 8.1.1).

> Note that you can override Cook's configuration file settings using command line options. To list the available command line options, enter:
>
> ```
> cook -h
> ```

8.1.1 Configuring Cook

You will find the Cook configuration file in the default location **/etc/escenic/cook-config.yaml**.

Make sure the following parameters in the configuration file are set and not commented out:

resolverURI
Set this to point to your Content Store's resolver web service. For example:

```
resolverURI : "http://my-escenic.com:8080/resolver"
```

The Content Store must be version 6.0 or higher, and its **resolver** web application (supplied with the Content Store) must have been deployed.

recipeLocation
Set this to point to the recipe the Cook is to use. For example:

```
recipeLocation: "/opt/mycompany/website/recipe/myrecipe.js"
```

cleaverURI
Set this to the URL of your Cleaver. If you are installing the Cleaver on the same machine, then it will be **localhost** plus the port number you specify in **cleaver-config.yaml** (the **listen** parameter) plus **/image-version**. For example:

```
cleaverURI: "http://localhost:8102/image-version"
```

listen
Set this to the port number on which you want the cook to listen. For example

```
listen: 8101
```

servers
You need to add **host**, **username** and **password** settings for your Content Store here. For example:

```
servers:
  - host: "my-escenic.com:8140"
    username: "mytestuser"
    password: "highly-secret"
```

Make sure all four lines are uncommented. The **host** setting must include both host name and port number. The Content Store user you specify here only needs to have read access, but must have read access to all your publications' sections and content types. If you want to be able to support cross-publishing, then the user must have access to all the publications from which content might be selected.

menuWebserviceURI
If your Content Store installation includes the Menu Editor plug-in, then uncomment this line and set it to point to the Content Store's menu web service. For example:

```
menuWebserviceURI: "http://my-escenic:8080/menu-webservice"
```

The Tomorrow Online demo publication makes use of the Menu Editor plug-in, so if it is not installed in your Content Store (or if you don't configure the web service URI here), then no menu will be displayed on the Tomorrow Online web site.

log-file
Make sure that this is set to point to a writeable location. For example:

```
log-file: "/var/log/escenic/cook.log"
```

It is important that you specify the log file location as an absolute path.

8.2 Install Cleaver

To install the Cleaver on Ubuntu 16.04:

1. Install the following prerequisites:

```
sudo apt-get update
sudo apt-get install -y apt-transport-https curl make unzip python3 python3-pip
```

2. Install the Cleaver itself:

```
curl --silent https://apt.escenic.com/repo.key | sudo apt-key add -
sudo echo "deb http://user:pass@apt.escenic.com stable main non-free" > /etc/apt/
sources.list.d/escenic.list
sudo apt-get update
sudo apt-get install -y escenic-cleaver
```

   where *user* and *pass* are your credentials for accessing the CCI Europe APT repository. If you do not have any such credentials, please contact Escenic Support.

3. Configure the Cleaver as described in section 8.2.1.
4. Start the Cleaver by entering:

```
/usr/share/escenic/escenic-cleaver/bin/python3 /usr/share/escenic/escenic-cleaver/
cleaver.py -c /etc/escenic/cleaver-config.yaml
```

You should now be abe to find the Cleaver by starting a browser and visiting **`http://localhost:8102`** (assuming you have specified **`8102`** as the Cleaver's **`listen`** port – see section 8.2.1).

Note that you can override Cleaver's configuration file settings using command line options. To list the available command line options, enter:

```
cleaver -h
```

8.2.1 Configuring Cleaver

You will find a Cleaver configuration file in the default location **`/etc/escenic/cleaver-config.yaml`**.

Make sure the following parameters in the configuration file are set and not commented out:

`download_dir`
The path of a folder in which the Cleaver can cache downloaded images. For example:

```
download_dir: "/var/cache/cleaver/"
```

`listen`
Set this to the port number on which you want the Cleaver to listen. For example

```
listen: 8102
```

`servers`
You need to add **`host`**, **`username`** and **`password`** settings for your Content Store here. For example:

```
servers:
  - host: "my-escenic.com:8140"
    username: "mytestuser"
    password: "highly-secret"
```

log-file

Make sure that this is set to point to a writeable location. For example:

```
log-file: "/var/log/escenic/cleaver.log"
```

It is important that you specify the log file location as an absolute path.

8.3 Install Waiter and Demo Publication

To install the Waiter and a demo publication on Ubuntu 16.04:

1. Install the following prerequisites:

```
sudo apt-get update
sudo apt-get install -y unzip composer php php-mbstring xsltproc xmlstarlet
```

2. Download and unpack the CUE Front start pack as follows:

```
cd
curl -O https://user:password@maven.escenic.com/com/escenic/cook/cue-front-start-
pack/1.4.0-3/cue-front-start-pack-1.4.0-3.tar.gz
tar -xzvf  cue-front-start-pack-1.4.0-3.tar.gz
rm cue-front-start-pack-1.4.0-3.tar.gz
ln -s cue-front-start-pack-1.4.0-3 cue-front
cd cue-front
```

3. Build the downloaded project:

```
make
```

4. Copy the default Waiter configuration file supplied in the **cue-front/config** folder:

```
cd config
cp waiter-config.yaml.default waiter-config.yaml
```

5. Configure the Waiter as described in section 8.3.1.
6. Start the Waiter and the Patternlab style guide by entering:

```
make start -C waiter
make start -C styleguide
```

You should now be able to find the front page of your publication by starting a browser and visiting **http://***host-name***:8100** (where *host-name* is one of the publication host names you specified in step 5). The publication will be rendered using the Twig templates you configured with the **publications/templateDir** parameter.

To view the style guide, open a browser and go to **http://localhost:8103/**. For further information, see section 3.3.

8.3.1 Configuring Waiter

Open the **waiter-config.yaml** file you have created in an editor, and set the following parameters:

cookBaseURL
Set this to URL of your Cook. For example:

```
cookBaseURL: "http://my-cook.com:8101/"
```

publications
Add sub-entries defining the publications you want the Waiter to serve. For example:

```
publications:
  - name: tomorrow-online
    hostNames:
      - localhost
    templateDir: ../templates/_base
  - name: mypub
    hostNames:
      - mypub.com
    templateDir: ../templates/mypub
  - name: myotherpub
    hostNames:
      - myotherpub.com
      - myotherpub.net
    templateDir: ../templates/myotherpub
```

For each publication to be served, you need to specify:

- name
Set this to the name of the CUE publication. For example:

```
  - name: "dailynews"
```

hostNames
Here you can add a list of host names you want to be associated with the publication. For example:

```
    hostNames:
     - "www.dailynews.net"
     - "www.dailynews.com"
```

templateDir
Here you must specify the path of the folder containing the Twig templates to be used for rendering this publication. For example:

```
    templateDir: "../templates_dailynews"
```

devmode
Add this and set it to **true**:

```
devmode: true
```

(Doing this ensures that links in the publication will work in your local set-up.)

log-file
Make sure that this is set to point to a writeable location. For example:

```
log-file: "../log/waiter.log"
```

8.4 Install Fridge

To install the Fridge on Ubuntu 16.04:

1. Enter the following commands:

```
curl --silent https://apt.escenic.com/repo.key | sudo apt-key add -
sudo echo "deb http://user:pass@apt.escenic.com stable main non-free" > /etc/apt/
sources.list.d/escenic.list
sudo apt-get update
sudo apt-get install -y escenic-fridge
```

   where *user* and *pass* are your credentials for accessing the CCI Europe APT repository. If you do not have any such credentials, please contact Escenic Support.

2. Configure the Fridge as described in section 8.4.1.
3. Start the Fridge by entering:

```
fridge -c /etc/escenic/fridge-config.yaml
```

You should now be abe to find the Fridge by starting a browser and visiting **`http://localhost:8104`** (assuming you have specified **`8104`** as the Fridge's **`listen`** port – see section 8.4.1).

> Note that you can override Fridge's configuration file settings using command line options. To list the available command line options, enter:
>
> ```
> fridge -h
> ```

8.4.1 Configuring Fridge

You will find a Fridge configuration file in the default location **`/etc/escenic/fridge-config.yaml`**. You can either edit this file, or copy it to some other location first. If the configuration file is not in this default location, then you will need to specify its location when starting the Fridge using the start-up command's **`-c`** option.

Make sure the following parameters in the configuration file are set and not commented out:

`listen`
Set this to the port number on which you want the Fridge to listen. For example:

```
listen: 8104
```

`document-root`
The path of the folder containing the static files served by the Fridge. For example:

```
document-root: "/var/cache/escenic/fridge"
```

`log-file`
Make sure that this is set to point to a writeable location. For example:

```
log-file: "/var/log/escenic/fridge.log"
```

> It is important that you specify the log file location as an absolute path.

`proxy`
Set this to **`true`** if you want the Fridge to function as a proxy, otherwise set it to **`false`**. For example:

```
proxy: "true"
```

In order to make the Fridge function as a proxy, and start caching files in its **`document-root`** folder, you need to do two things:

- Set **`proxy`** to **`true`** as described above.
- Configure either the Waiter or the Cook to use the Fridge depending on what your use case is. To do this, you need to add the following setting in your **`waiter-config.yaml`** or **`cook-config.yaml`** file:

  ```
  proxy: http://fridge:8104
  ```

  where *fridge* is the Fridge's domain name or IP address.

8.4.2 Change Log Daemon Setup

A Change Log Daemon can be installed and set up to ensure that the Fridge contents are always kept up-to-date. A script to be used by the Change Log Daemon for this purpose is included in the Fridge installation.

First, Install a Change Log Daemon as described in Change Log Daemon Installation. When you get to the configuration stage described in Configure the Daemon, set the following properties in the **`Daemon.properties`** configuration file:

`url`
The URL of the Content Store change log for the specific publication you are interested in:

```
url=http://content-engine-host/webservice/escenic/changelog/
publication/publication-id
```

`username`
A username for accessing the Content Engine. You only need read access, so you can use the same user account as you have used for the Cook and the Cleaver.

`password`
The password for the **`username`** you specified.

`proxy`
This is an additional property that is used by the update script delivered with the Fridge, it is not a standard Change Log Daemon property. Set it to point to the Fridge. It should be set to the same value as the **`proxy`** property in your Cook configuration file:

```
proxy: fridge:8104
```

where *fridge* is the Fridge's domain name or IP address.

Once the Change Log Daemon is correctly configured, then you should start it as follows (assuming that you have installed both the Change Log Daemon itself and the Fridge in the documented locations):

```
cd /usr/share/escenic/escenic-fridge/changelog
/opt/escenic/changelog-daemon-2.0.0-3/changelog.sh
```

In other words, the **`changelog.sh`** script **must** be run from the Fridge installation's **`changelog`** folder in order to locate various configuration files and the Fridge's **`postChanges.sh`** script.

9 The Setup Tool

The CUE Front **setup** tool is intended to simplify the configuration of the CUE Front components. Currently, **setup** only supports Docker-based installations. **setup** is a command line tool that runs in a Docker container.

If you don't use **setup**, then configuring CUE Front involves editing a number of different configuration files. In some cases, the same value must be specified in several places, both within the same file and in different files. It is relatively easy to make mistakes during this process. **setup** simplifies this task by issuing a series of prompts and using your responses to generate all these files. The tool carries out this task in three different phases:

add
: Prompt for values, create a configuration set folder and save the responses in a **blueprint.yaml** file in the folder (**cue-front/setup/configs/***configuration-name***/blueprint.yaml**, for example).

generate
: Generate a set of configuration files from **blueprint.yaml** and save them together with **blueprint.yaml** in the configuration folder.

use
: Copy **docker-config.yml** file from the configuration folder to the **cue-front** root folder, available for use the next time CUE Front is started, and verify the contents (by checking for, example, that specified URLs exist). By default, the copied file is renamed to **docker-***configuration-name***.yml**, so the **cue-front** root folder can contain many such files, allowing you to easily switch between configurations by specifying a different configuration file with the startup command.

setup has three corresponding subcommands that execute these phases:

add
: Executes the **add**, **generate** and **use** phases for a named configuration set.

generate
: Executes the **generate** and **use** phases for a named configuration set.

use
: Executes the **use** phase for a named configuration set. This command also has a **-d** option, which you can use to make set the specified configuration as the default.

In addition to **add**, **generate** and **use**, setup has the following subcommands:

list
: Lists the names of all available configuration sets.

check_configuration
: Checks the YAML syntax of the current configuration set.

verify_build_parameters
: Verifies that the CCI Europe download credentials required by CUE Front are available.

help
: Displays a short help message listing **setup**'s subcommands.

The **setup** command must always be run in the **cue-front** folder.

9.1 Creating a New Configuration

To create a new configuration, use the **add** subcommand as follows:

```
docker-compose -f setup.yml run setup add configuration-set
```

where *configuration-set* is the name you want to use for the new configuration.

setup displays the following prompts:

Enabled services
Specify the CUE Front services you want to be enabled. Press the up and down arrows to move the focus through the options, and press the space bar to select or deselect services. You can use the **all** and **none** options to select and deselect all the service options. Press **Enter** when you are satisfied with your selection.

Configuration
Specify what kind of configuration you want to carry out:

Quick
Only the most important configuration settings are displayed - default values are used for all the other settings.

Advanced
All configuration settings are displayed.

setup then displays a further sequence of prompts, requesting parameter values for the services you have decided to enable. How many prompts are displayed for each service depends on whether you selected **Quick** or **Advanced** configuration.

When you have responded to all the prompts, **setup** does the following:

1. Saves your responses in *path*/**cue-front/setup/configs/***configuration-set*/**blueprint.yaml**.
2. Generates a set of configuration files from the blueprint and saves them in the same folder.
3. Copies **docker-config.yml** from the *configuration-set* folder configuration files to *path*/**cue-front** and renames it to **docker-***configuration-set***.yml**.

This means you can now start CUE Front using the configuration you created by entering the command:

```
docker-compose -f docker-configuration-set.yml up -d
```

9.2 Switching Configurations

If you have created more than one configuration, you can switch between them as follows:

1. Stop and remove the current set of CUE Front containers:
   ```
   docker-compose -f docker-old-configuration-set.yml down
   ```

2. Restart specifying a different configuration set:

   ```
   docker-compose -f docker-new-configuration-set.yml up -d
   ```

   where *new-configuration-set* is the name of the configuration you want to switch to.

9.3 Making a Default Configuration

You can make one of your configurations the default configuration by entering:

```
docker-compose -f setup.yml run setup use -d configuration-set
```

This copies **docker-***configuration-set***.yml** to **docker-compose.yml**, which means you can then start and stop CUE Front using this configuration without specifying a configuration file:

```
docker-compose up -d
```

and

```
docker-compose down
```

9.4 Modifying a Configuration

To modify an existing configuration, use the **edit** subcommand as follows:

```
docker-compose -f setup.yml run setup edit configuration-set
```

where *configuration-set* is the name of the configuration you want to modify.

The **edit** subcommand basically works in the same way as the **add** subcommand – it displays the same sequence of prompts. In this case, however, the default values offered by **setup** are not the standard defaults, but the configuration set's existing values. This means you can just accept all defaults except for the specific values you want to modify.

> If the configuration set you have modified has been set as the default configuration, then you will also need to run the **setup use** command again to update the default configuration (see section 9.3).

9.5 Multi-publication Support

The prompts displayed by **setup** do not offer the option of creating more than one publication. CUE Front, however, is designed to support multiple publications. Once you have used the **setup** utility to get up and running with a single publication, you can switch to a multi-publication configuration by simply editing a few files (assuming that your Content Store hosts more than one publication).

When you create a CUE Front configuration using the **setup add** command, the prompts displayed by the setup utility are read from a file called **setup/defaults/blueprint.yaml**. The configuration that is generated by the command is stored in a folder called **setup/configs/***config*, where *config* is the name of your configuration. The configuration consists of a **blueprint.yaml** file, which contains your responses to the prompts displayed by **setup add**, and a set of program-

specific configuration files: **cook-config.yaml**, **fridge-config.yaml**, **cleaver-config.yaml**, **docker-compose.yml** and **waiter-config.yaml**. These configuration files are created by copying a set of default files from the **setup/defaults** folder and using the responses in **blueprint.yaml** to override selected settings.

Since **setup add** cannot prompt for multiple publications and **blueprint.yaml** cannot store settings for multiple publications, the recommended way to add extra publications to your setup is as follows:

1. Add the multiple publication settings to **setup/defaults/waiter-config.yaml** by manually editing the file.
2. Remove the publication prompt definitions from **setup/defaults/blueprint.yaml** by manually editing the file.
3. Run **setup add** to generate a new configuration containing the multiple publication definitions.
4. Add information about your new publications to the **nginx** configuration file, **waiter/docker/nginx.conf**.
5. Restart the Waiter and **nginx**.
6. Commit the changes you made in steps 1, 2 and 4 to your repository to make them available for other users.

9.5.1 Add multiple publication settings

Open **setup/defaults/waiter-config.yaml** in an editor. The first block of entries at the top of the file defines the publications the waiter is to serve, but there will only be entries for one publication:

```
publications:
  - name: 'my-publication-name'
    hostNames:
      - 'localhost'
    templateDir: '../templates/_base'
```

Replace these entries with entries defining all the publications you want the waiter to serve:

```
publications:
  - name: tomorrow-online
    hostNames:
      - localhost
    templateDir: ../templates/_base
  - name: mypub
    hostNames:
      - mypub.com
    templateDir: ../templates/mypub
  - name: myotherpub
    hostNames:
      - myotherpub.com
      - myotherpub.net
    templateDir: ../templates/myotherpub
```

The properties should be set as follows:

name
: The name of a publication in your Content Store.

hostNames
One or more entries, each of which is a domain name at which this publication is to be served.

templateDir
The folder containing the publication templates. For more about this, see section 3.4.

In the example shown above:

- **tomorrow-online** will be made available at **http://localhost:8100** and will look for templates in the **../templates/_base** folder.
- **mypub** will be made available at **http://mypub.com:8100** and will look for templates in the **../templates/mypub** folder.
- **myotherpub** will be made available at **http://myotherpub.com:8100** and **http://myotherpub.net:8100** and will look for templates in the **../templates/myotherpub** folder.

9.5.2 Remove publication prompt definitions

Open **setup/defaults/blueprint.yaml** in an editor and remove the following lines from the **waiter** section:

```
  publications-name:
    message: "Publication-name"
    default: "tomorrow-online"
  publications-hostNames:
    message: "Publication-hostname"
    default: "localhost"
```

This will prevent the **setup add** command from prompting for a publication name and host name.

9.5.3 Generate a new configuration

Run **setup add**:

```
docker-compose -f setup.yml run setup add configuration-set
```

where *configuration-set* is the name of your configuration.

Accept the displayed defaults (that is, the values you have specified previously).

9.5.4 Reconfigure nginx

Open **waiter/docker/nginx.conf** in an editor and add entries for your new publications. The publication definitions you add must match the definitions you have added to **setup/defaults/waiter-config.yaml**. The **server** section of the default **nginx.conf** included in the CUE Front start pack looks like this (with comments removed):

```
  server {
    listen       8100;
    server_name  localhost;

    root /srv/templates/_base;

    location / {
      try_files $uri @waiter;
    }
```

```
    location ~ \.css {
      root /srv/templates/_base;
    }

    location @waiter {
      include fastcgi_params;
      fastcgi_param SCRIPT_FILENAME /srv/waiter/index.php;
      fastcgi_pass   unix:/run/php/php7.0-fpm.sock;
    }
```

You need to make a copy of this section for each additional publication you want the Waiter to serve, and modify the highlighted fields (**`server_name`** and **`root`**). For a **`myotherpub`** publication served on **`myotherpub.com`** and **`myotherpub.net`**, you would need to add this **`server`** section:

```
  server {
    listen       8100;
    server_name  myotherpub.com myotherpub.net;

    root /srv/templates/myotherpub;

    location / {
      try_files $uri @waiter;
    }

    location ~ \.css {
      root /srv/templates/_base;
    }

    location @waiter {
      include fastcgi_params;
      fastcgi_param SCRIPT_FILENAME /srv/waiter/index.php;
      fastcgi_pass   unix:/run/php/php7.0-fpm.sock;
    }
```

9.5.5 Restart the Waiter

To restart the Waiter (and **`nginx`**, which runs in the same container as the Waiter), enter:

```
docker-compose restart waiter
```

You should now have access to all the publications you have defined.