CUE Front

Developer Guide

1.1.1-1

escenic

A CCI COMPANY

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 Engine and serves the content to clients as JSON data via an HTTP-based **content API**.

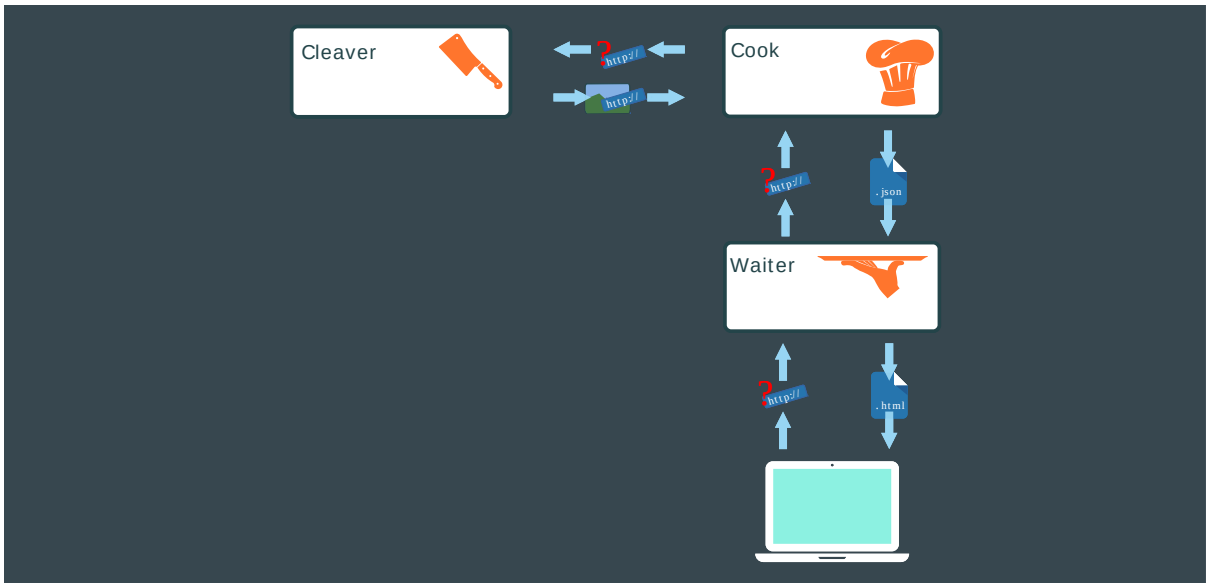
Cleaver
A back-end service that retrieves, crops and resizes images for the Cook.

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 Engine. They are free-standing entities that only communicate with the Content Engine 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 Engine's REST API or to a Fridge caching layer.

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

- **Technology independence:** Escenic'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 Engine, and required complicated caching strategies to avoid overloading the database. The CUE Front components are completely decoupled from the Content Engine and can be scaled separately. A complete copy of all the content in the Content Engine'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 Engine. 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 Engine 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 Engine, such deployments have no effect on the back end. Conversely, upgrading the Content Engine 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 Engine instances offline simultaneously without affecting published sites in any way (other than the lack of updates to the content).

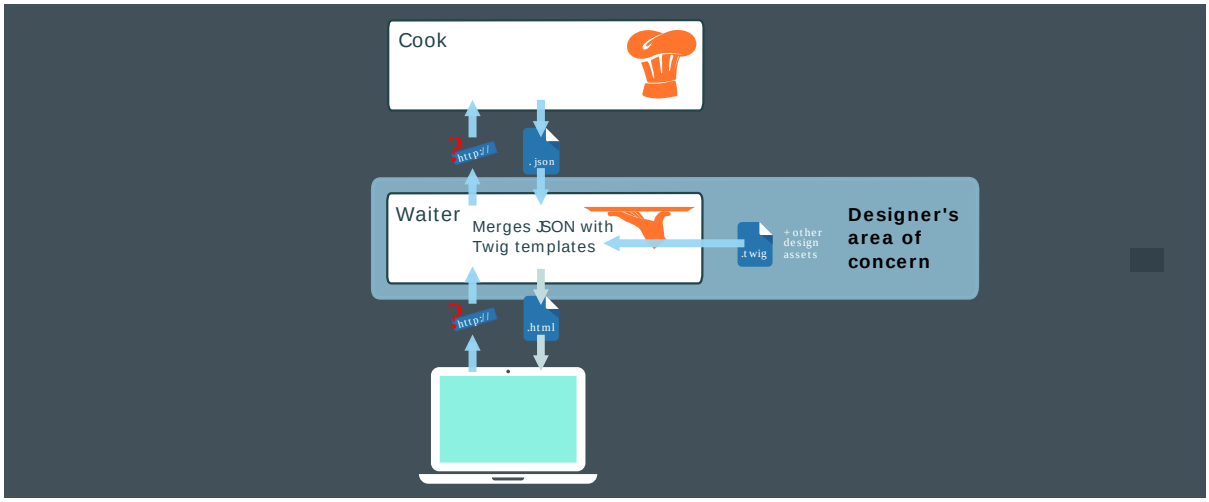
Breaking the presentation layer into separate services encourages separation of concerns: front-end developers/designers can work exclusively with the Waiter (or some other front-end service), and do not need to know anything about Cook or Cleaver. Similarly, back-end developers can concentrate on ensuring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

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.

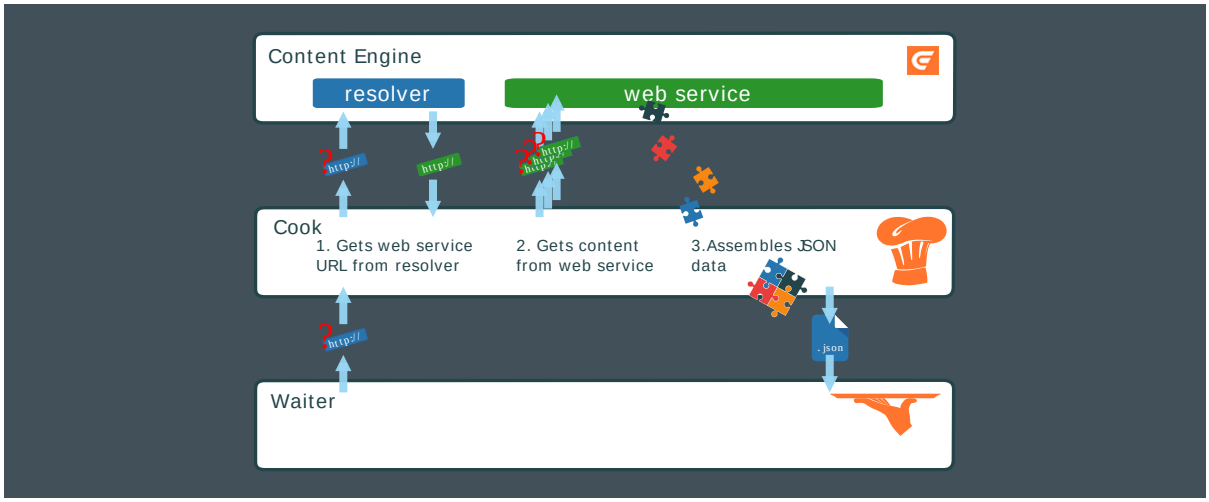
1.1.2 What is Twig?

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 Engine, but this complexity is hidden from the Waiter.



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

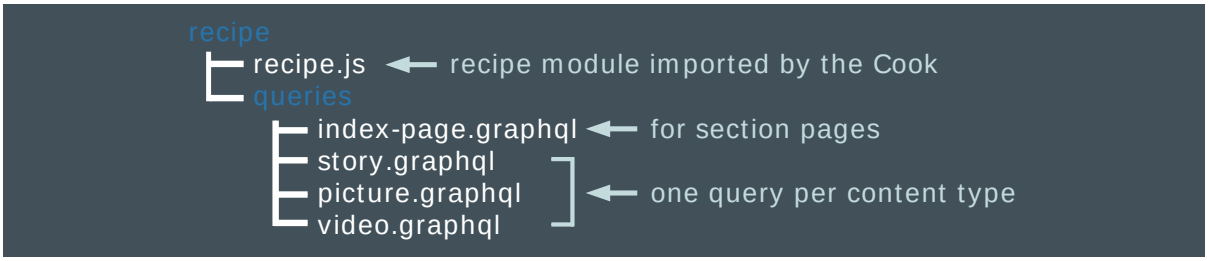
1. Sends the request URL to a Content Engine web service called **`resolver`**. The resolver converts this external "pretty" URL to an internal web service URL
2. Sends a request to the returned web service URL. The Content Engine web service returns data in the form of Atom XML resources. In order to 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 Engine 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 an application-specific Javascript code module used by the Cook to enable it to retrieve information from the Content Engine and/or other sources, and make it available in a useful form to the Waiter. The recipe consists of a generic "master" file called **`recipe.js`** plus 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 master 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 Escenic publication (see section 3.1). This means that creating a recipe for a new publication or family of related publications is mostly 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:

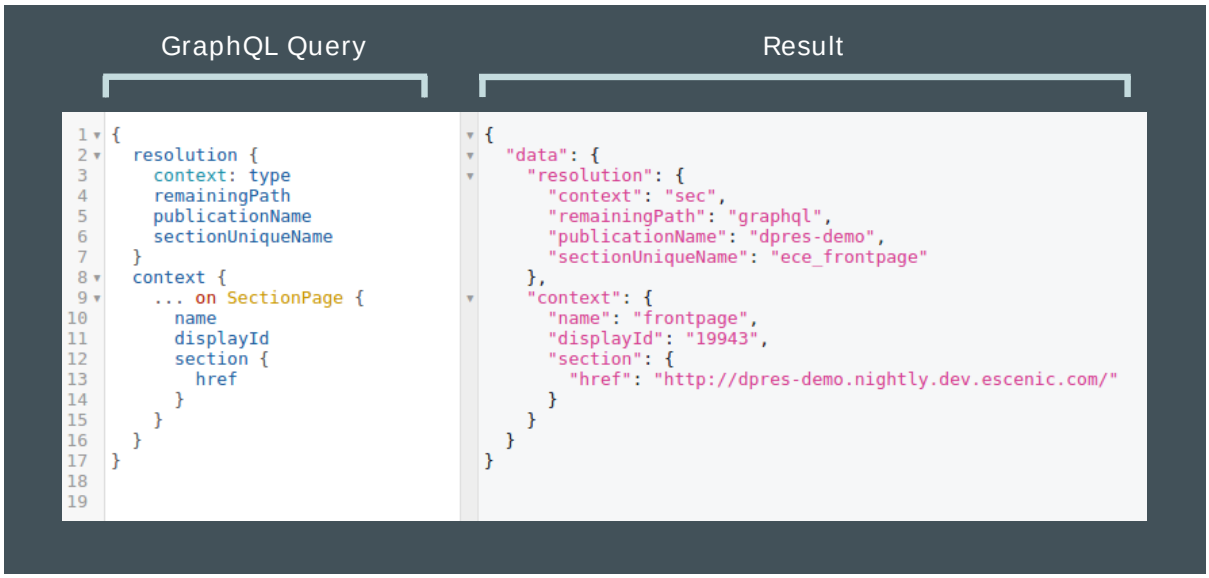
- The Waiter requires the output JSON data to be organized in a different way than the default output (which reflects the Content Engine's internal structure). GraphQL allows simple modifications to the output structure, such as omitting elements and renaming, but not complex reorganization.
- The Waiter requires data from sources other than the Content Engine 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 modified or extended. The supplied recipe has a number of built-in extension points which make this process relatively straightforward.

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 Escenic 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.

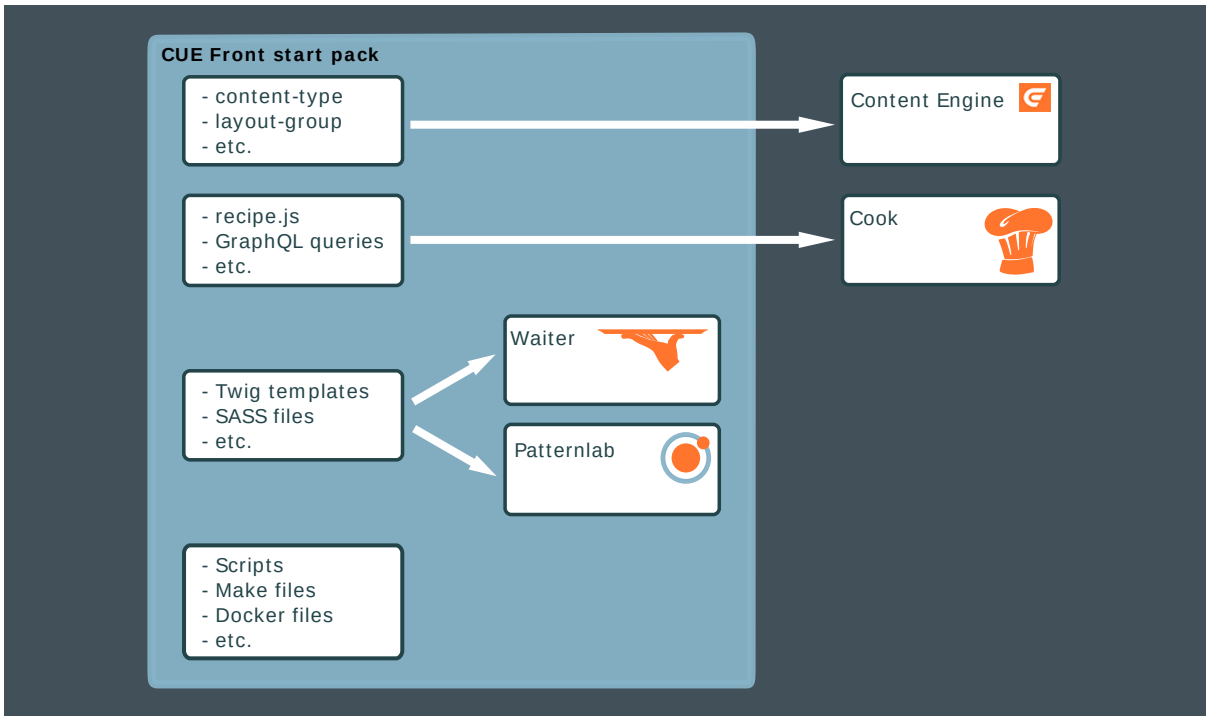
For more about this, see section 3.2.

1.2.3 What Does the Cleaver Do?

The Cleaver is an auxiliary service that handles images for the Cook. Images in Escenic 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 Engine, 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 Engine
- 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 Escenic 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 7. 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 Using Docker

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. Install the CUE Front components in Docker containers – see section 2.1.3

4. Upload the demo publication to your Content Engine if necessary – 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 is a **`docker.io`** package in the Ubuntu repositories, but it contains an old version and must not be used. You should install version 1.13 of **`docker`** and version 1.10 of **`docker-compose`** as follows.

1. Set up the Docker apt repository and install Docker:

   ```
   sudo apt-get install apt-transport-https ca-certificates
   curl -fsSL https://yum.dockerproject.org/gpg | sudo apt-key add -
   sudo add-apt-repository "deb https://apt.dockerproject.org/repo/ ubuntu-
   $(lsb_release -cs) main"
   sudo apt-get update
   sudo apt-get -y install docker-engine=1.13.0-0~ubuntu-xenial
   ```

   The above sequence of commands is a summary of the procedure described in the Docker documentation. If you get any problems installing Docker, or want more information, refer to this documentation.

2. Install **`docker-compose`**:

   ```
   sudo curl -L "https://github.com/docker/compose/releases/download/1.10.0/docker-
   compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
   sudo chmod +x /usr/local/bin/docker-compose
   ```

3. Create a user group called **`docker`**, and add your user to it. This gives you permission to run **`docker`** and **`docker-compose`** commands. If you don't do this, then you will have to prefix all your **`docker`** and **`docker-compose`** commands with **`sudo`**:

   ```
   sudo groupadd docker
   sudo gpasswd -a your-user-name docker
   sudo service docker restart
   newgrp docker
   ```

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/1.1.1-1/cue-front-start-pack-1.1.1-1.tar.gz
tar -xzvf  cue-front-start-pack-1.1.1-1.tar.gz
rm cue-front-start-pack-1.1.1-1.tar.gz
ln -s cue-front-start-pack-1.1.1-1 cue-front
cd cue-front
```

where *username* and *password* are your Escenic credentials. If you don't have a username and password, please contact Escenic 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 3.4.

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

2.1.3 Installing the CUE Front Components

Install and configure the CUE Front components as follows:

1. Create a hidden **.env** file in the **cue-front** folder containing the credentials you need to get the CUE Front components from the Escenic repository. You can do this by entering the following commands:

   ```
   echo 'APT_CREDENTIALS=username:password' > .env
   echo 'COOK_VERSION=1.1.1-1' >> .env
   ```

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

   > If you are installing CUE Front on Windows, then you need to add a third line to the **.env** file. Enter the following command in the Docker Quickstart Terminal:
   >
   > ```
   > echo 'COMPOSE_CONVERT_WINDOWS_PATHS=1' >> .env
   > ```

2. Enter the following command to install CUE Front in Docker:

   ```
   docker-compose build
   ```

   Make sure you are in the **cue-front** folder before you enter the command.

   If everything is OK, then the build process will take five or ten minutes to complete, and produces a lot of output in the terminal. If something is wrong then the command will probably terminate immediately, and output an error message.

3. Create configuration files by copying the supplied default files as follows:

   ```
   cp docker/defaults/cook-config.yaml docker/cook-config.yaml
   cp docker/defaults/cleaver-config.yaml docker/cleaver-config.yaml
   cp docker/defaults/waiter-config.yaml docker/waiter-config.yaml
   cp docker/defaults/fridge-config.yaml docker/fridge-config.yaml
   ```

4. Open three of the copied files for editing and set the following values:

   docker/cook-config.yaml

   resolverURI

   Uncomment and set to point to your Content Engine's resolver web service. For example:

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

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

servers

You need to add **host**, **username** and **password** settings for your Content Engine 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 Engine 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 Engine installation includes the Menu Editor plug-in, then uncomment this line and set it to point to the Content Engine's menu web service. For example:

```
menuWebserviceURI: "http://my-escenic.com: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 Engine (or if you don't configure the web service URI here), then no menu will be displayed on the Tomorrow Online web site.

docker/cleaver-config.yaml

servers

You need to add **host**, **username** and **password** settings for your Content Engine here (the same ones you used in **cook-config.yaml**). For example:

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

Make sure all four lines are uncommented.

docker/waiter-config.yaml

publications/name

Set this to the name of your Escenic publication — the demo publication is called **Tomorrow Online**, so, for example:

```
publications:
 - name: "tomorrow-online"
```

publications/hostNames

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

```
    hostNames:
     - "localhost"
     - "www.tomorrow-online.com"
```

5. If your Content Engine is running in a virtual machine on your PC, then you also 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 Engine, 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 Engine 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"
   ```

2.1.4 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 Engine.

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 Engine in the usual way. If you don't know how to do this, you will find instructions here. You only need to follow steps 1 - 7, the remaining steps are not required. Don't worry that the instructions specify the use of a **`.war`** file – the supplied **`.zip`** file will work.

2.1.5 Starting CUE Front

To start CUE Front, enter:

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

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

```
Creating network "cuefrontstartpack11012_default" with the default driver
Creating cuefrontstartpack11012_rsync_1
Creating cuefrontstartpack11012_fridge_1
Creating cuefrontstartpack11012_cleaver_1
Creating cuefrontstartpack11012_styles_1
Creating cuefrontstartpack11012_browsersync_1
Creating cuefrontstartpack11012_styleguide_1
Creating cuefrontstartpack11012_cook_1
Creating cuefrontstartpack11012_waiter_1
```

If you get problems at this point, the most likely reasons are:

- 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).
- There are syntax errors in one of the configuration files you edited. The YAML format used in the configuration files is very sensitive to white space errors - if you indent parameters too much or too little, it can cause errors.

In both cases, **`docker-compose`** will output error messages explaining the problem. In the case of syntax error messages, note that the line numbers in the error messages do not include comment lines.

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 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 cuefrontstartpack11012_waiter_1 ... done
Stopping cuefrontstartpack11012_cook_1 ... done
Stopping cuefrontstartpack11012_styleguide_1 ... done
Stopping cuefrontstartpack11012_browsersync_1 ... done
Stopping cuefrontstartpack11012_styles_1 ... done
Stopping cuefrontstartpack11012_cleaver_1 ... done
Stopping cuefrontstartpack11012_fridge_1 ... done
Stopping cuefrontstartpack11012_rsync_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 2.1.3. However, the following steps are different:

Step 2
The command you should enter is:

```
docker-compose build browsersync waiter styles styleguide rsync
```

This command names the specific Docker services you want to build. By default **`docker-compose`** builds all the services, including Cook and Cleaver, which you don't need. It therefore executes a good deal faster than **`docker-compose build`**.

Step 3
You only need to copy one file here:

```
cp docker/defaults/waiter-config.yaml docker/waiter-config.yaml
```

Step 4
You only need to edit **`waiter-config.yml`**, as follows:

`publications/name`
Set this to the name of your Escenic publication — for example:

```
publications:
 - name: "my-publication"
```

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

```
hostNames:
 - "www.my-publication.com"
```

cookBaseUrl
Set this to point to the Cook installation you want to use:

```
cookBaseURL: 'http://myserver/mycook/'
```

You should test the URL first by opening a browser and entering the URL there. All you should see at this address is:

```
{}
```

If, however, you add the name of the publication you are going to be working with (plus a final slash) to the URL – **http://*cue-front-host*:8101/my-publication/** – then some real JSON data should be displayed, something like this:

```
{"data":{"resolution":
{"context":"sec","remainingPath":"","publicationName":"my-
publication","sectionUniqueName":"ece_frontpage"},
"context":{"name":"frontpage","section":{"href":"http://myserver/
mycook:8081/my-publication/"},"rootGroup":{"top":[{"display":"",
"content":{"id":"2002","href":"http://myserver/mycook:8081/my-publication/
science/2016-12-14/Mercury-pollution-risk-to-Arctic-gull-2002.html",
...etc...
```

If you don't get results like this when you enter your Cook URL in the browser, then either the URL is wrong or there is something wrong with the Cook installation. You need to get help to find out what's wrong before you continue.

Step 5
You can skip this step completely.

2.2.2 Starting the CUE Front Design Tools

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

```
docker-compose up -d browsersync 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 network "cuefrontstartpack11012_default" with the default driver
Creating cuefrontstartpack11012_rsync_1
Creating cuefrontstartpack11012_fridge_1
Creating cuefrontstartpack11012_cleaver_1
Creating cuefrontstartpack11012_styles_1
Creating cuefrontstartpack11012_browsersync_1
Creating cuefrontstartpack11012_styleguide_1
Creating cuefrontstartpack11012_cook_1
Creating cuefrontstartpack11012_waiter_1
```

If you get problems at this point, the most likely reasons are:

- 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).
- There are syntax errors in one of the configuration files you edited. The YAML format used in the configuration files is very sensitive to white space errors - if you indent parameters too much or too little, it can cause errors.

In both cases, **`docker-compose`** will output error messages explaining the problem. In the case of syntax error messages, note that the line numbers in the error messages do not include comment lines.

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 2.1.6 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 Escenic 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 Engine 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 Engine'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 Engine 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

- 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 Engine, 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 Engine'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 7, 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

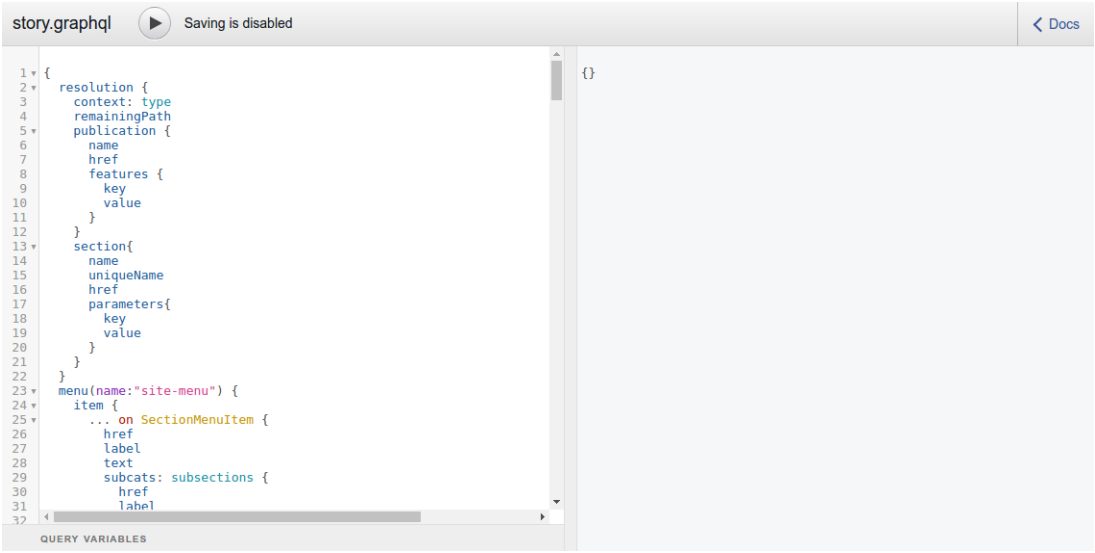
To view JSON data returned by the Cook in the GraphiQL editor, all you need to do is append **`edit`** to the URL you submit to the browser. Instead of

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

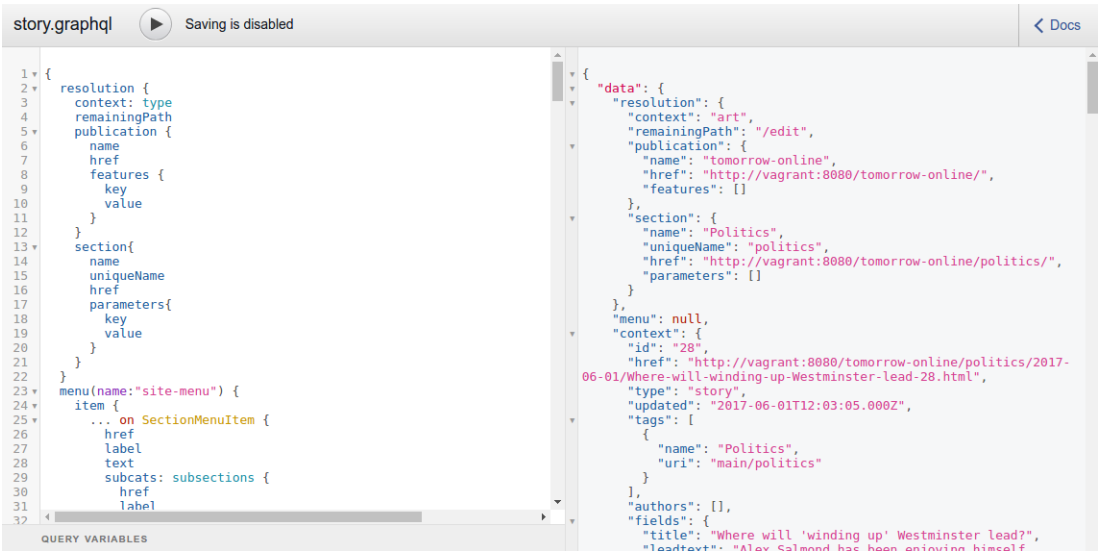
for example, enter:

```
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:

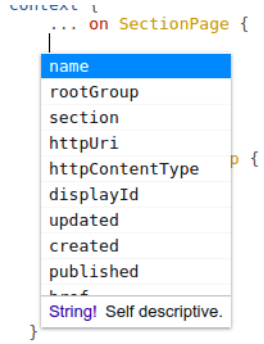


If you click on the ▶ button above the query, then the result of executing the query is displayed on the right side of the screen:

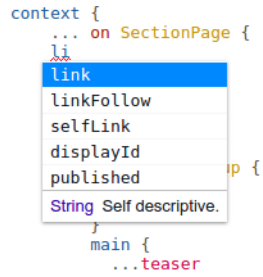


With the query and the results displayed side-by-side like this, it's relatively easy to see the relationship between them. GraphiQL is not just a viewer, it's an editor as well. If you edit the query displayed on the left and click the ▶ button again, then you will see the results of your modification 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.

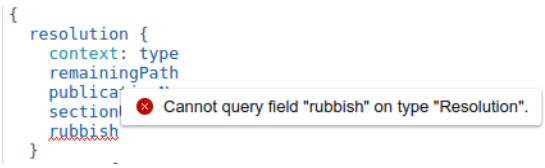
The editor offers you a lot of assistance while you are editing, including code completion. The Cook knows your publication's data structure, so it can tell you what fields are available at any point in the query. Try inserting a line somewhere in the query and pressing **`Ctrl-Space`**: the editor will 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 2.1.3 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 Escenic 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 Engine, 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 Engine'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 Engine 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:

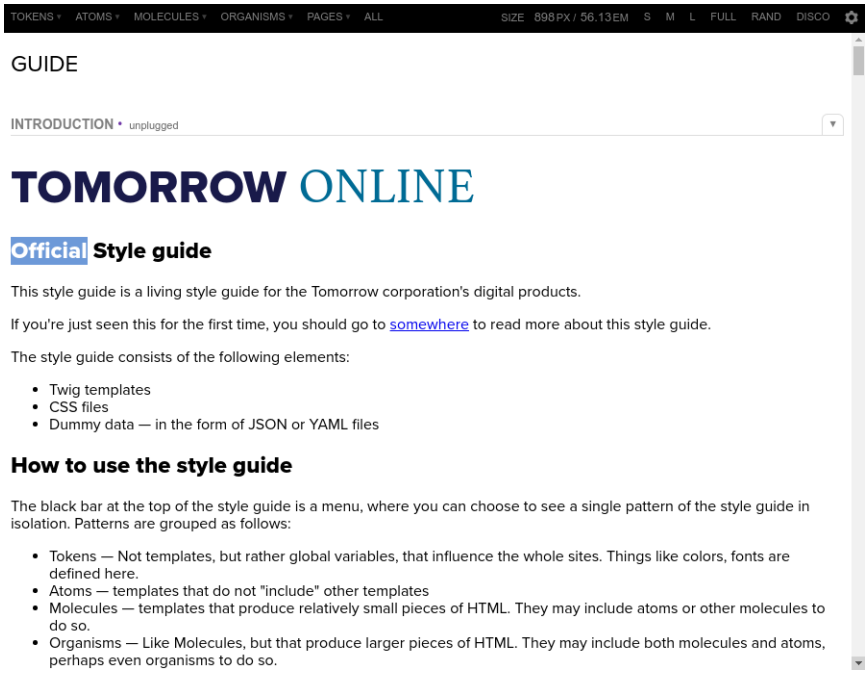
   ```
   docker-compose restart cook
   ```

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

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 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 Engine 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 Engine web services, binary files and so on.

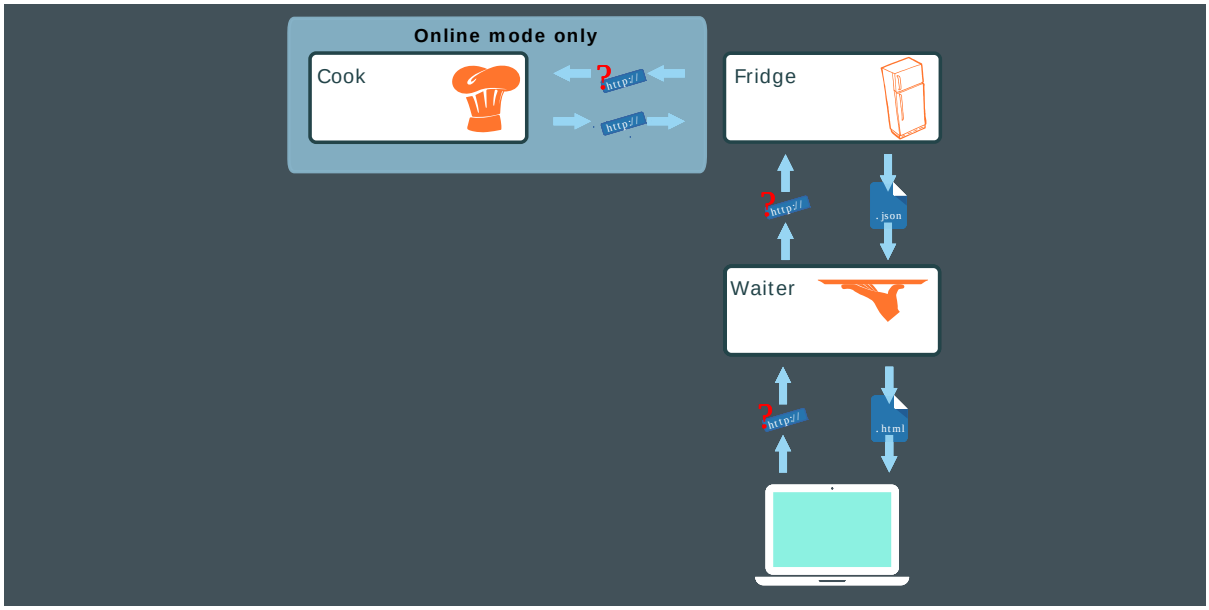
Internally, the Fridge can be configured to o

- **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 Engine.

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
```

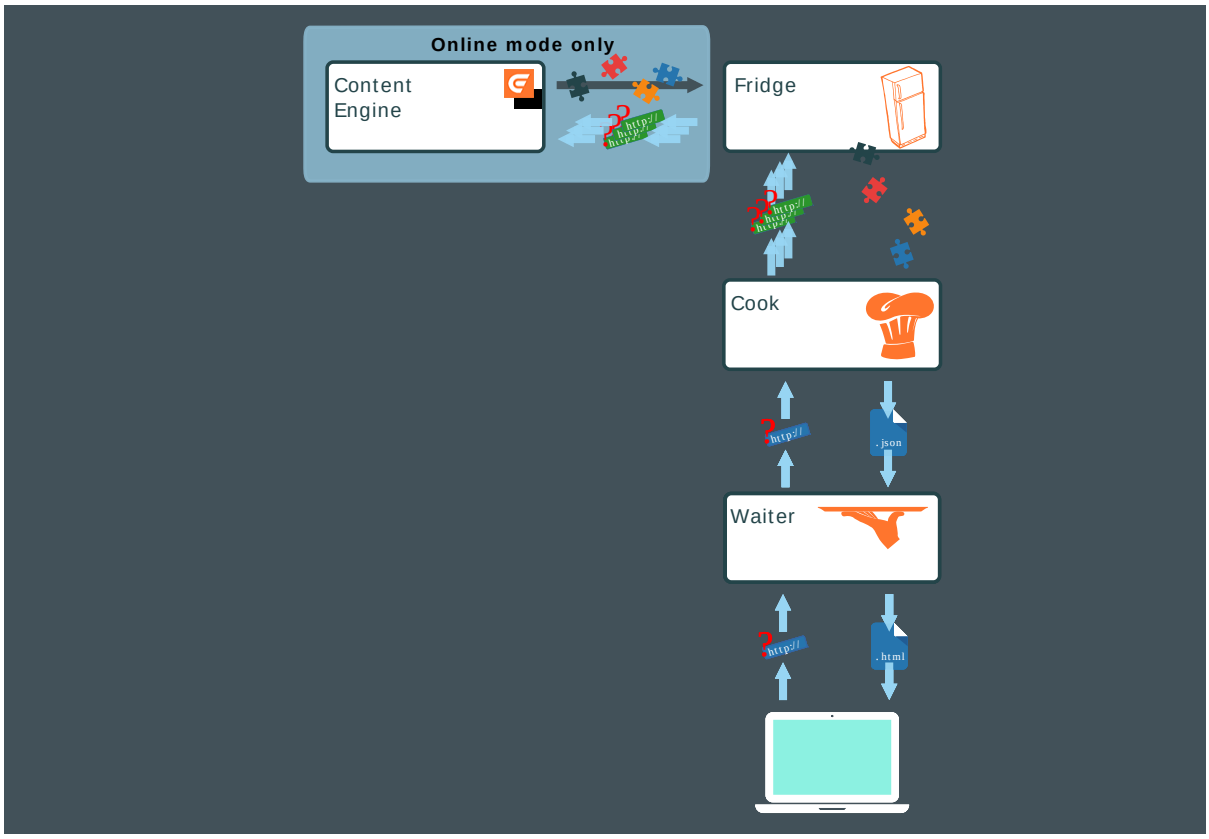
The Waiter will now direct all its requests to the Fridge instead of to the Cook. The Fridge will look for all requested resources in its cache and return them to the Waiter if found. If it cannot find the resources there, then it will forward the request to the Cook. When the Cook responds it will then 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 Engine Proxy



The Cook can be configured to use the Fridge as its data source instead of the Content Engine. 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 Engine. 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 Engine.

4.2.1 Configuring the Fridge as a Content Engine 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 Engine. 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 Engine. When the Content Engine 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 Engine. 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 Engine 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 Engine and the editorial system. Increases in audience can be met by simply duplicating CUE Front components, without any need to scale the Content Engine or its database.
- It improves the reliability of the system: the Content Engine 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 7.4.2.

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 Engine. 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 Engine's graph. Data source queries are in fact executed by Solr and offer a great deal of power and flexibility (although the initial implementation available in the current version of CUE Front is somewhat limited).

Currently, you can make the following kinds of data source queries:

- Get all content items of a specified type
- Get all content items belonging to a specified publication
- Get all content items tagged with a specified tag
- Get all content items that share one or more tags with the current content item

You can also make more complex queries by combining queries of the above type 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 a JSON structure containing both:

- The Solr query generated from the data source query
- The results of the Solr 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. To configure Solr access, open **`cook-config.yaml`** for editing and make sure the **`recipe/search`** section is filled out correctly:

```
recipedata:
  editor:
    allow-save: true
  search:
    uri-prefix: "http://my-escenic.com:8080/webservice/"
    solr:
      host: my-escenic.com
      port: 8983
      core: editorial
```

The entries you need to fill out are:

`uri-prefix`
: The URL of your Content Engine web service, including a final **`/`** character.

`solr/host`
: The host on which your Solr server is running. This may well be the same as your Content Engine host.

`solr/port`
: The port on which your Solr server is listening. Solr listens by default on port 8983.

`solr/core`
: The Solr core the Cook is to use. In a production environment, the Cook 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 Engine 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/edit**. This 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 must contain one top-level **and** or **or** element, which can then contain any number of child search elements. 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 search elements are combined with **AND** operators:

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

If the top level element were an **or** element instead, then the search elements would be combined with **OR** operators:

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

Even if your query only has one search element, the Cook requires you to have an **and** or **or** element 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 search elements of an **and** or **or** element may themselves be **and** or **or** elements. **not** elements 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:

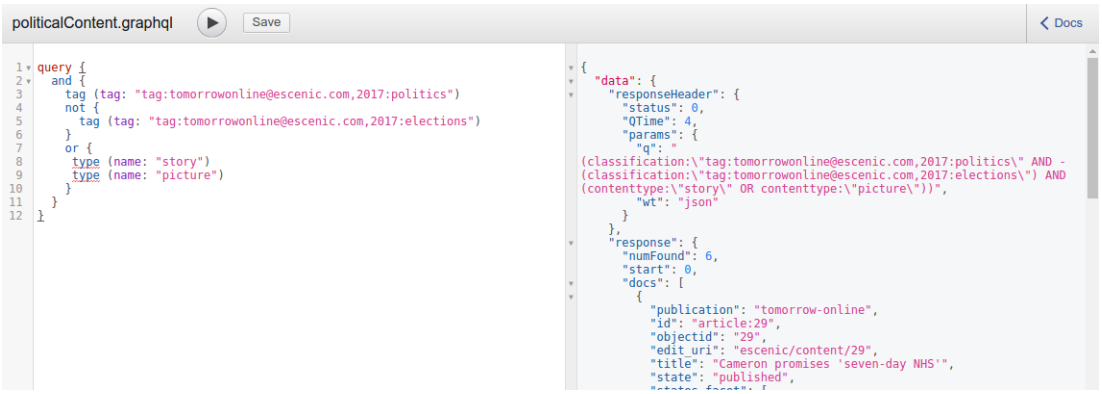
```
(classification:"tag:tomorrowonline@escenic.com,2017:politics"
AND -(classification:"tag:tomorrowonline@escenic.com,2017:elections")
AND (contenttype:"story" OR contenttype:"picture"))
```

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

When you execute a 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 save 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.

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 functions 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 function will return different results according to the context in which it is executed. Currently, the only context-dependent data source function is the **sharedTags** function, which 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.

5.3 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 Engine.

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...
    ]
...
```

6 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 Engine
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.

6.1 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.

7 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.

7.1 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 Escenic APT repository. If you do not have any such credentials, please contact Escenic Support.

3. Configure the Cook as described in section 7.1.1.
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 7.1.1).

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

7.1.1 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 Engine's resolver web service. For example:

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

The Content Engine must be version 6.0 or higher, and its **resolver** web application (supplied with the Content Engine) 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 Engine 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 Engine 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 Engine installation includes the Menu Editor plug-in, then uncomment this line and set it to point to the Content Engine'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 Engine (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.

7.2 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 Escenic APT repository. If you do not have any such credentials, please contact Escenic Support.

3. Configure the Cleaver as described in section 7.2.1.
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 7.2.1).

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

```
cleaver -h
```

7.2.1 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 Engine 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.

7.3 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/1.1.1-1/cue-front-start-pack-1.1.1-1.tar.gz
tar -xzvf  cue-front-start-pack-1.1.1-1.tar.gz
rm cue-front-start-pack-1.1.1-1.tar.gz
ln -s cue-front-start-pack-1.1.1-1 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 7.3.1.
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.

7.3.1 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/name
Set this to the name of your Escenic publication. For example:

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

publications/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"
```

publications/templateDir
Here you must specify the path of the folder containing the Twig templates to be used for rendering this publication. The start pack includes a set of templates located here:

```
    templateDir: "../templates"
```

but if you are working with an existing publication, then you will need to set this property to point to a folder containing the publication's templates.

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.)

browserSyncURL
Leave this set to:

```
browserSyncUrl: http://localhost:3000
```

When this parameter is set, the Waiter is automatically updated with any changes you make to

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

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

7.4 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 Escenic APT repository. If you do not have any such credentials, please contact Escenic Support.

2. Configure the Fridge as described in section 7.4.1.
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 7.4.1).

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

7.4.1 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.

7.4.2 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 Engine 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.