CUE Front

Developer Guide

1.13.9-3

Table of Contents

1 Introduction

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

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

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

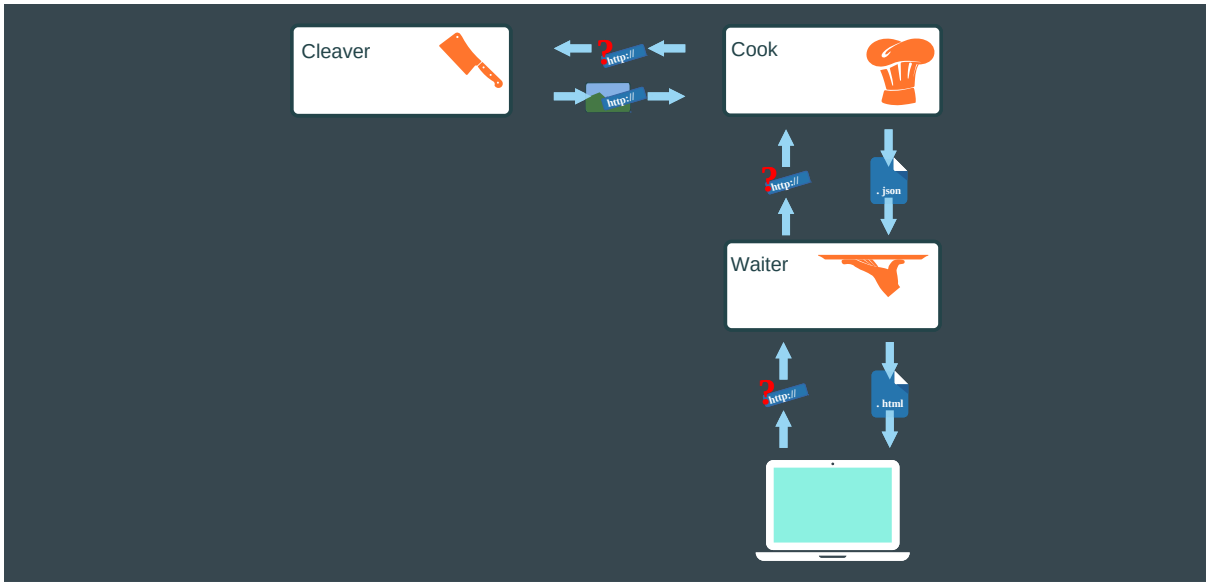
Waiter
A front-end service that responds to requests from browsers and other HTTP clients. The Waiter passes on incoming requests to the Cook and is responsible for rendering the JSON data returned by the Cook as HTML.

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

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

The CUE Front services are designed to be installed in Docker containers, and the CUE Front start pack contains all the configuration files and setup tools required to construct a working installation.

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



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

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

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

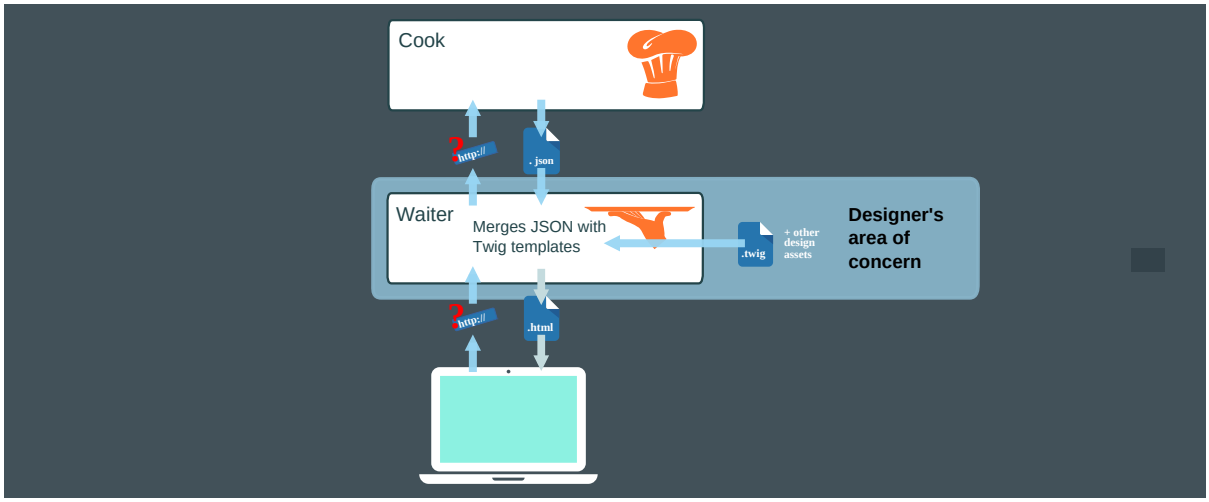
Breaking the presentation layer into separate services encourages separation of concerns: front-end developers/designers can work exclusively with the Waiter (or some other front-end service), and do not need to know anything about Cook or Cleaver. Similarly, back-end developers can concentrate on ensuring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

1.1 CUE Front for Designers

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

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

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



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

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

1.1.1 What is Patternlab?

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

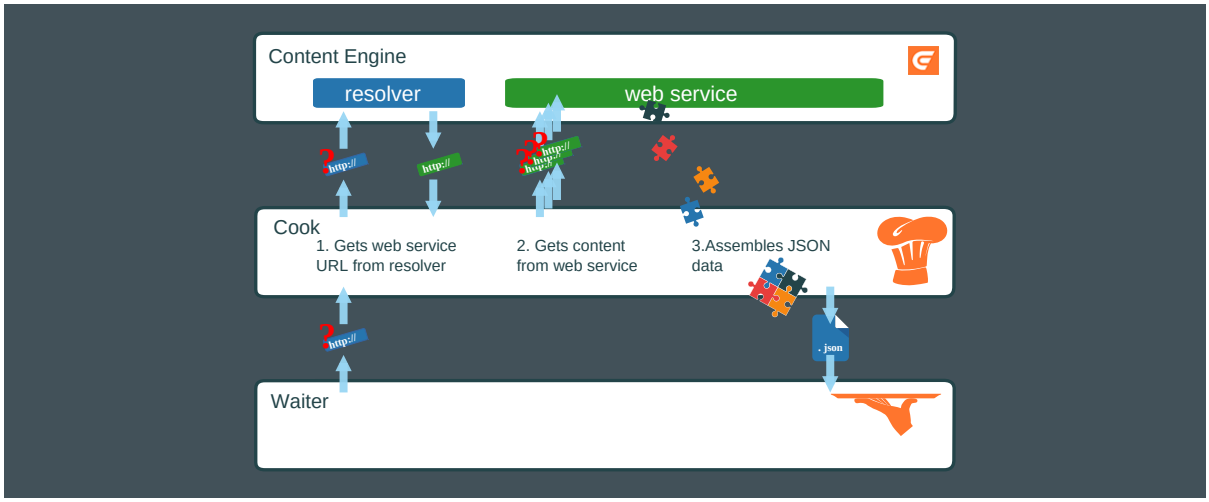
1.1.2 What is Twig?

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

1.2 CUE Front for Developers

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

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



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

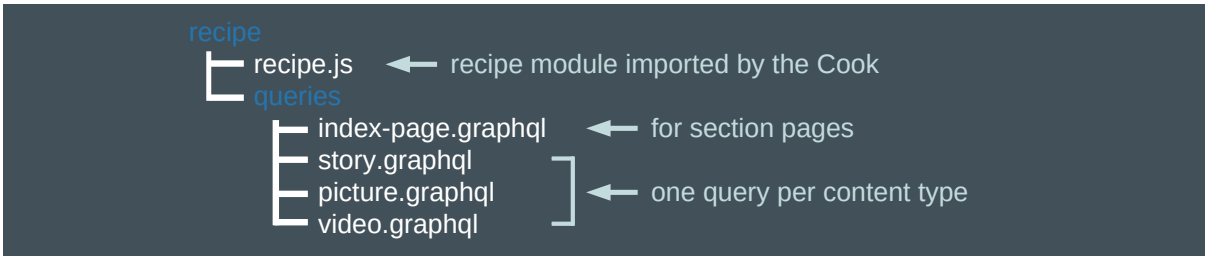
1. Sends the request URL to a Content Store web service called **`resolver`**. The resolver converts this external "pretty" URL to an internal web service URL
2. Sends a request to the returned web service URL. The Content Store web service returns data in the form of Atom XML resources. In order to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

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

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

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

1.2.1 What is a Recipe?



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

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

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

CUE Front includes a script called **`update-schema.sh`** that can automatically generate the GraphQL schema files for any CUE publication (see section 4.1). This means that creating a recipe for a new publication or family of related publications is in many cases just a matter of creating a set of suitable GraphQL queries.

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

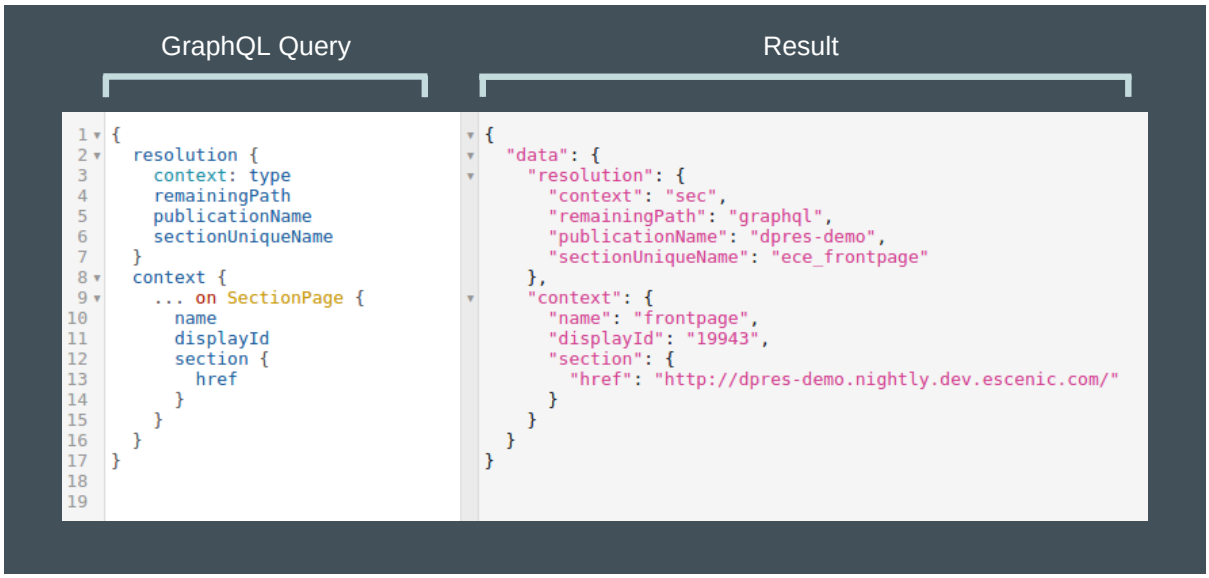
- The Waiter requires the output JSON data to be organized in a different way than the default output (which reflects the Content Store's internal structure). GraphQL allows simple modifications to the output structure, such as omitting elements and renaming, but not complex reorganization.
- The Waiter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

In such cases the default recipe supplied with the CUE Front start pack can be extended by writing an extension of your own. For more about this, see section 8.2.

1.2.2 What is GraphQL?

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

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



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

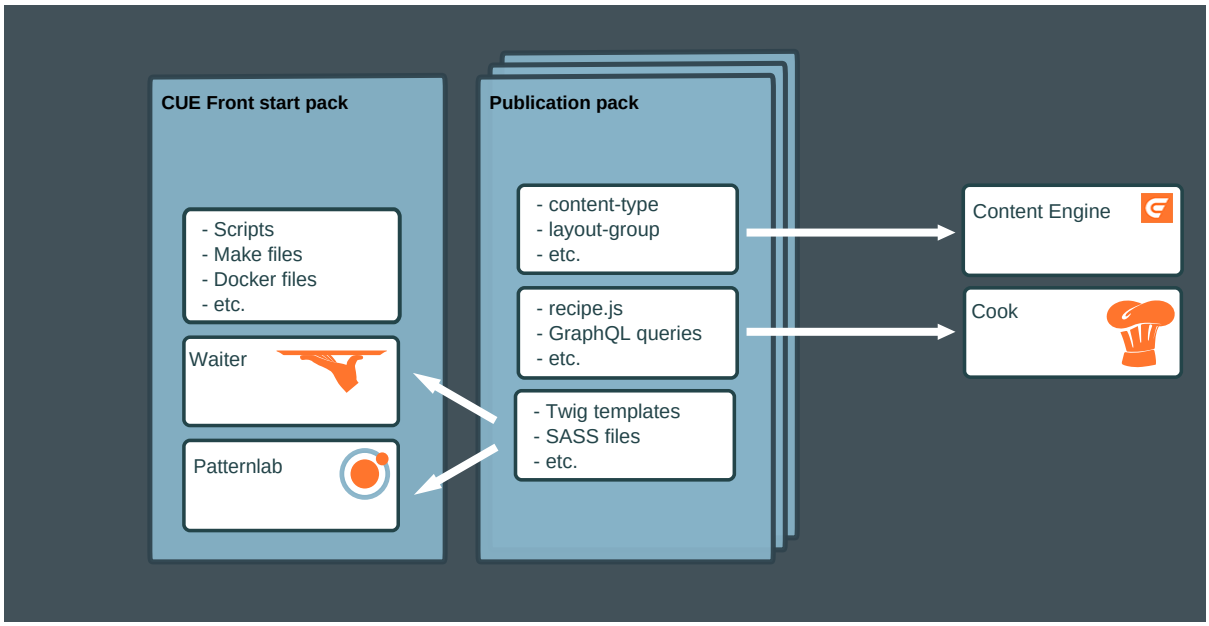
For more about this, see section 4.2.

1.2.3 What Does the Cleaver Do?

The Cleaver is an auxiliary service that handles images for the Cook. Images in CUE publications can include crop information that specifies what aspect ratio the image should have, and what part of the base image should actually be rendered in the specified location. When the Cook receives a request for an image from the Waiter, it forwards the request to the Cleaver, appending the required crop information as URL parameters. The Cleaver then retrieves the base image from the Content Store, carries out any required crop operations and returns the cropped image to the Cook. The Cook then serves this image to the Waiter. The Cleaver maintains a cache for the images it downloads from the Content Store 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 downloadable package that contains the infrastructure needed to get a CUE Front installation up and running - mostly consisting of Shell scripts and Make files, plus the Docker files needed to define the Docker containers in which the CUE Front services run. The start pack also contains:

- CUE Front's default Waiter (a PHP front end for serving publications)
- Patternlab, the style guide application supplied with CUE Front
- A setup tool for installing and configuring the CUE Front services

`cue-front-start-pack` is made available as a tarball that you can download from the Stibo DX Maven repository and modify to suit your requirements. The Maven repository also contains publication packs – tarballs containing sample publications that you can use together with the CUE Front start pack for various purposes. A publication pack contains:

- Publication resources (**`content-type`**, **`layout-group`** and **`layout`** files) for uploading to the Content Store, plus possibly one or more files containing demo content.
- A corresponding set of Twig templates, SASS files and other design assets for rendering content retrieved from the defined publication
- A recipe and set of GraphQL queries for retrieving page content from the Content Store

The currently available publication packs are:

`tomorrow-online`
: This pack contains the Content Store demo publication, **Tomorrow Online**. This publication is used as the basis for all examples and discussion in this manual.

`starter-publication`
: This pack contains a minimal publication that you can use as the starting point for your own projects.

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 actually an nginx web server instance used to maintain a cache, 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 6.

2.1 Quick Start for Test/Development

The general procedure is:

1. Install Docker on your machine – see section 2.1.1
2. Optionally download and unpack a sample publication, or check out an existing publication from your own repositories – see section 2.1.2
3. Download the CUE Front start pack and unpack it – see section 2.1.3
4. Install the CUE Front components in Docker containers – see section 2.1.4
5. Run the Docker containers – see section 2.1.5

2.1.1 Installing Docker

The installation method for Docker is platform-dependent.

2.1.1.1 Installing Docker on Ubuntu

These instructions are based on the use of Ubuntu 18.04 LTS.

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

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

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

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

You can now continue by following the instructions in section 2.1.3.

2.1.1.2 Installing Docker on Windows

The recommended Docker installation for Windows is Docker for Windows. This version of Docker, however, can only be installed on Windows 10 Pro or Windows 10 Enterprise Edition. If you have an earlier version of Windows, you can install Docker Toolbox instead. Docker Toolbox can be installed on any 64-bit version of Windows 7, 8 or 10.

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 for Windows uses Microsoft's Hyper-V to host the Linux virtual machine, while Docker Toolbox uses VirtualBox.

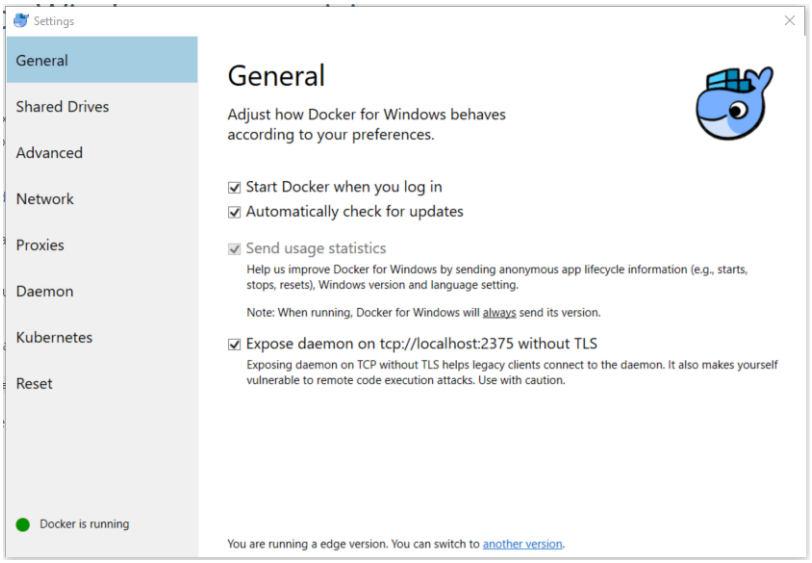
VirtualBox and Hyper-V cannot co-exist on the same machine, so if you need VirtualBox for other purposes, then you should use Docker Toolbox even on Windows 10.

2.1.1.2.1 Docker for Windows (Windows 10 Pro or better)

The following procedure describes how to install Docker for Windows and set up your system for working with CUE Front. The procedure also involves installing **Windows Subsystem for Linux**. This gives you access to an Ubuntu shell environment inside Windows that you can use for installing, configuring and controlling CUE Front.

1. Enable Hyper-V as described here.
2. Install Windows System for Linux as described here. Choose the Ubuntu Linux distribution.
3. Install Docker for Windows.

4. After installing, select **Expose daemon on tcp://localhost:2375 without TLS** in the Docker general settings:



   This is necessary to ensure that Docker is accessible from **Windows Subsystem for Linux**.

5. Open a Powershell window.
6. Continue as described in section 2.1.3.

2.1.1.2.2 Docker Toolbox (Earlier Versions)

The following procedure describes how to install 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 (see next step).
4. Continue as described in section 2.1.3. Everywhere the instructions tell you to use a URL containing **`localhost`**, specify the IP address of your Docker virtual machine instead of **`localhost`**.

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 2.1.3.

2.1.2 Getting a Publication Pack

If you are starting completely from scratch, you are going to need a publication to work with. Stibo DX provides two different starting points that you can download:

- **`tomorrow-online`**, which contains the CUE Content Store demo publication, "Tomorrow Online". We recommend that you download this publication if you are trying out CUE Front for the first time, as it includes some content, with useful examples of how to do various things.
- **`starter-publication`**, a simple publication that you can use as a starting point for your own publications.

If your organization is already using CUE Front, then there will probably be an existing publication that you can work with, in which case you may like to skip this step, and instead check out that publication from your repositories.

The following commands will unpack the **Tomorrow Online** publication in your home folder:

```
cd
curl -O https://user:password@maven.escenic.com/com/escenic/cook/tomorrow-
online/1.13.9-3/tomorrow-online-1.13.9-3.tar.gz
tar -xzvf  tomorrow-online-1.13.9-3.tar.gz
rm tomorrow-online-1.13.9-3.tar.gz
ln -s tomorrow-online-1.13.9-3 tomorrow-online
cd tomorrow-online
```

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

If you want to install the starter publication rather than **Tomorrow Online**, replace every occurrence of **`tomorrow-online`** in the above commands with **`starter-publication`**.

> If you are installing on a Mac, make sure you unpack the publication 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).

2.1.2.1 Making a git Repository

If you intend to make changes to the downloaded publication (even if you're only doing so as a learning exercise), then we recommend that you commit the publication 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 4.6.

2.1.2.2 Uploading the Publication

If the downloaded publication is not already installed at your site, and you don't have access to a copy running anywhere else, then you will need to upload it to your Content Store.

Create the publication by entering the following command:

```
make dist -C publication
```

You will then find two versions of the publication in the **tomorrow-online/publication/dist** folder:

tomorrow-online.zip
tomorrow-online-with-content.zip (we recommend that you use this one, which includes some content)

(You will also see a file called **tomorrow-sport-with-content.zip**. Please ignore this file for now.)

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

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

2.1.3 Getting the CUE Front start pack

Download and unpack the CUE Front start pack. The following commands will unpack it in your home folder:

```
cd
curl -O https://user:password@maven.escenic.com/com/escenic/cook/cue-front-start-
pack/1.13.9-3/cue-front-start-pack-1.13.9-3.tar.gz
tar -xzvf  cue-front-start-pack-1.13.9-3.tar.gz
rm cue-front-start-pack-1.13.9-3.tar.gz
ln -s cue-front-start-pack-1.13.9-3 cue-front
cd cue-front
```

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

2.1.4 Installing the CUE Front Components

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

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

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

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

2. Build the setup container by entering:

```
docker-compose build
```

3. Initialize the setup tool by entering:

```
docker-compose run setup login username
```

where *username* is the name of your account for downloading software from Stibo DX's repositories. You will be prompted to enter a password.

4. Open *path*`/cue-front/setup/.env` in a text editor, and add the following line:

```
location=publication-path
```

where *publication-path* is the path of the folder containing the publication you downloaded. For example:

```
location=~/tomorrow-online-1.13.9-3
```

> On Windows, *publication-path* must be specified using Windows syntax (that is, using `\` rather than `/`).

5. Optionally add the following line to *path*`/cue-front/setup/.env`:

```
NODE_VERSION=version
```

where *version* is the version of `node.js` you want CUE Front to use, specified in the format *major*`.x` (`12.x`, for example, which will give you the latest update of version 12). The `node.js` version you specify will be used in all the Docker containers created by CUE Front. If you do not specify a version, then CUE Front will use its current default version.

6. To run the setup tool, enter:

```
docker-compose run setup add configuration-set
```

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

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

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

Which Host OS is in use?
Press `Space` to select your operating system and then press `Enter`.

Type of setup?
Press `Space` to select the required setup type and then press `Enter`. If you select **development**, then changes made to the recipe, schema, GraphQL queries, and data sources while the Cook is running take immediate effect; if you select **production**, then

they won't take effect unless the Cook is restarted. For most purposes you should select **development**.

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

[fridge] Which port (nnnn) or port range (nnnn-nnnn) should be exposed on the host?

The port number the Fridge will listen to.If you enter a range of port numbers then several Fridges will be started, one listening on each port number.

[cook] Which port (nnnn) or port range (nnnn-nnnn) should be exposed on the host?

The port number(s) the Cook will listen to.If you enter a range of port numbers then several Cooks will be started, one listening on each port number.

[cook] Escenic Content Store hostname

The host name or IP address to which the Cook will send web service requests. Note that even if you are installing CUE Front on the same machine as Content Store, you must not specify **`localhost`** (or **`127.0.0.1`**) here: specify your machine's public IP address.

[cook] Access credentials username

The CUE username the Cook is to use when accessing the Content Store. For demo/test purposes it is most convenient to use the publication's admin username, which is always *publication-name*`_admin`. For example, If you called the publication **`tomorrow-online`** when you uploaded it to the Content Store (see section 2.1.2.2), then the admin username is **`tomorrow-online_admin`**.

> Using the admin user for CUE Front is **not** a good idea for production systems. In a production system you should create a special user for CUE Front that has read-only access to all your publication's sections and content types. If you want to be

able to support cross-publishing, then this user must also have read access to all the publications from which content might be selected.

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

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

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

[waiter] Which port (nnnn) or port range (nnnn-nnnn) should be exposed on the host?
The port number(s) the Waiter will listen to. If you enter a range of port numbers then several Waiters will be started, one listening on each port number.

[waiter] Publication name
The name of the CUE publication you want to publish using CUE Front (Tomorrow Online in this case). This is the name you specified when uploading Tomorrow Online to the Content Store (see section 2.1.2.2).

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

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

[matomo-db] Database host root password
The root password for the Matomo database's host machine. Make a note of this value as you will need to use it later.

[matomo-db] Database user name
The user name for the Matomo database. Make a note of this value as you will need to use it later.

[matomo-db] Database user password
The user password for the Matomo database. Make a note of this value as you will need to use it later.

7. Once you have answered all the prompts, the setup tool verifies your responses, generates a set of configuration files, checks for errors, and if all is OK downloads the requested components and builds Docker containers for them. When it is finished, you will see that it has created a folder for the configuration inside the **`setup`** folder (**`setup/myconfig`**, for example). Go to this configuration folder:

```
cd ../myconfig
```

8. If your Content Store is running in a virtual machine on your PC and is accessed via a host name specified in your PC's **`hosts`** file, then you need to open the file **`docker-compose.yml`** in the **`setup/myconfig`** folder and add an **`extra_hosts`** property to both the Cook and Cleaver sections of the file. The **`extra_hosts`** properties let you provide the Cook and Cleaver containers with a host name mapping for the Content Store, since they do not have access to your PC's

hosts file. So if you have the following entry in your **hosts** file to set up a host name for your Content Store VM:

```
192.168.56.101 engine.local
```

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

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

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

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

```
docker-compose build
```

CUE Front is now installed and **almost** ready to run based on the configuration you created. If you want to make configuration changes later, you can edit the configuration using the **setup edit** command, or create a different configuration set with a different name. The setup tool has an advanced mode which allows you to configure the components in more detail. For more information about all these options, see chapter 13.

At present, the **setup** tool is unable to automatically configure the Matomo installation used to provide web analytics functionality for CUE Front publications. This means that after starting CUE Front for the first time you should configure the Matomo installation as described in section 2.1.7. Until you have done this, Tomorrow Online will not contain any "most read" links.

2.1.5 Starting CUE Front

To start CUE Front, make sure you are in the **myconfig** folder, and 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 "cuefrontstartpack12019_default" with the default driver
Creating cuefrontstartpack12019_fridge_1 ...
Creating cuefrontstartpack12019_cleaver_1 ...
Creating cuefrontstartpack12019_rsync_1 ...
Creating cuefrontstartpack12019_cleaver_1
Creating cuefrontstartpack12019_fridge_1
Creating cuefrontstartpack12019_fridge_1 ... done
Creating cuefrontstartpack12019_cook_1 ...
Creating cuefrontstartpack12019_rsync_1 ... done
Creating cuefrontstartpack12019_browsersync_1 ...
Creating cuefrontstartpack12019_styles_1 ...
Creating cuefrontstartpack12019_styles_1
Creating cuefrontstartpack12019_styles_1 ... done
Creating cuefrontstartpack12019_styleguide_1 ...
Creating cuefrontstartpack12019_waiter_1 ...
Creating cuefrontstartpack12019_styleguide_1
```

```
Creating cuefrontstartpack12019_waiter_1 ... done
Creating cuefrontstartpack12019_loadBalancer_1 ...
Creating cuefrontstartpack12019_styleguide_1 ... done
```

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

Assuming all went well, start a browser — you should be able to find the services listed below:

The demo publication

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

The Cook

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

```
{
  error: "Failed to resolve context: /"
}
```

If, however, you add the name of the demo publication (plus a final slash) to the URL – **`http://localhost: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://localhost: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 4.2.

The Cleaver

At **`http://localhost:8102/`** you should find the Cleaver. All you will see is:

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

Patternlab

At **`http://localhost: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 4.3.

Matomo

At **`http://localhost:8106/`** you should find the Matomo administration interface. You can use this to complete the configuration of the Matomo web analytics platform. For more about this, see section 2.1.7.

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

You can then restart the CUE Front by entering:

```
docker-compose start
```

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

To stop CUE Front and remove the containers, enter:

```
docker-compose down
```

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

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

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

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

To restart the Waiter, for example, enter:

```
docker-compose restart waiter
```

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

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

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

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

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

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

> This is not the recommended way of viewing log messages – see section 11.2 for a better way.

2.1.6.1 Fixing Docker Problems

Removing unwanted containers

You may find from time to time that you get problems when starting up. This is often due to the existence of rogue Docker containers that should have been stopped but are still running. The following error message, for example:

```
port is already allocated
```

probably means that a Docker container that should have been closed is still running, and occupying a port you want to use.

The following commands are useful for solving these kinds of issues:

```
$ docker ps -a
```

This lists any Docker containers that are currently running. If you aren't currently running CUE Front, and don't have any other Docker-based software running on your computer, it should not return anything. If it does return a list of container IDs, you probably want to terminate them, which you can do as follows:

```
$ docker rm -fv container-id
```

If you want to delete **all** the listed containers, you can do it like this:

```
$ docker rm -fv $(docker ps -aq)
```

> **Don't do this unless you are sure you want to delete all the containers on your machine.** There might be other software besides CUE Front running in Docker containers, and this command will remove them too.

Removing unwanted images

You may also find after a while that Docker is filling up your disk due to unused Docker images.

To list all Docker images, enter:

```
$ docker images
```

To delete an unwanted image, enter:

```
$ docker rmi image-id
```

To delete **all** your Docker images, enter:

```
$ docker rmi $(docker images -q)
```

> **Don't do this unless you are sure you want to delete all the Docker images on your machine.** There might be other software besides CUE Front running in Docker containers, and this command will remove them too. Even if CUE Front is the only Docker-based software running

on your machine, you may not want to delete **all** images. It means that restarting CUE Front will take a long time, as all the containers have to be recreated and provisioned with software.

2.1.7 Analytics Configuration

CUE Front includes an analytics extension based on Matomo, a popular open source web analytics platform. Matomo runs in its own container (the **`analytics`** container) and stores its analytics data in a MariaDB database that runs in another container called **`matomo-db`**.

For technical reasons, the setup tool is not currently able to automatically configure the Matomo installation running in the **`analytics`** container. Therefore, you must do it yourself. Matomo incorporates an easy-to-use web administration interface, so this is quite easy to do. Once you have started CUE Front as described in section 2.1.5, you can access the Matomo administration interface by starting a browser and going to **`http://localhost:8106/`**.

You will find general instructions for how to configure Matomo here. Here is some additional information you need to ensure that the Matomo installation will work correctly together with the other CUE Front components:

- On the **Database Setup** page:
  - Enter **`matomo-db`** in the **Database Server** field.
  - Enter the same value in the **Login** field as you entered at the **[matomo-db] Database user name** prompt in **`setup`** (this is **`root`** if you accepted the default).
  - Enter the same value in the **Password** field as you entered at the **[matomo-db] Database password** prompt in **`setup`**.
  - Enter **`matomo`** in the **Database Name** field.
- On the **Super User** page you can enter any values you choose.
- On the **Setup a Website** page enter the details of one of your web sites. For the Tomorrow Online demo website, you should enter **`http://tomorrow-online:8100/`**.
- For the Tomorrow Online website you can ignore the Javascript tracking code supplied at the end of the configuration process, since it is already included in the Tomorrow Online templates. In other cases, you need to make sure the code is included in your templates.

After completing the basic configuration, you must:

1. Find the administration interface's **Manage Users** page
2. Change the access of the **`anonymous`** user to **`View`**.

Once you have made these changes you should be able to see "most read" links in Tomorrow Online.

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 2.1.3
3. Install the CUE Front components in Docker containers as described below in section 2.2.1.

4. Run the Docker containers as described below in section 2.2.2

2.2.1 Installing for Designers

The basic installation procedure for designers is the same as the general Docker installation procedure described in section 2.1.4. Step 5, however is different. When you get to this step you should answer the setup tool's prompts as follows:

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

Configuration
Select **Quick**.

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

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

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

2.2.2 Starting the CUE Front Design Tools

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

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

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

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

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

Assuming all went well, start a browser — you should be able to find the services listed below:

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

Patternlab
At **`http://localhost: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 4.3.

See section 2.1.6 for information on how to stop the CUE Front services you have started.

3 Upgrading

New versions of the CUE Front start pack are released at regular intervals, usually together with new versions of the CUE Content Store and CUE editor. Customers are recommended to follow this upgrade cycle in order to ensure that they are running a recent version of the CUE Front infrastructure.

You should be aware that CUE Front has three groups of components, each of which can be upgraded independently of the other two. These groups are:

The start pack
This is the CUE Front infrastructure:

- Docker files that define the containers in which the various components run, download and install the Cook, Cleaver and Fridge.
- The default Waiter code
- Patternlab, the setup tool and other CUE Front utilities

This component is upgraded by downloading a tarball from the Stibo DX Maven repo and unpacking it. For further information, see section 3.1.

Cook and Cleaver
Whenever a new version of the CUE Front start pack is released, new versions of the Cook and Cleaver are also released and made available on Stibo DX's Debian repo. An upgraded start pack will not automatically download and install new versions of these components however. The start pack's Docker files are configured to look for a version number in the appropriate publication definition folder, and download the version specified there.

To upgrade the Cook and Cleaver versions used at your installation, therefore, you must edit the `cue.yaml` file in the root folder of your publication definition. For more information about how to do this, see section 3.1.1.

Recipe extensions
Upgrading the start pack does not affect your publication definition: your recipe, GraphQL queries, templates and so on are not touched by the upgrade process in any way.

The default recipe extensions made by Stibo DX are published on the NPM server `npm.escenic.com`. Each extension published on `npm.escenic.com` is separately maintained and has its own version number. As corrections and improvements are made to an extension, new versions may be published. This process is completely independent of the CUE Front release cycle. You will only ever get a new version of a recipe extension if you explicitly request it, and you can upgrade a recipe extension at any time.

Nevertheless, you may choose to upgrade recipe extensions in combination with a CUE Front upgrade. For more information about how to do this, see section 8.4.

3.1 Upgrade Procedure

> If you are upgrading from CUE Front version 1.4, then you cannot use this procedure to upgrade to version 1.13.9-3. You will need to follow the procedure described here instead (and replacing the version number `1.5.0-4` in the instructions with the number `1.13.9-3`).

The basic upgrade procedure is:

1. Download the new start pack in a new location, as described in section 2.1.3.
2. Copy your config folders from the old installation to the new installation. For example:

   ```
   cp old-installation/myconfig new-installation
   ```

3. Copy any other files that you have modified from the old installation to the new installation (for example, the **nginx.conf** file in *old-installation*/**service/waiter/docker** and the **fridge.conf** file in *old-installation*/**service/fridge**, if you have modified either of them).
4. Delete your old installation.
5. Rename your new installation to the same name as your old installation.
6. Upgrade Cook and Cleaver, if required. See section 3.1.1 for details.
7. Restart your CUE Front containers as described in section 2.1.6.

3.1.1 Upgrading Cook and Cleaver

> If you are upgrading from CUE Front version 1.5 to version 1.13.9-3, then your publication root folder will not contain a **cue.yaml** file as described below – you will need to create it.

To upgrade Cook and Cleaver:

1. Open *publication-folder*/**cue.yaml** in an editor:

   ```
   versions:
    cue-front: "old-version"
   ```

2. Replace the old version number with the number of the version you want to upgrade to:

   ```
   versions:
    cue-front: "1.13.9-3"
   ```

3. Save the file.
4. Go to your CUE Front installation's **setup** folder and regenerate your configs:

   ```
   cd cue-front-path/setup
   docker-compose -f setup.yml run setup generate configuration-set
   ```

4 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 4.1 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.

4.1 Updating a GraphQL Schema

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

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

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

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

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

The script's parameters are:

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

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

Important notes

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

  ```
  docker-compose restart cook
  ```

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

4.2 Working with GraphQL

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

If you have installed the CUE Front components as described in section 2.1, 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://localhost: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://localhost: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 4.2.1) interface.

4.2.1 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

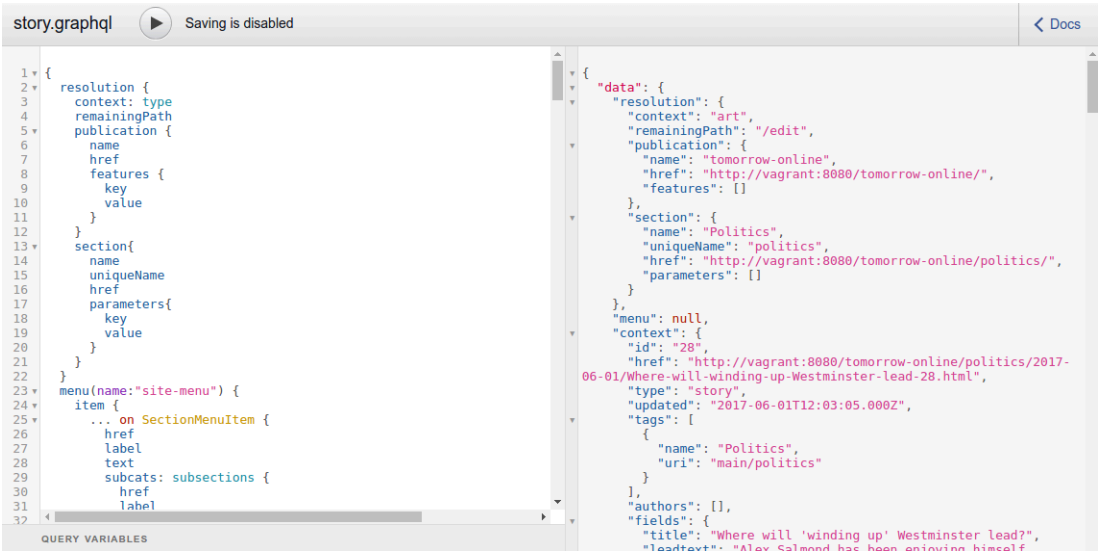
```
http://localhost:8101/tomorrow-online/
```

for example, enter:

```
http://localhost:8101/tomorrow-online/edit
```

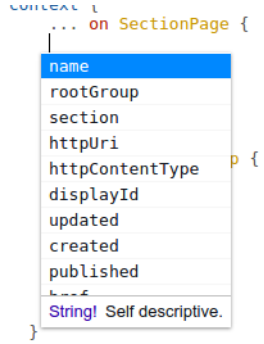
Now, instead of simply displaying the JSON data normally returned by the Cook, the browser displays a vertically split screen, on the left side of which is the GraphQL query that the Cook would use to retrieve the page data:

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:

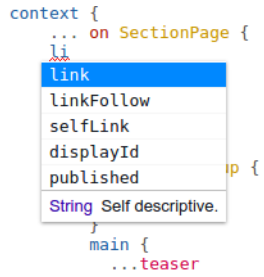


With the query and the results displayed side-by-side like this, it's relatively easy to see the relationship between them. GraphiQL is not just a viewer, it's an editor as well. If you edit the query displayed on the left and click the ▶ button again, then you will see the results of your modification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

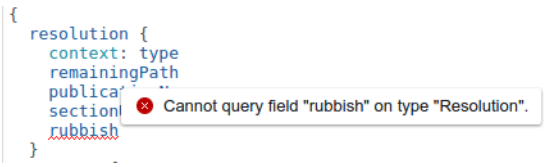
The editor offers you a lot of assistance while you are editing, including code completion. The Cook knows your publication's data structure, so it can tell you what fields are available at any point in the query. Try inserting a line somewhere in the query and pressing **`Ctrl-Space`**: the editor will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

4.2.1.1 Saving Your GraphQL changes

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

Clicking on this button will save any changes you have made. The changes are saved directly into the **`cue-front/recipe/queries`** folder used by the Cook, so the saved changes will take immediate effect. If, for example you remove a field from the JSON data output by the query, then that content

will disappear from any pages in which it is used on the web site. Conversely, any fields you add to the output will immediately be available for use by the front end.

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

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

   ```
   editor: enabled
   ```

   with:

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

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

4.2.2 Understanding CUE Front GraphQL Queries

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

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

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

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

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

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

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

nop
Not used.

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

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

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

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

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

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

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

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

```
... on SectionPage
```

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

Another useful GraphQL construct is:

...name

for example:

```
top {
```

```
...teaser
```

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

```
fragment teaser on AtomLink
```

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

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

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

CUE Front's GraphQL snippet feature enables more extensive reuse of GraphQL code. A snippet is stored in its own file and can be included in multiple queries. For more information, see section 4.2.4.

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

4.2.3 Mapping URLs To GraphQL Queries

The GraphQL query used to retrieve content for any given URL is determined by a set of configurable mapping rules. A default mapping configuration is included in the CUE Front start pack that allows you to 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.
- Any URL that starts with **`.esi/`** will be directed to queries in the **`recipe/queries/esi`** folder. The query called **`recipe/queries/esi/header.graphql`**, for example, will be used to respond to requests for **`.esi/header`**. This convention is intended to support the use of Edge Side Includes. For further information, see chapter 9.

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://localhost: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.

Changing the default URL-query mapping

All the naming conventions described above are just the default mapping rules included in the start pack. If they do not meet your needs, you can easily extend them, or completely replace them with mappings of your own. For information on how to do this, seesection 8.1.3.

4.2.4 GraphQL Snippets

Fragments are useful way of reusing GraphQL code. However, they only allow you to reuse code within a single GraphQL query. Snippets, on the other hand allow you to share code between GraphQL queries. Snippets are segments of GraphQL code that are stored in special snippet files and included when queries are loaded.

To create a snippet, you simply surround the code you want to re-use with a pair of special comments like this:

```
#define snippet-name
your-reusable-code
#enddefine
```

When you save a GraphQL query that contains such a snippet definition, the snippet definition is stripped out of the Graphql query and saved as a **.snippet** file in the **/recipe/queries** folder. The snippet definition in the GraphQL query is replaced by an **#include** comment:

```
#include snippet-name
```

So if you save a GraphQL query that looks like this:

```
query {
  resolution {
    ....
    #define section-info
    section {
      name
      uniqueName
      href
      parameters {
        key
        value
      }
    }
    #enddefine
    ....
```

```
    }
    ......
  }
```

what is actually saved in the query file is:

```
query {
  resolution {
    ....
    #include section-info
    ....
  }
  ......
}
```

In addition, a new file, **`/recipe/queries/section-info.snippet`**, is created:

```
section {
  name
  uniqueName
  href
  parameters {
    key
    value
  }
}
```

You can now include this snippet in other queries as follows:

```
 #include section-info
```

Note that although only **`#include`** commands are saved in query files, when you open a query in the editor, you will see complete, expanded snippets like this:

```
query {
  resolution {
    ....
    #shared section-info
    section {
      name
      uniqueName
      href
      parameters {
        key
        value
      }
    }
    #endshared
    ....
  }
  ......
}
```

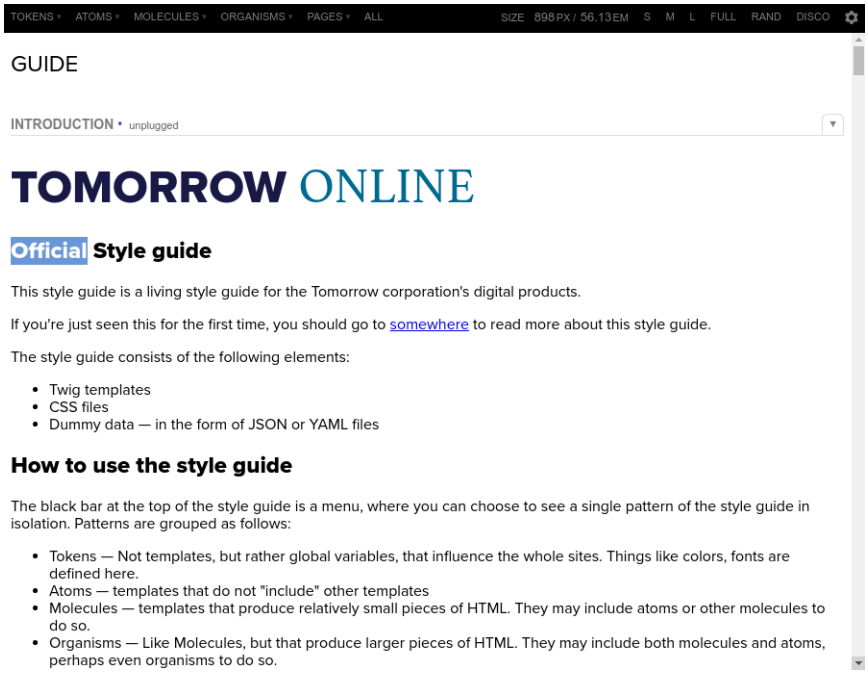
You can edit the contents of the **`#shared`** segment: when you save, the relevant snippet file is updated, effectively updating all the queries that include it.

> The **`#define`**, **`#enddefine`**, **`#shared`**, **`#endshared`** and **`#include`** keywords must be written exactly as specified with no space after the **`#`** character otherwise they will be treated as ordinary comments and ignored.

4.3 Working With Twig and Patternlab

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

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



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

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

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

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

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

Patternlab requires the templates that make up a pattern library to be stored in a known location, in accordance with specific naming conventions. A publication's patterns are stored in its **`templates/_patterns`** folder. In the **`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 4.3.1. For more detailed information about Patternlab, see the Patternlab documentation.

4.3.1 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 **`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)

4.3.2 Standard Template Structure

Tomorrow Online and the default starter publication provided by Stibo DX adhere to a standard template structure that makes the templates relatively easy to extend and modify. This structure makes use of **layouts**, special top-level templates that provide the overall framework of different page types. Understanding and adhering to this structure will make it much easier to work with your publications.

4.3.2.1 The pages-index Template

The first template loaded by the Waiter is **`pages-index`**: that is, a template called **`index.twig`**, located somewhere in the **`_patterns/10-pages`** folder. In Tomorrow Online, this is **`templates/_patterns/10-pages/other/index.twig`**. This initial template should not contain any markup: all it should do is examine the data supplied by the cook, and based on that data, select an appropriate page template to include. Here is the content of Tomorrow Online's **`index.twig`**:

```
{%- if data.tag -%}
  {%- include [ "pages-topic-page" ] -%}
{%- elseif data.resolution.context == "sec" -%}
  {%- include [ "pages-" ~ data.resolution.section.uniqueName ~ "-page" ,  "pages-" ~
 data.resolution.section.name ,  "pages-generic-section-page" ] -%}
{%- elseif data.resolution.context == "art" -%}
  {%- include [ "pages-" ~ data.context.type | lower, "pages-generic-article-page" ] -
%}
{%- endif -%}
```

If the data supplied by the Cook contains a **`tag`** field, then the template **`pages-topic-page`** is included.

If the **`resolution.context`** field has the value **`"sec"`**, then a section template is included. First a section-specific template is searched for (that is, a template with a name based on the current section's unique name or name). If no section-specific template is found, then the **`pages-generic-section-page`** template is selected.

If the **`resolution.context`** field has the value **`"art"`**, then an article template is included. First a content type-specific template is searched for (that is, a template with a name based on the current content item's unique name or name). If no content type-specific template is found, then the **`pages-generic-article-page`** template is selected. Note that Tomorrow Online does not actually provide a **`pages-generic-article-page`** template. Since there are content type-specific templates available for all the publication's content types, a generic template is not needed.

4.3.2.2 Layouts

The various page templates in a publication will most often have a lot of markup in common: page headers and footers, navigation controls and so on. In order to avoid duplicating this in all the page templates, you can instead create **layouts**. A layout is a specialized top-level template designed to be extensible at specific points. Here is the default layout provided with Tomorrow Online, called **`default-master.twig`**:

```
<!DOCTYPE html>
<html lang="en" prefix="og: http://ogp.me/ns#">
  <head>
    <title>
      {% block title %}
        {{ data.context.name | default(data.context.title) |
 default(data.context.fields.title) }}
```

```
      {% endblock %}
    </title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    {% include "globals-meta" %}
    {% include "globals-stylesheet" %}
    <script src="https://use.typekit.net/cgd1ygv.js"></script>
    <script>try {
        Typekit.load({ async: true });
      }
      catch (e) {
      }</script>
    {% include "globals-browserSync" %}
  </head>

  <body class="base-font bg-primary">
    {% if (devmode) and  errors | length > 0 %}
      {% include "atoms-error-header" with {"errors" : errors} only %}
    {% endif %}

    {% block header %}
      {% include "organisms-header" with {"menu" : data.headerMenu} %}
    {% endblock %}

    <div class="container">
      {% block main %}{% endblock %}
    </div>

    {% block footer %}
      {%- include "organisms-footer"  with {"menu": data.footerMenu} -%}
    {% endblock %}

    {% include "globals-scriptsBlock" with {"config": config} %}
  </body>
</html>
```

As you can see, it contains the basic skeleton of an HTML page. Every section of the layout that is enclosed in a **`{% block`** *name* **`%} {% endblock %}`** structure can be replaced by a template that extends this layout. So, for example, Tomorrow Online's **`topic-page`** template looks like this:

```
{% extends "default-master.twig" %}

{% set intro = data.intro[0] %}
{% set stories = data.results %}

{% block main %}
  <div class="mw-row center cf">
    {% if intro %}
      <div class="cf">
        {% include "atoms-article-header-uppercase" with { "text" :
 intro.fields.title } %}
        {% include "atoms-article-teaser" with {
          "class":"f5 f4-l normal primary fl-l w-two-thirds-l ph4 mt0",
          "text": intro.fields.body_raw
        } %}
      </div>
    {% endif %}

    <div class="bt mh4 mt1 pv3 light-grey f6 relative"></div>
```

```
    <!-- list latest 6 in the tag page -->

    <section class="mw-row center cf pb4 w-100">
      {% for story in stories[:6] %}
        <div class="w-third-l fl-l ph4 pa4-l pv0-l clear-every-third">
          {% include "molecules-story-teaser" with {"story": {"content": story}} %}
        </div>
      {% endfor %}
    </section>

    <div class="bt mh4 mt1 pv3 light-grey f6 relative"></div>

    <!-- List the rest of the tags -->
    <section class="mw-row center cf pb4 w-100">
      {% for story in stories[6:] %}
        <div class="w-two-thirds-l ph4">
          {% include "molecules-top-bottom-styled-list-item-with-section-link" %}
        </div>
      {% endfor %}
    </section>
  </div>
{% endblock %}
```

On the first line, the **`{% extends "default-master.twig" %}`** declaration says that this template is to be based on the layout **`default-master.twig`**. This declaration can then be followed by one or more **`block`** declarations:

```
{% block name %}
...
{% endblock %}
```

where *name* references one of the blocks declared in **`default-master.twig`**.

The content of any blocks declared in a template override the content of the corresponding blocks in the layout. In this case, the **`topic-page`** template only contains one block declaration - **`main`**. The template therefore inherits all the mark-up in the **`default-master.twig`** layout except for the **`main`** block, which it replaces with local markup.

Layouts, unlike all other Twig templates are not stored in the **`templates/_patterns`** folder, but in the **`templates/_layouts`** folder. One layout may be sufficient for many publications, but for publications with pages that don't all have the same overall look and feel it may be necessary to create several layouts.

The following conventions must be followed when naming and referencing layouts:

- The layouts must be stored in the **`templates/_layouts`** folder
- When referencing a layout in an **`extends`** statement, you must include the **`.twig`** extension in the layout name, as in the example above.

4.4 Managing Multiple Publications

The Waiter can be configured to serve multiple publications from the same Cook and Content Store. Depending on how similar the publications are they can either be rendered using shared templates

and styles or a completely separate template tree. Whichever method you choose, separate Patternlab style guides are generated for each publication, so you can use Patternlab to review layouts for all your publications.

4.4.1 Shared Templates and Styles

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

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

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

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

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

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

where:

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

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

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

To make a **`myotherpub`** overlay for the template **`02-medium-teaser.twig`**, for example, you would copy it to a file called **`02-medium-teaser+myotherpub.twig`** and then make whatever changes

you want in the copied file. Similarly, to make a **myotherpub** overlay for **_colors.scss**, you would copy it to a file called **_colors+myotherpub.scss** and then make your required changes.

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

The CUE Front synchronization process copies the contents of the **templates** folder to one or more folders created in a Docker volume that is mounted in the Waiter and Patternlab containers. In an installation with no overlays this volume has only one templates folder, called **templates/_base**, which contains an unmodified copy of the source **templates** folder. If there are overlays, then the volume's **templates** folder contains one subfolder for each overlay name, plus a **_base** subfolder. If you include **+mypub** and **+myotherpub** overlay names in your source templates folder, for example, then the Waiter and Patternlab containers will have access to the following generated templates folders:

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

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

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

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

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

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

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

> A new overlay template folder is automatically created as soon as you rename any file using an overlay name that you haven't used before. In other words, renaming **_colors.scss** to **_colors**

+newpub.scss will cause a **templates/newpub** folder to be created if it doesn't already exist. This is not the case for CSS files, however: a **newpub.css** file will not be automatically created. In order to trigger the creation of a new CSS file, you must restart the **styles** Docker container:

```
docker-compose restart styles
```

This is only required to create a new CSS output file. If you subsequently make further CSS changes, **newpub.css** will be regenerated automatically.

4.4.1.1 Creating a Publication Overlay

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

1. Find the SCSS files in the **templates** folder (in the **tomorrow-online** demo publication, they are located in **templates/theme/css**).
2. Make copies of any SCSS files you want to modify and rename them by adding a *+overlay-name* suffix as described above.
3. Modify the renamed SCSS files as required.
4. Find any Twig templates that contain references to the CSS file **layout.css**. In most cases there should only be two such templates. In the case of the **tomorrow-online** demo publication you would need to update the following two templates:
   - **templates/_patterns/01-globals/05-nonvisual/_stylesheet.twig**: this file references **layout.css** for the publication
   - **templates/_meta/_00-head.twig**: this file references **layout.css** for the style guide
5. Make copies of these templates and rename them by adding a *+overlay-name* suffix as described above.
6. Open the renamed templates for editing and replace **layout.css** with *overlay-name*.**css**.
7. Find any other files that you want to modify (**.twig**, **.json**, **.md**, **.ico**). Copy them, rename them as described above and modify them to meet your requirements.
8. Reconfigure the Waiter to recognise and handle the publication you have created the overlays for as described in section 13.7.
9. Reconfigure **nginx** to recognise and handle the publication you have created the overlays for (also described in section 13.7).
10. Finally, restart the Waiter and Styles containers:

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

4.5 Extending CUE Front

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

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

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

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

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

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

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

What kind of front end do you have?
If you are using CUE's Waiter as your front end and you want to extend the front end, then you might want to create a separate component that sits alongside the Waiter rather than directly extending the Waiter itself. This will give you complete control over how you implement the service: you can make use of the languages, tools and libraries of your choice without having to conform to any limitations imposed by the Waiter. If, however, you wish to extend the Waiter directly, it offers a set of hooks that you can use to modify and extend its behavior in various ways. For details see chapter 5.

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

Routing

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

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

4.6 CUE Front Development Environment

To be supplied.

5 Writing Waiter Extensions

The Waiter provides a simple extensions framework in the form of:

- A **`WaiterExtension`** class that you can extend to create your extensions
- A set of hooks that the Waiter calls at various points during the handling of a request

To implement an extension you create a class based on **`WaiterExtension`** that listens for one or more hooks, and takes appropriate action when the hooks are called. The hooks are all called in sequence at specified points during a request, and the Waiter passes a hook object to the extension. The hook object is an array. Each hook can add items of information to the hook object, so that the total amount of information available increases as the request progresses. An extension can also add information to the hook object, or modify existing information in the object.

Waiter extensions must belong to the namespace **`Extensions`**.

In order to be found by the Waiter, an extension must be placed in your CUE Front installation's **`service/waiter/waiter-extensions/Extensions`** folder.

5.1 The WaiterExtension Class

```
<?php namespace Extensions;

class WaiterExtension {
  public function register(\Waiter\WaiterClient $waiterClient, $extensionConfig =
 array()) {
    new static($waiterClient, $extensionConfig);
  }
}
```

5.2 Registering Hooks

Register a hook in your extension class as follows:

```
on ( hook-name, [ $this, "function-name"], [ condition ] )
```

where:

hook-name
: is one of the supplied Hook constants - **`Hook::INIT`**, for example.

function-name
: Is the name of a function in your extension class. This is the callback function that will be executed when the hook is called.

condition
: Is an optional Boolean expression that determines whether or not *function-name* will actually be called. The default value of *condition* is **`true`**, so if you don't specify a condition then *function-name* will always be called.

5.3 Callback Function Return Values

A callback function may either return nothing at all, or it can return an array (which may be the hook object). Whatever is returned is merged with the hook object. Merging in this case means that any new elements in the returned array are added to the hook object, and any elements that already exist in the hook object are overwritten by the returned object's elements.

5.4 The Extension Hooks

You can use the following **`Hook`** constants to refer to the hooks in your extensions. The hooks are called in the following order (except for **`ERROR`**, which may of course be called at any point):

`INIT`
This hook is called immediately after a request is received by the Waiter. The hook object is empty.

`AFTER_CONTEXT_RESOLVED`
This hook is called once the Waiter has determined which publication the request is directed to. The following items have been added to the hook object:

`currentPageUrl`
The URL of the requested page.

`contextPublication`
An object containing information about the publication to which the requested page belongs.

`BEFORE_COOK_REQUEST`
This hook is called when the request is ready to be forwarded to the Cook. The following items have been added to the hook object:

`cookURL`
The URL of the Cook to which the request is to be forwarded.

`requestOptions`
The request headers to be sent to the Cook with the request.

`AFTER_COOK_REQUEST`
This hook is called immediately after a response is received from the Cook. The following item has been added to the hook object:

`response`
The response returned by the Cook.

`AFTER_TWIG_ENVIRONMENT_READY`
This hook is called once the Waiter has prepared the Twig environment for processing the response. The following item has been added to the hook object:

`twig`
An object containing the Twig environment for processing the response.

`BEFORE_TEMPLATE_RENDER`
This hook is called immediately before the response is sent for rendering by Twig. The following item has been added to the hook object:

response
The final response, as prepared by the Waiter. It overwrites the earlier value of response added by the **AFTER_COOK_REQUEST** hook.

ERROR
This hook can be called at any time if an error occurs. The following item has been added to the hook object:

error
The exception object.

5.5 Registering Extensions

In order to be used, an extension must be registered in the Waiter's configuration file, **waiter-config.yaml** as follows:

```
extensions:
  - name: \Extensions\extension-name
```

where *extension-name* is the name of your extension without its **.php** extension. If, for example you have an extension called **MyExtension.php**, then you would register it as follows:

```
extensions:
  - name: \Extensions\MyExtension
```

You can also include configuration parameters for your extension under a **config** entry. For example:

```
extensions:
  - name: \Extensions\MyExtension
    config
      myparam: myvalue
```

You can structure the configuration values in any way you like. Here, for example is the configuration for the Waiter's built-in landing page resolver:

```
extensions:
  - name: \Extensions\LandingPageResolver
    config:
      landingPages:
        subtitles: pages-vtt
        oembed: pages-oembed-index
```

5.6 Example Extension

The Waiter's built-in landing page resolver is implemented as an extension, and illustrates how an extension should be written:

```
<?php namespace Extensions;

use \Waiter\WaiterClient;
use \Waiter\Hook;

class LandingPageResolver extends WaiterExtension {
```

```
  private $config = array();

  public function __construct(WaiterClient $waiterClient, $extensionConfig) {
    $this->config = $extensionConfig;
    $waiterClient->getHooks()->on(Hook::BEFORE_TEMPLATE_RENDER, [$this,
"resolveStartingTemplate"]);
  }

  public function resolveStartingTemplate($hookObject) {
    $response = $hookObject["response"];
    $twig = $hookObject["twig"];
    $landingPages = $this->config['landingPages'];

    $wireframe = isset($response['data']) && isset($response['data']['wireframe']) ?
      $response['data']['wireframe'] : null;

    if (isset($wireframe)) {
      $this->renderTemplate('pages-' . $wireframe . '-index', $twig, $response);
    }
    else if ($this->hasSubtitles($response)) {
      header('Content-type: text/vtt');
      $this->renderTemplate($landingPages['subtitles'], $twig, $response);
    }
    else if ($this->isOEmbedResponse($response)) {
      //to get around oembed template for picture. Right now multiple recipe extension
doesn't work in chain.
      // as there is an extension for image already in place, oembed extension doesn't
have any effect.
      $this->renderTemplate($landingPages['oembed'], $twig, $response);
    }
  }

  private function hasSubtitles($response) {
    return isset($response['data']) && isset($response['data']['subtitles']);
  }

  private function isOEmbedResponse($response) {
    $remainingPath = isset($response['data']['resolution']['remainingPath']) ?
      $response['data']['resolution']['remainingPath'] : '';

    return strpos($remainingPath, 'oembed') > 0;
  }

  private function renderTemplate($template, $twig, $response) {
    echo $twig->render($template, $response);
    exit;
  }
}
```

6 Using the Fridge

The Fridge is an nginx instance run as a cache. You can use the Fridge in two different ways:

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

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

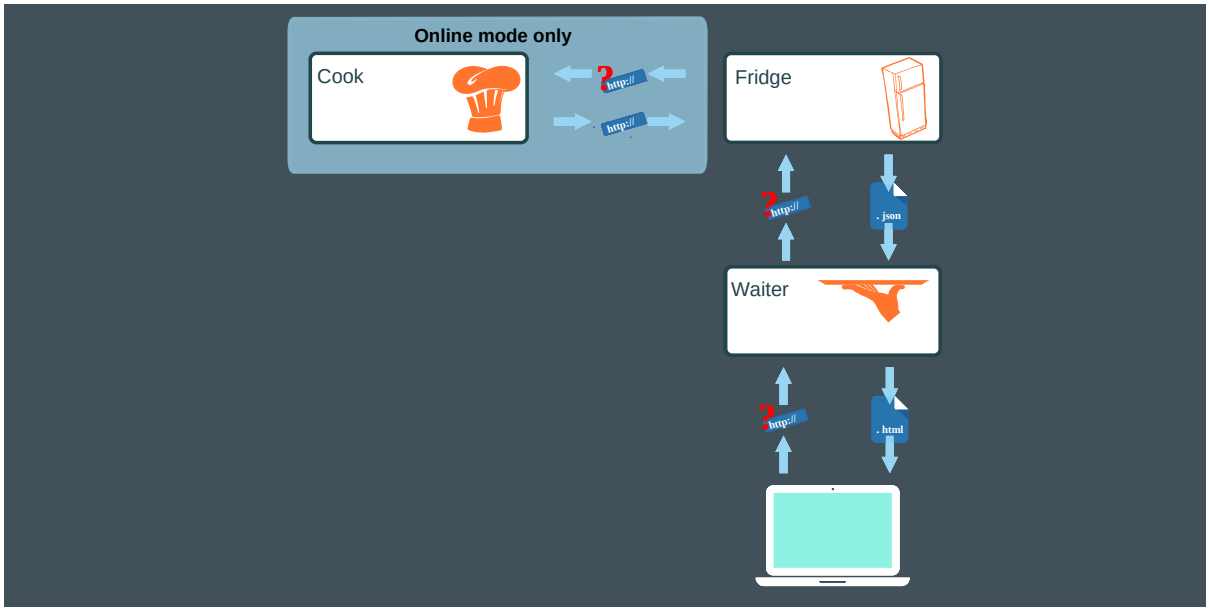
Internally, the Fridge can be configured to operate in two different modes:

- **Offline mode**, in which case it only ever looks for resources in its cache folder. If a requested resource cannot be found there, then the Fridge returns an `HTTP 404` not found response.
- **Online mode**, in which case the Fridge acts as a true proxy: if a requested resource cannot be found in the cache folder, then the Fridge forwards the request to the original server (the Content Store 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

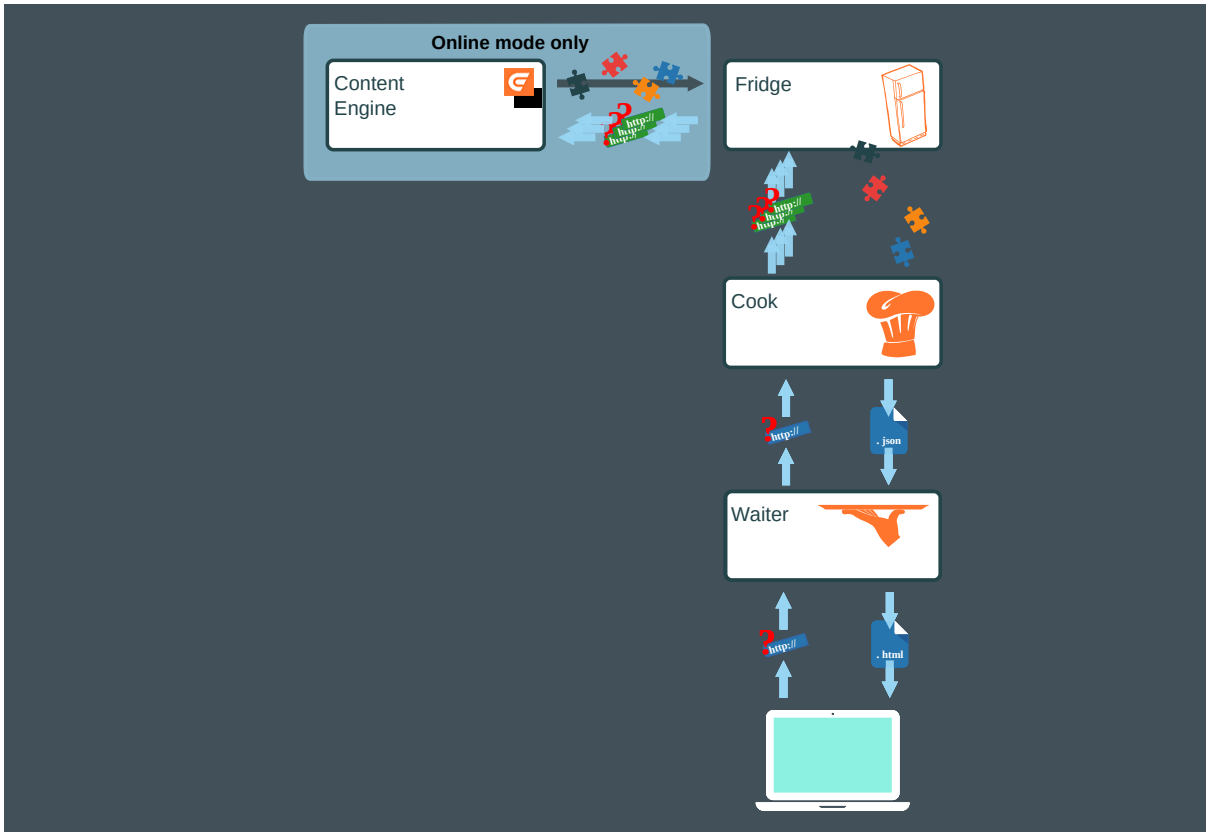
6.1 Fridge as Cook Proxy



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

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

6.2 Fridge as Content Store Proxy



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

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

6.3 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 Store. 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 Store.

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, a web service called the Fridge Stocker monitors all changes made to the content in the Content Store. Every time a change is made to any content, the Fridge Stocker pushes the change to the Fridge, ensuring that the Fridge's contents are always fresh. As long as the Fridge Stocker is enabled in the current setup, it is automatically started together with the Fridge. (If you selected All when running `setup`, then the Fridge Stocker **is** enabled.)

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

- It improves the scalability of the system by completely decoupling the presentation layer from the Content Store and the editorial system. Increases in audience can be met by simply duplicating CUE Front components, without any need to scale the Content Store or its database.
- It improves the reliability of the system: the Content Store can be taken off line without affecting the presentation layer in any way.
- It can enable improved performance by allowing the Fridge's cache to be stored in a content delivery network, for example.

> If you configure the Fridge Stocker manually rather than via the `setup` utility, it is important to ensure that it is configured with the same user credentials as the Cook. If this is not the case, then the Cook will not get updated content.

6.3.1 Ensuring Plug-in Data is Handled

The Fridge Stocker constantly monitors the Content Store for content changes, and pushes those changes to the Fridge, ensuring that the Fridge contains a true copy of the Content Store. However, the Fridge Stocker does not monitor changes to additional data managed by Content Store plug-ins. The following plug-ins generate or manage data that the Fridge Stocker does not monitor:

- Menu Editor
- CUE Live

- Poll

You can, however, ensure that changes to data managed by these plugins propagates through to the web site by configuring the Fridge's **nginx** cache to use its standard expiration mechanism for the paths used by the plug-ins. To configure nginx you need to edit the Fridge's config file, **fridge.conf**, which you will find in your CUE Front installation's **service/fridge** folder. For each of the above plug-ins installed at your site, add an entry containing the **proxy_cache_valid** parameter as follows:

```
location ~ ^/menu-webservice/ {
  proxy_pass http://$http_host$request_uri;
  # Valid for 1 hour
  proxy_cache_valid any 60m;
}

location ~ ^/poll-ws/poll/.* {
   proxy_pass http://$http_host$request_uri;
   # Valid for 20 seconds
   proxy_cache_valid any 20s;
}

location ~ ^/live-center-presentation-webservice/.* {
   proxy_pass http://$http_host$request_uri;
   # Valid for 20 seconds
   proxy_cache_valid any 20s;
}
```

The **proxy_cache_valid** parameters sets the expiration time for content on the path specified with **location**. Content items are marked as expired after the specified time has elapsed, and will be retrieved from the Content Store the next time they are requested. For the examples shown above, this means that:

- Any menu changes that are made may take up to an hour to propagate through to the web site.
- Poll and CUE Live data changes will only take up to twenty seconds to propagate through to the web site.

You can set **proxy_cache_valid** to whatever time period you think is acceptable in each case. For further information about the **proxy_cache_valid** parameter, see http://nginx.org/en/docs/http/ngx_http_proxy_module.html#proxy_cache_valid.

> If you add these settings to an existing Fridge configuration, you should be aware that the changes only apply to newly cached objects. You will therefore need to manually clear any existing content from the cache for the affected plug-ins.

7 Using Data Sources

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

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

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

You can make the following kinds of data source queries:

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

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

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

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

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

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

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

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

7.1 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://localhost: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 4.2.1.1).

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

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

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

```
query {

}
```

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

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

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

```
      type(name: "story")
    }
  }
```

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

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

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

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

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

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

will generate this Solr query:

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

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

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

will generate this Solr query:

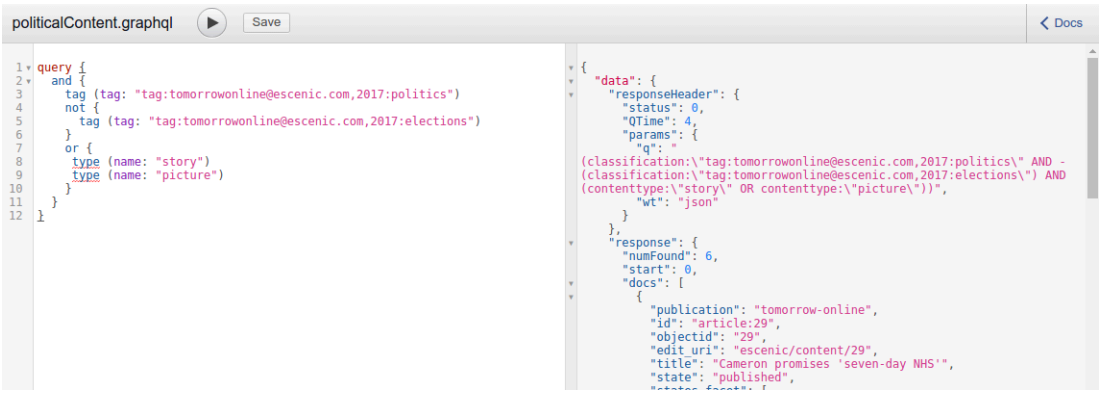
```
(classification:"tag:tomorrowonline@escenic.com,2017:politics"
AND -(classification:"tag:tomorrowonline@escenic.com,2017:elections")
AND (contenttype:"story" OR contenttype:"picture"))
```

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

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

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

For example:



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

> The top level filter **`related()`** is different from all the other data source filters in that it is not implemented using Solr, but works by directly accessing the Content Store web service. For further information, see section 7.3.12.

7.1.1 Data Source Context

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

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

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

and execute it at **http://localhost: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://localhost:8101/tomorrow-online/_datasource/tagRelatedStories/edit**, then it will return no results because the context is a section, and sections have no tags.

7.1.2 Using Filter Aliases

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

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

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

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

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

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

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

7.2 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 one of the content retrieval functions called **datasource()** or **extendedDatasource()**. If you go back to editing the section page content retrieval query (**index-page.graphql**) at **http://localhost: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, but by default it does have to be a top-level entry in the query, at the same level as **resolution**, **context**, **menu** and so on. You can, however, configure the Cook to allow the use of the **datasource()** function in different locations, see section 7.2.3.)

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

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

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

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

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

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

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

7.2.1 The extendedDatasource Function

The **extendedDatasource()** function has exactly the same input parameters as the **datasource()** function, but returns some additional information. Along with the actual results from the executed data source, **extendedDatasource()** returns metadata about the result set. The metadata includes the total number of result items, query execution time, offset, item count and so on. This information can be used (for example) to paginate data source search results.

The inclusion of these additional fields mean that the result set is returned in an array called **items**. Whereas **datasource()** directly returns an array of **AtomEntry** objects, **extendedDatasource()** returns the following structure:

```
total: Int
time: Float
offset: Int
count: Int
items: [AtomEntry!]!
```

7.2.2 Datasource Function Parameters

The **datasource()** and **extendedDatasource()** functions have identical parameters that can be used to control, limit and organize the returned content items:

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

Only the **name** parameter is required.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

7.2.3 Changing The Data Source Context

Some data source filters (currently only one, the **`sharedTags()`** filter) make use of the data source context. The **`sharedTags()`** filter returns content items that share tags with the content item in the data source context. By default, that context is the context of the Graphql query from which the data source is called. So if the current context is a section, for example, **`sharedTags()`** will always return 0 results.

You can, however, control the context of the data source by calling **`datasource()`** (or **`extendedDatasource()`**) at the content level of your query rather than at the top level. For example, while processing a teaser on a section page:

```
fragment teaser on AtomLink {
  ...
  content: follow {
    type: __typename
    ... on Story {
      relatedStories: datasource(name:"shared_tags") {
         ...
      }
      ...
    }
}
```

In this case, the content item referenced in the teaser is passed to the data source as its context.

By default you cannot do this: the GraphiQL editor will not offer **`datasource()`** as an option at this location in the query. You can, however, configure it to do so for specific, named content types. If you want to be able to use your **`story`** and **`video`** content types as data source contexts then you can enable this by adding the following to your Cook config file:

```
recipedata:
  extensions:
    - name: '@escenic/cue-front-extension-datasources'
      config:
        extendTypes:
          - Story
          - Video
```

> Note that the first letter of the content type name must be capitalized when specified here, even if it isn't capitalized in the **`content-type`** resource. This is because you're actually specifying the name of the GraphQL type that corresponds to your content type, not the content type itself.

7.3 Data Source Reference

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

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

7.3.1 Query

```
query(
  [$param-name:param-type]*
)
{
  [ and | or | related() ]
}
```

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

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

The root of a data source query. The results of all the **`query`**'s child filters are concatenated in a single result data structure. No attempt is made to sort results or remove duplicate entries. This can, however, be done by the content retrieval **`datasource()`** function that calls a data source query (see chapter 7).

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

```
{
```

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

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

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

where:

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

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

The parameters can be referenced in the body of the query using the same names, also preceded by a **$** sign. The parameters are passed in to the query from the **datasource()** function's **params** parameter (see section 7.2.2).

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

7.3.2 And

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

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

7.3.3 Or

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

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

7.3.4 Not

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

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

7.3.5 Publication

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

Returns all content items belonging to a publication.

Parameters

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

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

7.3.6 Section

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

Returns all content items belonging to a specified section.

> The **`section`** filter requires the CUE Content Store to be correctly configured, or else it will not work. Specifically, the **`types`** property in the Content Store's index configuration must include the value **`section`**. If the Cook is configured to use the **`editorial`** Solr core, then this should be the case by default, but if it is using the **`presentation`** Solr core, then you will need to explicitly add it. To do so, open **`/etc/escenic/engine/common/com/escenic/search/index/PresentationIndexConfiguration.properties`** for editing on your Content Store host, and make sure any **`types`** entry contains the value **`section`**. If there is no **`types`** entry in the file (or if there was no **`PresentationIndexConfiguration.properties`** file and you have created one), then add the following setting to the file:
>
> ```
> types=section,article
> ```
>
> The Content Store must then be restarted.
>
> For general information about editing Content Store configuration files, see Configuring The Content Store.

Parameters

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

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

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

7.3.7 Author

```
author
```

Filters the result set by author, where the author to filter on is derived from the context in which the data source is executed. In other words, this filter returns all content items authored by the current

person. If the current object is a content item rather than a person, then the results returned by author depend on whether it is the child of an **or** filter or an **and** filter. As the child of an **or** filter it will return all content items authored by **at least one** of the current content item's authors. As the child of an **and** filter it will return all content items authored by **all** the current content item's authors. This filter can only return data when the context object is either a person or a content item. In all other contexts **author** will always return 0 results.

> The **author** filter requires both Solr and the CUE Content Store to be correctly configured, or else it will not work. The specific requirements are:
>
> - The Solr schema (**/etc/escenic/solr/***solr-core***/schema.xml** on your Content Store host) must contain the following field definitions:
>
>   ```
>   <field name="author_id_s" type="string" indexed="true" stored="false"
>    multiValued="true" />
>   ```
>
>   This definition is **not** included in the default **editorial** and **presentation** schemas delivered with the Content Store. After making the change you will need to regenerate the Solr index. For general information about modifying the Content Store's Solr schemas and re-indexing, see Modifying The Standard Configuration.
> - The **types** property in the Content Store's index configuration must include the value **person**. To add it, open **/etc/escenic/engine/common/com/escenic/search/index/PresentationIndexConfiguration.properties** for editing on your Content Store host, and add the following setting to the file:
>
>   ```
>   types=section,article,person
>   ```
>
>   The Content Store must then be restarted.
>
>   For general information about editing Content Store configuration files, see Configuring The Content Store.

7.3.8 Type

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

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

Parameters

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

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

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

7.3.9 Tag

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

```
)
```

Returns all content items tagged with a specified tag.

Parameters

The **`tag`** parameter is required.

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

Examples

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

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

7.3.10 Shared Tags

```
sharedTags
```

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

7.3.11 Field

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

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

Parameters

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

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

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

Examples

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

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

7.3.12 Related

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

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

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

Note the following:

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

Parameters

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

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

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

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

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

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

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

Examples

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

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

Returns a, b, c, d and e.

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

Returns d, e, a, b and c.

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

Returns b, c, d and e.

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

Returns a and b.

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

Returns b and c.

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

Returns b, c and e.

The purpose of the related() filter

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

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

7.3.13 WithRelationToMe

```
withRelationToMe(
  type: String
)
```

Returns all content items that have a relation pointing to the context content item (the reverse of the search performed by the **`related()`** filter).

Parameters

`type`: ***String***
The type of relation to search for. If specified, then only content items that have the specified type of relation will be returned. Otherwise, all content items with relations pointing to the context content item are returned, irrespective of type.

Examples

```
withRelationToMe()
```

Returns all content items with a relation pointing to the context content item.

```
withRelationToMe(type: "relatedMedia")
```

Returns all content items with a **`relatedMedia`** relation pointing to the context content item.

7.3.14 WithRelationTo

```
withRelationTo(
  contentItem: Int!
  type: String
)
```

Returns all content items that have a relation pointing to the specified content item.

Parameters

`contentItem`: ***Int!***
The content item from which to search, specified by ID (that is, Content Store internal ID).

`type`: ***String***
The type of relation to search for. If specified, then only content items that have the specified type of relation will be returned. Otherwise, all content items with relations pointing to specified content item are returned, irrespective of type.

Examples

```
withRelationTo(contentItem: 45)
```

Returns all content items with a relation pointing to the content item with the Content Store internal ID **`45`**.

```
withRelationTo(contentItem: 45, type: "relatedMedia")
```

Returns all content items with a **`relatedMedia`** relation pointing to the content item with the Content Store internal ID **`45`**.

8 Working with the Recipe

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

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

The recipe is responsible for:

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

Much of the recipe's work is done by GraphQL queries, and therefore a great deal of customization work can be done using GraphQL (see section 4.2) and data sources (see chapter 7). At many installations, the recipe itself will never need to be modified.

You may in some cases need to modify the behavior of a recipe extension by setting a configuration parameter. For information on how to do this, see section 8.1.

Larger customizations may require changes to the recipe, for example:

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

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

8.1 Configuring Recipe Extensions

Recipe extensions can expose configuration parameters that are set in the Cook's configuration file, **`cook-config.yml`**. Most of the default extensions supplied by Stibo DX (the ones downloaded from **`npm.escenic.com`**) have very few configuration parameters, and where necessary, the setup tool ensures that they are supplied with sensible values. You may, however, need to modify some of the defaults set by the setup tool or add some additional settings of your own.

If you do need to configure a recipe extension, you can do so by creating a **`cook-config.yaml`** override file in your publication repository, as described in section 13.6, and adding the necessary parameter settings to it.

For information about all the configuration parameters exposed by the default extensions, check the **readme** files on http://npm.escenic.com/. You will, for example, find information about the **cue-front-extension-binary** extension at http://npm.escenic.com/#/detail/@escenic/cue-front-extension-binary.

Reasons for needing to configure a recipe extension include:

- Support for image content types that are not called **picture** (see section 8.1.1)
- Support for custom story element types (see section 8.1.2)
- Changing the default URL-GraphQL query mappings (See section 8.1.3)

8.1.1 Configuring Image Content Types

By default, the **cue-front-extension-image** extension is configured to assume that:

- Content items of the type **picture** contain images
- The image's crop definitions are store in a content item field called **alternates**.

This is the case for the **tomorrow-online** demo publication, so it works out of the box. If it is not the case for your publication, then you will need to reconfigure it. To do so, add something like this to the **cook-config.yaml** override file in your publication repo:

```
recipedata:
  extensions:
    - name: '@escenic/cue-front-extension-image'
      config:
        representations:
          photo: crops
```

This configuration specifies that:

- Content items of the type **photo** contain images
- The image's crop definitions are store in a content item field called **crops**.

You might actually have several content types that contain images, in which case you will need to add several entries under **representations**:

```
recipedata:
  extensions:
    - name: '@escenic/cue-front-extension-image'
      config:
        representations:
          photo: crops
          image: crops
          map: crops
```

Note that the **cue-front-extension-image**'s **endpoint** parameter should **not** be included in the override file: this is set by the setup tool, ensuring consistency with other endpoint settings in the configuration files.

8.1.2 Configuring Story Element Types

A number of default story element types are delivered with the Content Store, and the **cue-front-extension-storyline** recipe extension is configured to support all of these out of the box.

If, however, you add your own story element types, then you **may** need to add corresponding configurations to the **cook-config.yaml** override file in your publication repo. You only need to do this for story element types that contain links to related content items (such as the default **image** story element type).

By default, **cue-front-extension-storyline** is configured as follows:

```
recipedata:
  extensions:
    - name: @escenic/cue-front-extension-storyline
      config:
        extendType: "Storyline"
        relation-elements:
          internal_link: relation
          relation: relation
          image: relation
```

the highlighted entries configure the extension to expect **internal_link**, **relation** and **image** story element types to contain links to content items. If you have created a story element type of your own called **my_link** that contains links to content items, then you would need to add this to the **cook-config.yaml** override file in your publication repo:

```
recipedata:
  extensions:
    - name: @escenic/cue-front-extension-storyline
      config:
        relation-elements:
          my_link: relation
```

The default configuration of **cue-front-extension-storyline** only supports one storyline content type, called **Storyline**. You might, however, want to create other storyline content types (a longform storyline, for example), for which different templates and so on are used. The **extendType** field in the **cue-front-extension-storyline** config file can be used as an array, making this possible. To add a **Longform** content type, you would need to add the following to the **cook-config.yaml** override file in your publication repo:

```
recipedata:
  extensions:
    - name: @escenic/cue-front-extension-storyline
      config:
        extendType:
          - "Storyline"
          - "Longform"
```

8.1.3 Configuring URL-GraphQL query mappings

The GraphQL query that is executed to retrieve content for any given URL is determined by the **cue-front-extension-path-mapper** recipe extension. **cue-front-extension-path-mapper** does this by applying mapping rules specified in **cook-config.yaml**.

By default, **cook-config.yaml** contains the following mapping rules:

```
    - name: @escenic/cue-front-extension-path-mapper
      config:
        directory: "esi"
        criteria:
```

```
          match: '\.esi\/([a-zA-Z0-9-]+)'
        base-name: "$1"
        lookup:
        - BASE
    - name: @escenic/cue-front-extension-path-mapper
      config:
        directory: "."
        lookup:
        - TYPE-SECTION
        - TYPE
```

These rules basically say:

- The response for any URL that starts with the string **`.esi/`** will be generated by the query called **`esi/`***base-name***`.graphql`** where *base-name* matches the URL segment identified by **`$1`** (in this case, the remainder of the URL – so long as it only contains alphanumeric characters and hyphens). This rule is using the **`BASE`** lookup algorithm.
- The response for any other URL will be generated according to the rules described in section 4.2.3, using the **`TYPE-SECTION`** and **`TYPE`** lookup algorithms.

`cook-config.yaml` may contain as many of these mapping rules as you need. The rules are applied in the order they are defined in **`cook-config.yaml`**, and the first rule to match a URL is used. The same applies when multiple lookup algorithms are applied within a single rule as in the second rule above: the first algorithm to get a match wins.

For a full description of all the available **`cue-front-extension-path-mapper`** lookup algorithms and how to use them, see the cue-front-extension-path-mapper README file.

8.2 Making a Recipe Extension

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

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

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

`context`
The **context object**, an object containing information about the current request. For details see section 8.2.2.

This method is **required**.

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

`assembler`
The SchemaAssembler object.

`model`
The CUE Front GraphQL model.

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

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

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

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

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

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

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

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

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

`context`
The **context object**, an object containing information about the current request. For details see section 8.2.2.

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

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

`recipe`
The recipe object itself, created in **`recipe.js`**. For a detailed description of the recipe object, see section 8.2.1.

`context`
The **context object**, an object containing information about the current request. For details see section 8.2.2.

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

This method is **required**.

8.2.1 The Recipe Object

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

`log`
A Bunyan logger object.

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

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

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

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

8.2.2 The Context Object

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

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

`log`
A Bunyan logger object.

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

cook
The CUE Front Cook.

schema
The assembled GraphQL schema object.

model
The model type for the current request.

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

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

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

8.2.3 Extension Configuration

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

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

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

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

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

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

```
recipedata:
...
  extensions:
```

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

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

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

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

8.3 Recipe Extension Tutorial

This tutorial provides a step-by-step guide to building a simple recipe extension. Before you start, make sure that you have a correctly installed CUE Front start pack and demo publication (Tomorrow Online) that you can work with.

8.3.1 Creating the Recipe Extension

If you haven't created any recipe extensions before, the the first thing you need to do is to create a folder to hold your extensions. CUE Front doesn't insist on any particular location, but it's a good idea to keep them in the **recipe** folder, so create an **extensions** folder there:

```
cd path/cue-front
mkdir recipe/extensions/
```

Create a file called **tutorial.js** in the new folder:

```
touch recipe/extensions/tutorial.js
```

and open it in a text editor. For example:

```
nano recipe/extensions/tutorial.js
```

Paste the following code into the file and save it:

```
exports.pattern = '/?hello';
exports.run = function(recipe, context, done) {
  done({entity: { "Hello": "world"}});
```

```
};
```

All this extension does is:

- Supply a regular expression in **exports.pattern**.
- Supply a function to be executed in **exports.run**.

The recipe uses **exports.pattern** to test the **resolvedRemainingPath** variable, and if it matches, executes the function supplied with **exports.run**. The function in our extension terminates the recipe execution and supplies the value **{entity: { "Hello": "world"}}** for the recipe to return to the Cook.

> The pattern **/?hello** is used in order to match the string **/hello** both at the end of a section URL such as **http://localhost:8101/tomorrow-online/hello** and at the end of a story URL such as **http://localhost:8101/tomorrow-online/politics/2017-11-29/Where-will-winding-up-Westminster-lead-1754.html/hello**. When the resolver parses a URL, it treats **/** characters as "belonging to" sections and removes them from the **resolvedRemainingPath**. So for the URL **http://localhost:8101/tomorrow-online/hello**, the resolver will leave just **hello** in **resolvedRemainingPath**, while for **http://localhost:8101/tomorrow-online/politics/2017-11-29/Where-will-winding-up-Westminster-lead-1754.html/hello** it will leave **/hello** in **resolvedRemainingPath**. The pattern **/?hello** matches both **hello** and **/hello**, and will therefore catch both occurrences.

8.3.2 Configuring Docker

By default, the Cook will not be able to access your new extension because it runs in a Docker container, and only has access to files and folders that are explicitly mounted inside that container. To make Docker mount your **extensions** folder:

1. Open **docker-compose.yml** in a text editor.
2. Search for the following line:

   ```
   #- ./recipe/extensions:/srv/recipe/extensions:ro
   ```

3. Remove the **#** comment marker at the start of the line. Make sure that the hyphen at the start of the resulting line is indented the same amount as the lines above and below.
4. Save the file.

The next time you restart the container, the **/recipe/extensions** folder will be available inside the Cook's container as **/srv/recipe/extensions**.

8.3.3 Configuring the Cook

Your extension will now be available inside the Cook's container the next time it is started, so the next step is to tell the Cook to include the extension in it's recipe. You do this by modifying the Cook configuration file, **cook-config.yml**. The **cook-config.yml** used by the Cook is one you have generated with the **setup** utility. If, for example, you created a configuration set called **myconfig** when installing CUE Front, then the Cook will be using a configuration file called **setup/myconfig/cook-config.yml**. You could, in theory, edit this file directly, but it is not a good idea to do so because if you did so then your changes would be lost if you later modified your configuration set using **setup**.

The recommended way to make this change is to edit the default **cook-config.yml** (**setup/defaults/cook-config.yaml**) and then use **setup** to regenerate your configuration set:

1. Open **setup/defaults/cook-config.yaml** in a text editor.
2. Search for:

   ```
   recipedata:
   ```

   Then under **recipedata** look for the indented subentry

   ```
     extensions:
   ```

3. Under **extensions** you will see a long list of extension declarations starting with **- name**. Add your extension to the bottom of this list:

   ```
       - name: @escenic/cue-front-extension-notfound
       - name: @escenic/cue-front-extension-binary
       - name: "/srv/recipe/extensions/tutorial"
   ```

   Note that the path of your extension is different from the path of the built-in extensions. The built-in extensions are downloaded from Stibo DX's NPM server, whereas your extension is loaded from the Cook container's file system.
4. Regenerate your configuration set using the modified defaults:

   ```
   cd cue-front-path/setup
   docker-compose run --rm setup generate myconfig
   ```

5. Restart the CUE Front containers:

   ```
   cd cue-front-path/myconfig
   docker-compose down
   docker-compose up -d
   ```

8.3.4 Testing and Debugging

You should now have a working extension, which you can test by visiting the Cook's endpoint (**localhost:8101/tomorrow-online/**)in a browser or using **curl** in a terminal window:

```
curl http://localhost:8101/tomorrow-online/
```

should return the normal JSON output for the Tomorrow Online front page, but

```
curl http://localhost:8101/tomorrow-online/hello
```

should return just **{Hello: "world"}**.

The same thing should happen if you visit a story page:

```
curl http://localhost:8101/tomorrow-online/politics/2017-11-29/Where-will-winding-up-
Westminster-lead-1754.html
```

and then append **/hello** to the URL:

```
curl http://localhost:8101/tomorrow-online/politics/2017-11-29/Where-will-winding-up-
Westminster-lead-1754.html/hello
```

8.3.5 Restricting the Extension's Scope

The scope of the **tutorial** extension is currently unlimited: it responds to **/hello** at the end of any Tomorrow Online URL – even an invalid one that would return "page not found" without the **/hello** suffix.

It is possible, however, to limit the extension's scope, so that it only responds to **/hello** in certain contexts. You might, for example, only want the extension to respond for URLs in the publication's Sports section. To do this you need to add a **constraint** to the **exports** object in your **tutorial.js** extension, as follows:

```
exports.pattern = '/?hello';
exports.constraint = function(context) {
  const resolution = context.request.resolution.entity;
  return resolution.section == 'sport';
};
exports.run = function(recipe, context, done) {
  done({entity: { "Hello": "world"}});
};
```

Visiting **http://localhost:8101/tomorrow-online/hello** or **http://localhost:8101/tomorrow-online/politics/2017-11-29/Where-will-winding-up-Westminster-lead-1754.html/hello** will now return **{ error: "Page not found!" }**. Appending **/hello** to a URL in the Sports section, however, such as:

```
curl http://localhost:8101/tomorrow-online/sport/hello
```

or

```
curl http://localhost:8101/tomorrow-online/sport/2017-11-29/Four-miss-Englands-trip-to-Italy-1718.html/hello
```

will trigger the extension and return **{ Hello: "world" }**.

If you only want the extension to be triggered for section pages (i.e **http://localhost:8101/tomorrow-online/sport/hello**) and not for actual stories such as **http://localhost:8101/tomorrow-online/sport/2017-11-29/Four-miss-Englands-trip-to-Italy-1718.html/hello**, you can do so by adding a **resolution.context** condition to the constraint as follows:

```
exports.constraint = function(context) {
  const resolution = context.request.resolution.entity;
  return resolution.section == 'sport' && resolution.context == 'sec';
};
```

This limits the extension to section pages only. If you want to limit it to content items only then you need to specify **&& resolution.context == 'art';** instead.

8.3.6 Parsing the URL

So far, the **tutorial.js** extension has just used a simple URL suffix (**hello**) as a mechanism to trigger the extension. You can, however, also allow the extension to be triggered by a range of different suffixes that match a specified pattern. In such cases, you will then often want to parse the suffix in order to extract specific values from it, so that you can use them for some purpose. A regular expression like this:

```
^/?([a-z]{2})/([0-9]{2})$
```

does both these things:

- It matches any string consisting of an optional **/** followed by two lower case characters followed by a **/** followed by two digits. So it will match **ab/12** or **gc/23** or **/fl/93** but not **foo/12** or **a3**.
- The brackets around **[a-z]{2}** and **[0-9]{2}** mean that a regular expression engine will return the values that match those parts of the expression as separate values.

So a more sophisticated version of **tutorial.js** can take advantage of this possibility as follows. First, change **exports.pattern** to use this more complex regular expression:

```
exports.pattern = '^/?([a-z]{2})/([0-9]{2})$';
```

Then insert an extra line to keep a copy of the regular expression:

```
const pattern = new RegExp(exports.pattern);
```

Now, in the **exports.run** function, test the remaining path (which is available as **context.resolvedRemainingPath**) against the saved regular expression:

```
  const items = pattern.exec(context.resolvedRemainingPath);
```

This will return an array of values to **items**. If the string matches the pattern (and we know in this case that it always will, since otherwise the **exports.run** function would never be called), the first element of the array (**items[0]**) will contain the entire matched string and the following elements will contain substrings matched by bracketed segments of the regular expression. So for **/fl/93**, **items[1]** will contain **fl** and **items[2]** will contain **93**. This means you can return these values in the extension's output as follows:

```
  done({entity: {"letters": items[1], "digits": parseInt(items[2])}});
```

Here is the new improved version of **tutorial.js**:

```
exports.pattern = '^/?([a-z]{2})/([0-9]{2})$';

const pattern = new RegExp(exports.pattern);

exports.constraint = function(context) {
  const resolution = context.request.resolution.entity;
  return resolution.section == 'sport' && resolution.context == 'sec';
};

exports.run = function(recipe, context, done) {
  const items = pattern.exec(context.resolvedRemainingPath);
  done({entity: {"letters": items[1], "digits": parseInt(items[2])}});
};
```

If you now visit your publication's **/sport/ab/05** page:

```
curl http://localhost:8101/tomorrow-online/sport/ab/05
```

you should get the response:

```
{
  letters: "ab",
  digits: 5
```

```
}
```

8.3.7 Passing on the Request

A recipe is composed of many different extensions, which are executed in sequence. The order in which they are executed is determined by **priority**. Most extensions do not have an explicitly set priority, and are therefore executed in the order they appear in **`recipe/package.json`**.

`cue-front-extension-run-default` and **`cue-front-extension-not-found`**, however, are exceptions to this rule:

- **`cue-front-extension-run-default`** has a priority of 98 and runs second to last.
- **`cue-front-extension-not-found`** has a priority of 99 and runs last.

`cue-front-extension-run-default` is the extension that runs the GraphQL query selected for the current page request. If and only if **`resolvedRemainingPath`** is empty, it runs the GraphQL query, returns the results and terminates the request. Otherwise, it passes the request forward to **`cue-front-extension-not-found`**, which returns a "page not found" message and terminates the request.

Assuming you added **`tutorial.js`** to the end of the extension list in **`recipe/package.json`**, then it will be executed as the last of all the unprioritized extensions, immediately before **`cue-front-extension-run-default`**.

A **`context`** object containing information about the request is passed forward through all the extensions. Extensions may add information to it, but should not remove or modify anything. In this way all extensions have the ability to pass information downstream.

Currently, however, **`tutorial.js`** does not add any information to the **`context`** object. Instead, if it is triggered, it terminates the request. This call to **`done()`**:

```
    done({entity: {"letters": items[1], "digits": parseInt(items[2])}});
```

terminates the recipe's processing of the request and supplies an object to return to the Cook. The content of the returned object's **`entity`** field becomes the Cook's response to the request. You can also add headers to the response by including a **`headers`** field in the returned object:

```
    done({entity: {"letters": items[1], "digits": parseInt(items[2])}, "headers":{test:
   "abc"});
```

Terminating the request in this way may sometimes be what you want your extension to do, but often you don't want to replace the standard response, you just want to pass some information forward in order to influence how the request is processed by following extensions. You can, for example, add GraphQL parameters to the **`context`** object. **`cue-front-extension-run-default`** will pass any GraphQL parameters it finds in the **`context`** object to the GraphQL queries it executes.

If you don't want your extension to terminate the request, then instead of passing an object to the **`done()`** function, simply pass the value **`true`**:

```
    done(true);
```

The request's **`context`** object will then be passed on to the next extension, including any additions you have made.

8.3.8 Setting GraphQL Parameters

To make your extension pass parameters to the GraphQL query selected for the current page request, your extension's **`exports.run`** function must:

- Add the parameter definitions to **`context.graphqlParams`**.
- Pass true to the **`done()`** function so that the request is not terminated.

Preceding extensions may already have added parameters to **`context.graphqlParams`**, so you must be careful to not overwrite them. Here is a new version of the **`exports.run`** function that passes the **`letters`** and **`digits`** values as GraphQL parameters:

```
exports.run = function(recipe, context, done) {
  const items = pattern.exec(context.resolvedRemainingPath);
  context.graphqlParams = context.graphqlParams || {};
  context.graphqlParams.letters = items[1];
  context.graphqlParams.digits = parseInt(items[2]);
  done(true);
};
```

If you now visit your publication's **`/sport/ab/05`** page you will see the standards Sports section page contents again, but the **`context`** object that is passed forward to following extensions now contains at least two GraphQL parameters: a string parameter called **`letters`** with the value **`ab`**, and an integer parameters called **`digits`** with the value **`5`**.

To be used in a GraphQL query, parameters must be declared at the start of the query:

```
query($letters: String!, $digits: Int) {
  ...
}
```

> The **`query`** keyword that identifies the root of a GraphQL query is often omitted, but you must specify it when declaring parameters.

Once the parameters are declared in this way, they can be used in the body of the query – for example, passed on to a data source as follows:

```
query($letters: String!, $digits: Int) {
  ...
   datasource(name: $letters, count: $digits) {
    displayId
  }
  ...
}
```

In this example, if the incoming request URL is **`http://localhost:8101/tomorrow-online/sport/ab/12`**, then the query will call a data source called **`ab`** with a **`count`** parameter of **`12`**. if the incoming request URL is **`http://localhost:8101/tomorrow-online/sport/cd/05`**, then the query will call a data source called **`cd`** with a **`count`** parameter of **`5`**.

8.3.8.1 Dealing With Non-null Parameters

Note that the **`$letters`** parameter type is specified as **`String!`** rather than **`String`**. The **`!`** character means that the parameter is not allowed to be null. **`$letters`** must be defined as **`String!`** because the place where we want to use it (the **`datasource()`** function's **`name`** parameter) is defined as a **`String!`**.

The **!** symbol can be used in combination with all GraphQL type declarations to indicate that null values are not allowed.

If a GraphQL query parameter is defined as not null, then a value must be supplied or the query will fail with a message like this:

```
Variable "$letters" of required type "String!" was not provided.
```

This presents a problem: if you make the above changes to your existing Sports page GraphQL query, then it will work fine when users visit the **sport/ab/12** page, but when users visit the **sport** page it will fail with the above error because the extension hasn't run. You can fix this problem as follows:

1. Change **exports.pattern** so that the extension is run for both the **sport/ab/12** page (and other similar URLs) and for the **sport** page:

   ```
   exports.pattern = '^$|^/?([a-z]{2})/([0-9]{2})$';
   ```

   (The **^$** at the beginning of the regular expression matches an empty string in the remaining path.)

2. Change the **exports.run** function so that **context.graphqlParams.letters** and **context.graphqlParams.digits** get set even if the remaining path is empty:

   ```
   context.graphqlParams.letters = items[1] || "ab";
   context.graphqlParams.digits = parseInt(items[2]) || 0;
   ```

Now the GraphQL **$letters** and **$digits** parameters will always be assigned values and the GraphQL query will therefore be executed. Here is what **tutorial.js** looks like after these changes:

```
exports.pattern = '^$|^/?([a-z]{2})/([0-9]{2})$';

const pattern = new RegExp(exports.pattern);

exports.constraint = function(context) {
  const resolution = context.request.resolution.entity;
  return resolution.section == 'sport' && resolution.context == 'sec';
};

exports.run = function(recipe, context, done) {
  const items = pattern.exec(context.resolvedRemainingPath);
  context.graphqlParams = context.graphqlParams || {};
  context.graphqlParams.letters = items[1] || "ab";
  context.graphqlParams.digits = parseInt(items[2]) || 0;
  done(true);
};
```

8.3.9 Providing an Alternate GraphQL Query

Instead of just passing parameters to the GraphQL query associated with the resolved request, you can make your extension replace it with a completely different query. This can be useful, for example, in situations where the URLs captured by your extension are used to power Ajax calls that only need to modify the displayed page.

The request's default GraphQL query is held in the **context.query** object. Your extension can therefore force a different query to be used by replacing this object (actually **replacing** it, not modifying it). For example:

```
const myQuery = {
  query : `query($letters: String!, $digits: String!) {resolution {
  section {
    my_text: _static(value:$letters)
    my_number: _static(value:$digits)
  }
}
}`
 };

exports.run = function(recipe, context, done) {
  const items = pattern.exec(context.resolvedRemainingPath);
  context.graphqlParams = context.graphqlParams || {};
  context.graphqlParams.letters = items[1] || "ab";
  context.graphqlParams.digits = items[2] || "0";
  context.query = myQuery;
  done(true);
};
```

If you do this, you then probably don't want your extension to capture requests to the main Sports page so the pattern should ignore the empty string again:

```
exports.pattern = '^/?([a-z]{2})/([0-9]{2})$';
```

Here is the complete modified extension:

```
const myQuery = {
  query : `query($letters: String!, $digits: String!) {resolution {
  section {
    my_text: _static(value:$letters)
    my_number: _static(value:$digits)
  }
}
}`
 };

exports.pattern = '^/?([a-z]{2})/([0-9]{2})$';

exports.constraint = function(context) {
  const resolution = context.request.resolution.entity;
  return resolution.section == 'sport' ;
};

exports.run = function(recipe, context, done) {
  const items = pattern.exec(context.resolvedRemainingPath);
  context.graphqlParams = context.graphqlParams || {};
  context.graphqlParams.letters = items[1] || "ab";
  context.graphqlParams.digits = items[2] || "0";
  context.query = myQuery;
  done(true);
};
```

Now, the normal Sports section query is run for **`http://localhost:8101/tomorrow-online/sport/`** requests, but the replacement query is run for URLs such as **`http://localhost:8101/tomorrow-online/sport/ab/12`** or **`http://localhost:8101/tomorrow-online/sport/cd/34`.**

8.3.10 Parsing Request Parameters

In some cases you may want your extension to extract information not from the request URL itself, but from parameters supplied with the URL such as **`.../?order=newest`**.

The Cook doesn't automatically parse these parameters for you, and they don't get passed forward in **`context.resolvedRemainingPath`**. They are, however, available in **`context.request.incoming.url`**. This property contains the full URL submitted to the Cook, which will look something like this:

```
/tomorrow-online/sport/?param1=value1&param2=value2
```

You can therefore use standard query parsing techniques to parse the string. As when parsing the URL itself you can use the **`exports.constraint`** function to guard against malicious or invalid data in the supplied parameters, only accepting parameters that match strictly specified conditions. The following function, for example, only accepts an **`order`** parameter that has a value of either **`newest`** or **`relevant`**. No other parameters are accepted.

```
exports.constraint = function(context) {
  const mark = context.request.incoming.url.indexOf("?");
  const params = (mark == -1) ? {} :
 querystring.parse(context.request.incoming.url.substring(mark + 1));
  if (order in params == false || [undefined, 'relevant',
 'newest'].indexOf(params.order) == -1) {
    return false;
  }
};
```

Bear in mind that the **`exports.constraint`** function runs for all requests and that its purpose is to determine whether or not the **`exports.run`** function should be executed. First, it grabs the request string and splits it at the **`?`** character (if present) into a URL path and a series of query parameters, which it parses and stores in the **`params`** object.

```
  const mark = context.request.incoming.url.indexOf("?");
  const params = (mark == -1) ? {} :
 querystring.parse(context.request.incoming.url.substring(mark + 1));
```

It then checks **`params.order`**. Unless it is undefined or contains one of the two allowed values, the function returns **`false`** and exits, ensuring that **`exports.run`** is not executed:

```
  if ([undefined, 'relevant', 'newest'].indexOf(params.order) == -1) { return false; }
```

If **`params.order`** is defined with a legal value, then it can be used in the same ways as values extracted from **`context.resolvedRemainingPath`**:

- Returned in the **`done`** object's **`entity`** field
- Added to the **`context`** object for use by following extensions

8.4 Upgrading Recipe Extensions

The starter publications provided by Stibo DX (such as **`tomorrow-online`** and **`starter-publication`**) include a default recipe, **`recipe/recipe.js`**. Most of the recipe functionality, however, is provided by recipe extensions. The recipe extensions provided by Stibo DX are published

on an NPM server, **npm.escenic.com** and are downloaded from **npm.escenic.com** the first time they are required (that is, the first time the recipe is executed).

Each extension published on **npm.escenic.com** is separately maintained and has its own version number. As corrections and improvements are made to an extension, new versions may be published. This will not, however, have any effect on your publications, since your recipe depends on a specific version of each extension, specified in the file **recipe/package.json**. If you upgrade your CUE Front installation, you will get new versions of the Cook, Cleaver and so on, but the recipe will not be upgraded. The recipe is part of **your** application: you may well have made changes to **recipe/package.json** (added extensions of your own, for example) that should not be overridden.

In order to upgrade recipe extensions, therefore, you must explicitly require new versions by editing **recipe/package.json** yourself. In order to be able to do this, you need to know which recipe extensions have been upgraded and what changes have been made. You can upgrade recipe extensions in two ways:

- As part of general CUE Front upgrades
- Between CUE Front releases

> Sometimes, upgrading a recipe extension will involve additional tasks. Before upgrading a recipe extension, therefore, you should always review the extension's **README** file (published on npm.escenic.com) and check for upgrade instructions.

8.4.1 Upgrading with CUE Front

All releases of CUE Front are accompanied by a set of release notes describing all significant changes in the new version. You can find a link to these release notes on the main documentation page for each version (here, for the current version). In most releases, some of the changes listed in the release notes will either be or include changes to one or more recipe extensions. In such cases, the change description will include the names and versions of the affected extensions. In order to apply these changes to your installation, therefore, you will need to edit your **recipe/package.json** file and increment the version numbers of the affected extensions as specified.

8.4.2 Upgrading Between CUE Front Releases

Recipe extension changes are in many cases published on **npm.escenic.com** as soon as they are ready and tested, and if they can be used independently of changes to other CUE Front components. It is therefore possible to take advantage of some error corrections and improvements without waiting for a new CUE Front release. Stibo DX does not send automatic notifications about upgrades to individual recipe extensions, but you can get information about available changes in the following ways:

From Stibo DX support
: You may be told by Stibo DX support that a bug you have reported can be fixed by upgrading an extension to a specific version, or that a feature you have requested is already implemented and can be obtained by upgrading an extension.

By querying the NPM server
: You can get information about recently published extension versions from **npm.escenic.com**, by entering the following command in the CUE Front installation folder (while the Cook is running):

```
docker-compose exec cook npm show @escenic/extension-name changes
```

where *extension-name* is the name of the extension you are interested in.

This will produce a list of the most recently published versions available in the repository, and for each version, a list of the changes introduced. For example:

```
# Changelog

Recent changes to @escenic/cue-front-graphql@v0.0.25

Changes between v0.0.14 and v0.0.25

 * DPRES-899: Fixed typo in accessing attributes of HTMLElement.
 * DPRES-806: Added support to remove empty publication feature properties.
 * DPRES-896: Fixed broken rich text fields
 * DPRES-832: Added support for primitive type array fields.
 * DPRES-832: Added support for complex field.
 * DPRES-896: Inline elements now resolve from the correct content item
 * DPRES-510: Only published inline relations are shown.
 * DPRES-831: Use label + description of content type
 * DPRES-831: Added comment for each field too.
 * DPRES-831: Added a comment to the generated schema
 * DPRES-831: Documented most of the schema
 * DPRES-832: fixed bugs in dealing with complex and arrays
 * DPRES-832: whitespace changes to make it easier to spot differences
```

8.4.2.1 Upgrade Procedure

Before you upgrade a recipe extension, you should visit **`npm.escenic.com`** in a browser, log in using your Stibo DX credentials and view the **`README`** file for the extension you intend to upgrade. This file will contain configuration instructions for the extension, including information about any configuration changes you may need to make for the new version.

To upgrade an extension:

1. Open **`recipe/package.json`** in a text editor.
2. Find the entry for the extension you want to upgrade. For example:

   ```
   "@escenic/cue-front-graphql": "0.0.14",
   ```

3. Replace the version number.
4. Save the file.
5. Make any required configuration changes.
6. Restart the Cook.

If for any reason the upgraded version of the extension does not work as required, you can go back to your old version by simply reverting the changes you have made and restarting the Cook again.

9 ESI Support

Edge Side Includes (ESI) is a widely-used standard for caching of web pages. The basic idea is that parts of a web page that do not get updated frequently can be identified using special ESI markup, and cached in servers close to clients.

CUE Front can support ESI if correctly configured, and the Tomorrow Online demo publication makes use of ESI for displaying page headers and footers. This section contains a brief description of the Tomorrow Online implementation.

Tomorrow Online's **`default-master.twig`** template contains the following lines:

```
{% if data.headerMenuESI %}
  <esi:include src="{{ data.headerMenuESI }}" onerror="continue"/>
{% elseif data.headerMenu %}
  {% include "organisms-header" with {"menu" : data.headerMenu} %}
{% endif %}
```

Here, the template sends an ESI tag instead of executing a header template if the data received from the Cook indicates that ESI is in use (that is, if it contains a **`headerMenuESI`** field. It replaces **`{{ data.headerMenuESI }}`** with the value of the field (**`.esi/header-menu`**).

The response passes through an ESI caching server on the way to the client. The ESI server replaces the **`esi:include`** tag with the actual header menu which it obtains either from its cache or by sending a request for the **`.esi/header-menu`** resource back to the Tomorrow Online server. After inserting the header menu, the caching server passes the completed page on to the client.

Any response received by the Cook for a URL starting with the characters **`.esi/`** is handled specially, as described in section 4.2.3. In the case of Tomorrow Online, this means that the response will be generated by the **`esi/header-menu.graphql`** query, which returns a header menu.

The mechanisms used to implement ESI here are very flexible, and can be modified to meet your needs. In particular, the rule that treats URLs starting with the characters **`.esi/`** in a special way is defined in a simple recipe configuration, and can easily be modified as described in section 8.1.3.

The Waiter incorporates a simple ESI parser for use during development, so that you do not need to include an ESI cache in your development environment. The parser is enabled by default. To disable it in a production system, replace

```
enableESIParser: true
```

with

```
enableESIParser: false
```

in your **`waiter-config.yaml`** file.

> The Waiter's ESI parser only supports the **`esi:include`** element, and does not support the use of unbalanced templates. What that means is that the HTML content replaced by an **`esi:include`** element must contain a balanced set of HTML start and end tags - you can't have the start of an element in one template and the end in another. This kind of templating is actually allowed by the ESI standard, but it is generally regarded as bad practice and is not handled by the Waiter's parser.

10 Cache Configuration

The Cook forwards incoming requests to the Content Store and passes responses back to the requester (usually the Waiter). The Content Store web service returns data in the form of Atom XML resources, which need to be parsed and converted to JSON objects before they can be returned to the Waiter. This parsing is relatively CPU-intensive. A significant proportion of the requests the Cook handles are repeat requests for the same resources; often resources that change relatively infrequently such as publications, sections, section pages, popular tags and so on.

In order to minimize repeat requests of this kind, the Cook incorporates a cache in which ready-parsed responses are stored for re-use where possible. The cache is an LRU (least recently used) cache with **`ETag`**-based validation. This means that:

- All responses are saved until the cache is full. Once the cache is full, space for new responses is created by throwing out the least-recently used old response.
- The Cook uses a response's **`ETag`** to determine whether or not a cached response can still be used. An **`ETag`** is a unique ID (such as a checksum) derived from the content of a response that the server (i.e the Content Store) includes in the response header. The Cook saves this **`ETag`** in the cache along with each response. If the cache contains a response for an incoming request, the Cook includes that response's **`ETag`** in the request it forwards to the Content Store. The Content Store checks this **`ETag`** against the **`ETag`** of the current content, and if they are identical, returns an empty response indicating that the requested content has not changed. The Cook can then return the cached response. If they are not identical, a full response is returned and used to replace the old cached response.

You can control the details of how the cache behaves by adding settings to the Cook's configuration file, **`cook-config.yaml`**. All the settings must be added as children of the **`shared-cache`** property. The mostly commonly used settings are:

`max`
: The maximum size of the cache. Once this limit it reached, an old response is thrown out every time a new response is added. The default setting is **`5000`** responses.

`maxAge`
: The maximum allowed age for cache contents, specified in milliseconds. A cached response which is older than the specified age will initially be regarded as out-of-date or **stale**. What happens to a stale response is determined by how the **`stale`** property is set.

`stale`
: Determines whether or not **`ETag`** validation is performed. The possible values are:

  `true` **(default)**
  : **`ETag`** validation **is** performed for stale responses. The Cook forwards the request to the Content Store along with the cached **`ETag`**, and if the Content Store says the **`ETag`** is still valid, the cached response is returned to the waiter. If the **`ETag`** is not valid then the new response returned from the Content Store is passed back to the Waiter and used to update the cache.

  `false`
  : **`ETag`** validation **is not** performed for stale responses. The request is forwarded to the Content Store without an **`ETag`**. The response is returned to the Waiter and used to update the cache.

The following examples show how the **maxAge** and **stale** properties work together:

maxAge=0, stale=false
: Caching is disabled.

maxAge=0, stale=true
: **ETag** validation is performed on all cached responses.

maxAge=2000, stale=false
: Cached responses under 2 seconds old are used without validation, older responses are not used at all.

maxAge=2000, stale=true
: Cached responses under 2 seconds old are used without validation, **ETag** validation is performed on all older responses.

Here is an example cache configuration:

```
shared-cache:
  max: 10000 # default: 5000
  maxAge: 5000 # 5 seconds
  stale: false # default: true
```

The cache does have additional settings that you might find useful in some circumstances. For full information, see here. Note, however, that the Cook only supports the use of properties with simple values. Properties such as **length** and **dispose** that require functions as values are not supported.

For general information on how to add configuration properties to **cook-config.yaml** so that they will not be overridden by CUE Front upgrades, see section 13.6.

10.1 Layout-sensitive caching

The Cook's built-in cache provides content-only caching. A rendered web page, however, consists of more than just the content. It also includes a lot of HTML, CSS and Javascript code, boilerplate content and so on. This layout-related content is added by the Waiter, and may also need to be cached. Many CUE Front installations therefore include a caching server such as Varnish placed in front of the Waiter.

The **ETag**s generated by Content Store are not sufficient for such a configuration to work effectively, since they only change when the content changes, and take no account of changes to a publication's Twig templates, CSS and Javascript files. You can, however, make this caching layer sensitive to layout changes by adding an **etagSuffix** attribute to your Waiter configuration file, **waiter-config.yaml**.

This attribute, as its name suggests, defines a suffix that the Waiter appends to the **ETag** of every response it receives from the Cook before passing the response on to its client. The idea is that you configure your application deployment procedure to modify the **etagSuffix** every time changes are deployed. You could, for example, use a build number or a time stamp as an **etagSuffix** value. This means that the **ETag**s seen by an external caching server will change not only when a page's content changes, but also when its layout may have changed, ensuring that out-of-date layouts are never served to the end user. Exactly what strategy you choose for updating the **etagSuffix** will depend on your requirements and your development procedures.

Since **`etagSuffix`** must be explicitly set each time your application is deployed (or at least each time a significant layout change has been made), there may be no risk of it being overridden by CUE Front upgrades. If, however, your chosen **`etagSuffix`** update strategy means this is still a risk, see section 13.6 for general information on how to avoid it.

> If you find that your web site is not reflecting layout changes even after introducing an **`etagSuffix`** update strategy, a possible reason may be that the Waiter's Twig cache contains some old templates, and needs to be cleared. This should not in general be a problem.

11 Diagnostics and Monitoring

This chapter contains descriptions of diagnostic and monitoring tools supplied with CUE Front.

11.1 Cook Diagnostic Resources

The Cook provides some useful diagnostic information under the **/-** URL. If you open a browser and go to *your-site*:**8101/-** you will see a list of links to the following resources:

Incoming requests
This resource is located at *your-site*:**8101/-/incoming**. It displays a constantly updated list of the requests currently being handled by the Cook. For each request the following information is displayed:

- The number of seconds the request has been running
- The path of the request (**/tomorrow-online/tag/ramsey**, for example)
- The request's **X-Request-ID**

For example:

```
1.243     /tomorrow-online/tag/ramsey     e59431a6-a44c-11e8-8957-67f31fc15fef
```

The list is updated every 3 seconds by default. You can change the refresh interval by editing the page's URL. To change the interval to 5 seconds, for example enter:

```
your-site:8101/-/incoming?refresh=5
```

Outgoing requests
This resource is located at *your-site*:**8101/-/outgoing**. It displays a constantly updated list of outgoing requests made by the Cook that have not yet completed. For each request the following information is displayed:

- The number of seconds the request has been running
- The request URL (**http://solr.example.com:8983/solr/presentation/select?id**, for example)
- The request's **X-Request-ID**

For example:

```
1.243     http://solr.example.com:8983/solr/presentation/select?id     e59431a6-
a44c-11e8-8957-67f31fc15fef
```

The list is updated every 3 seconds by default. You can change the refresh interval by editing the page's URL. To change the interval to 5 seconds, for example enter:

```
your-site:8101/-/outgoing?refresh=5
```

Incoming request log
This resource is located at *your-site*:**8101/-/incoming-log**. It displays a list of recently completed incoming requests. For each request the following information is displayed:

- The time at which the request was made

- The number of seconds the request took to complete
- The path of the request (**`/tomorrow-online/tag/ramsey`**, for example)
- The type of response given (HTTP response code)
- The request's **`X-Request-ID`**

For example:

```
2018-10-30T13:28:06.073Z   1.195    /tomorrow-online/  200  f292fae7-
adf0-4387-981c-2943dd708f9b
```

By default, the log shows the last 50 incoming requests. You can modify this by specifying a parameter in **`cook-config.yml`**. To increase the number to 100, for example:

```
log-incoming-count: 100
```

Outgoing request log

This resource is located at *your-site*`:8101/-/outgoing-log`. It displays a list of recently completed outgoing requests. For each request the following information is displayed:

- The time at which the request was made
- The number of seconds the request took to complete
- The request URL (**`http://solr.example.com:8983/solr/presentation/select?id`**, for example)
- The type of response received (HTTP response code)
- The request's **`X-Request-ID`**

For example:

```
2018-10-30T15:56:38.448Z 0.197    http://to.example.com:8080/webservice/escenic/
section/8 200 1b829a02-3e75-43d4-b9d2-1f2c4e542566
```

By default, the log shows the last 100 outgoing requests. You can modify this by specifying a parameter in **`cook-config.yml`**. To increase the number to 200, for example:

```
log-outgoing-count: 200
```

The **`X-Request-ID`** associated with an incoming request is passed on by the Cook to all associated outgoing requests, making it easy to see which requests in the log belong together.

11.2 Monitoring Log Messages

The easiest way to monitor log messages output by the CUE Front services is to use a web-based log management tool. There are a number of commercial log management services available, such Loggly. **Dozzle**, however, is a freely available open source solution that works well together with CUE Front. It is itself available as a Docker container, making it very easy to install.

To install dozzle, enter:

```
docker pull amir20/dozzle:latest
```

Then to start it, enter:

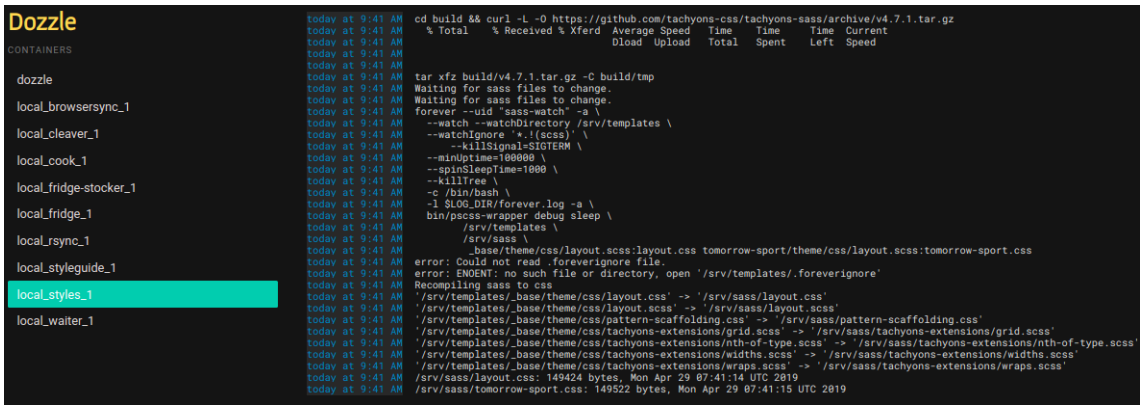
```
docker run --name dozzle -d --volume=/var/run/docker.sock:/var/run/docker.sock -p
 8888:8080 amir20/dozzle:latest
```

You will then be able to view the logs of all Docker containers running on your machine at **`localhost:8888`**. If you want to use a different port, just modify the command accordingly.

It is, of course possible to view the same log messages by simply entering:

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

in a terminal window, but Dozzle output is much more useful because it provides a separate log for each container:



To switch between logs, simply select from the menu on the left.

12 Advanced Cleaver Features

The Cleaver provides basic image management functionality out of the box, and for many uses does not require any special configuration. It does, however offer additional features that can be enabled if required:

- Cloud-based image caching
- Image Filters

12.1 Cloud-based Image Caching

By default the Cleaver retains copies of all cropped images in a local cache. When an image is requested, it looks for the image in this cache first. If the requested image is not present in the cache, then it downloads the original image from the Content Store, applies the requested crop, returns the requested image and saves it in the cache for next time. This mechanism works well for a single-Cleaver installation with unlimited disk space, but can run into the following problems:

- The cache may need to be pruned periodically due to disk space limitations by running a **`cron`** job to remove infrequently requested images. This means, however, that the deleted images will need to be re-cropped the next time they are requested. Scrapers and indexers making many requests for old images can then result in load spikes.
- Adding extra Cleaver instances to improve performance can actually have the opposite effect initially, since each new instance will start up with an empty cache that needs filling. It is in general inefficient for each Cleaver to maintain its own cache, since each Cleaver will crop its own version of every requested image, even if another Cleaver has already done the same job.

You can solve both these performance problems by configuring the Cleaver(s) to make use of a cloud-based secondary cache. If you do this, then when a cropped image is saved to the local cache, it is also saved to the cloud cache, which is shared between all Cleaver instances. When a request is received by a Cleaver, it will first check its local cache, and then if necessary, the cloud cache. Only if the image is not found there will the Cleaver request the original image from the Content Store and crop it. Ideally, the cloud cache should never be pruned, so that almost all image crops are cached, even those that are used very infrequently.

Currently the Cleaver only supports the use of Amazon S3 buckets as image caches.

12.1.1 S3 Cache Configuration

To make use of this feature you need to set up an Amazon S3 account. Once you have done that you can enable the cloud caching feature by running the **`setup`** tool in advanced mode. The Cleaver-related prompts displayed by **`setup`** will then include an option to enable S3 storage, plus a number of configuration parameters:

Enable AWS S3 storage support?
Select **`true`**.

AWS S3 bucket to use?
Enter the name of S3 bucket to be used for image storage

Subdirectory(or subdirectories) under the bucket root to use for storing images?
Enter the path of the folder to be used for image storage. You might choose a path based on your publication name – **`tomorrow-online`** or **`publication/tomorrow-online`**, for example. If you do not specify a path, then the images are stored directly in the bucket's root folder.

Configure credentials using setup tool?
Select **`true`** to enter your S3 credentials here in **`setup`**. You might want to do this directly in the S3 configuration interface, since doing so is more secure and the S3 interface offers more configuration options. If you want to use S3 to configure Cleaver access, see here for details of how to do it.

S3 access key?
Enter your S3 access key. This prompt is only displayed if you have selected **`setup`**-based S3 configuration.

S3 secret key?
Enter your S3 secret key. This prompt is only displayed if you have selected **`setup`**-based S3 configuration.

S3 region name?
Enter the name of S3 region you are using. This prompt is only displayed if you have selected **`setup`**-based S3 configuration.

Answering all the above prompts will result in a Cleaver configuration that looks something like this:

```
servers:
  -
    host: 'engine:8080'
    username: tomorrow-online_admin
    password: admin
log-file: 'stdout:'
download-dir: /var/cache/cleaver/
listen: '0.0.0.0:8102'
retry_download_count: 0
debug: true
s3:
  enabled: true
  bucket_name: "your-bucket-name"
  key_prefix: "your-image-cache-path" # example: 'tomororw-online', 'publication/
tomorrow-online'
  aws_access_key_id: your-access-key
  aws_secret_access_key: 'your-secret-key'
  region_name: 'eu-west-1'
```

You can manually add further settings under the **`s3:`** entry if required. These settings all correspond to S3 configuration parameters that can also be set via the S3 configuration interface:

```
use_ssl: False

 # Supported upload arguments.
 # For details: https://boto3.amazonaws.com/v1/documentation/api/latest/reference/
customizations/s3.html#boto3.s3.transfer.S3Transfer.ALLOWED_UPLOAD_ARGS
 upload_args:
   ServerSideEncryption: "AES256"

 # Supported download arguments.
 # For details: https://boto3.amazonaws.com/v1/documentation/api/latest/reference/
customizations/s3.html#boto3.s3.transfer.S3Transfer.ALLOWED_DOWNLOAD_ARGS
 download_args:
```

```
  # How many uploads to perform concurrently
  upload_concurrency: 10 # default: 5

  # Supported transfer config for downloading.
  # For details: https://boto3.amazonaws.com/v1/documentation/api/latest/reference/
customizations/s3.html#boto3.s3.transfer.TransferConfig
  download_conf:

  # S3 library specif debug logs
  debug: true

  # S3 Library specific debug log format.
  # For details: https://docs.python.org/3/library/logging.html#logrecord-attributes
  debug_log_format: "%(name)s - %(thread)d - %(threadName)s - %(levelname)s -
  %(message)s"
```

`use_ssl`: Whether or not to use HTTPS for communication with S3 (default is false)
`upload_args`: See here for details.
`download_args`: See here for details.
`upload_concurrency`: Maximum number of concurrent uploads to allow (default is 5)
`download_conf`: See here for details.
`debug`:
`debug_log_format`: See here for details.

12.2 Image Filters

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

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

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

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

12.2.1 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.

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.

13 The Setup Tool

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

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

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

generate
: Generates a set of configuration files by merging the responses in **cue-front/***configuration-name***/blueprint.yaml** with the default configuration files in the **cue-front/setup/defaults** folder. The generated configuration files are saved together with **blueprint.yaml** in the configuration folder.

setup has two corresponding subcommands that execute these phases:

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

generate
: Executes the **generate** phase for a named configuration set.

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

login
: Prompts for the credentials needed to download software from Stibo DX's repositories.

edit
: Lets the user modify an existing configuration set.

list
: Lists the names of all available configuration sets.

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

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

13.1 Initializing Setup

In order to be able to function, **setup** needs access to the Stibo DX repositories. Before you do anything else, therefore, you need to supply your login credentials. The setup uses these credentials to acquire and store an authentication token it can use when downloading software from the repositories.

To initialize **setup**, use the **login** subcommand as follows:

```
cd cue-front-path/setup
docker-compose run setup login username
```

where *username* is the account name you use to access Stibo DX software repositories.

setup displays the following prompts:

Password for *username*
Enter your password.

13.2 Creating a New Configuration

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

```
cd cue-front-path/setup
docker-compose run setup add configuration-set
```

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

setup displays the following prompts:

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

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

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

Advanced
All configuration settings are displayed.

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

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

1. Saves your responses in *path*/**cue-front/setup/***configuration-set*/**blueprint.yaml**.
2. Generates a set of configuration files from the blueprint and saves them in the same folder.

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

```
cd cue-front-path/configuration-set
docker-compose up -d
```

13.3 Regenerating a Configuration

The easy way to modify a configuration is to use **`setup edit`** (see section 13.5). You can, however, also do it by manually editing a configuration's **`blueprint.yaml`** file and then regenerating the configuration from the blueprint as follows:

```
cd cue-front-path/setup
docker-compose run setup generate configuration-set
```

13.4 Switching Configurations

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

1. Stop and remove the current set of CUE Front containers:

   ```
   cd cue-front-path/old-configuration-set
   docker-compose down
   ```

2. Restart specifying a different configuration set:

   ```
   cd cue-front-path/new-configuration-set
   docker-compose up -d
   ```

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

13.5 Modifying a Configuration

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

```
cd cue-front-path/setup
docker-compose run setup edit configuration-set
```

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

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

13.6 Overriding Setup Defaults

The values prompted for when you add a new configuration, and the default values offered by **`setup`** are all defined in the **`setup/defaults`** folder. This folder contains a set of default configuration files (**`cook-config.yaml`**, **`fridge-config.yaml`**, **`cleaver-config.yaml`**, **`docker-compose.yml`** and **`waiter-config.yaml`**) plus a **`blueprint.yaml`** file that defines:

- The values **`setup add`** is to prompt for
- The prompt texts to display
- Defaults for the prompted values

When you create a CUE Front configuration using the **`setup add`** *myconfig* command, the setup utility:

1. Displays the sequence of prompts defined in CUE Front's **`setup/defaults/blueprint.yaml`** file.
2. Creates a **`cue-front/myconfig`** folder and saves your responses in **`cue-front/myconfig/blueprint.yaml`**.
3. Merges the values in **`cue-front/myconfig/blueprint.yaml`** with the default configuration files in the **`cue-front/setup/defaults`** folder
4. Saves the merged configuration files in the **`cue-front/myconfig/`** folder together with **`blueprint.yaml`**.

If you wanted to change the setup defaults, you could do so by modifying these files. You could, for example, change the default value for one of the setup prompts by editing **`setup/defaults/blueprint.yaml`**, or you could change the content of all the **`cook-config.yaml`** files generated by the setup tool by editing **`setup/defaults/cook-config.yaml`**. However, any changes you made in this way would be overwritten the next time CUE Front was upgraded.

The correct way to override the supplied defaults, therefore, is to create a copy of the **`cue-front/setup/defaults`** folder in your publication repo, and edit the copied files. When **`setup add`** is run, it will actually check your publication repo first, and if it finds any setup defaults, use those instead of the defaults supplied with CUE Front.

In detail, do as follows to create your own customized set of defaults:

1. Copy the setup defaults folder from the CUE Front folder to your publication repo:

   ```
   mkdir publication-path/setup
   cp cue-front-path/setup/defaults/ publication-path/setup/
   ```

2. Delete any copied files that you don't intend to modify, for example:

   ```
   cd publication-path/setup/defaults/
   rm cook-config.yaml fridge-config.yaml cleaver-config.yaml docker-compose.yml
   ```

   > You should always delete **`docker-compose.yml`** from your publication **`setup/defaults`** folder. Currently, it is not possible to customize the default **`docker-compose.yml`** file.

3. Edit the files to meet your needs.

When editing actual configuration files in your override folder (i.e, **`waiter-config.yaml`** in this case), simply make the changes you need. When editing **`blueprint.yaml`**, however, you should edit the entries you want to change and **delete all the other entries**. In this way, only the settings you are actually interested in will be overridden. If, for example, you want to change the default value of the Waiter's **`devmode`** property to **`false`**, then you can do so by creating a **`blueprint.yaml`** file that contains just the following lines:

```
waiter:
  devmode:
    advanced: true
    default: false
```

If you want to prevent setup from prompting for a value rather than just modify the default, you can do so by adding the following property:

```
    condition: "false"
```

to its definition. For example:

```
waiter:
  devmode:
    advanced: true
    default: false
    condition: "false"
```

Note that the **condition** property must be a string, **"false"**, not a boolean value.

If you want to remove a property completely from the output **blueprint.yaml** file generated by **setup add**, then you can do so by setting the default value to nothing and adding **condition: "false"**. The following, for example, completely removes the Waiter's **publications-name** property from the output **blueprint.yaml** file:

```
waiter:
  publications-name:
    message: "Publication-name"
    default: ""
    condition: "false"
```

For an example of why you might need to be able to do this, see section 13.7.

13.7 Multi-publication Support

The prompts displayed by **setup** do not offer the option of creating more than one publication. CUE Front, however, is designed to support multiple publications. Once you have used the **setup** utility to get up and running with a single publication, you can switch to a multi-publication configuration by overriding the **setup** defaults as described in section 13.6.

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

1. Make a copy of the setup defaults folder in your publication repo, and delete everything except **blueprint.yaml** and **waiter-config.yaml**.
2. Add the multiple publication settings to *your-publication***/setup/defaults/waiter-config.yaml** by manually editing the file.
3. Remove the publication prompt definitions from *your-publication***/defaults/blueprint.yaml** by manually editing the file.
4. Run **setup add** to generate a new configuration containing the multiple publication definitions.
5. Add information about your new publications to the **nginx** configuration file, **waiter/docker/nginx.conf**.
6. Restart the Waiter and **nginx**.

These steps are described in greater detail in the following sections.

13.7.1 Copy Setup Defaults

Copy **setup/defaults/waiter-config.yml** folder from the CUE Front folder to a **setup** folder in your publication repo:

```
mkdir publication-path/setup
cp cue-front-path/setup/defaults/waiter-config.yml publication-path/setup/
```

13.7.2 Add multiple publication settings

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

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

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

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

The properties should be set as follows:

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

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

`templateDir`
The folder containing the publication templates. For more about this, see section 4.4.

In the example shown above:

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

13.7.3 Remove publication prompt definitions

Open *cue-front-path*`/setup/defaults/blueprint.yml` in an editor and remove the following lines:

```
publications-name:
  message: "Publication-name"
  default: "tomorrow-online"

publications-hostNames:
  message: "Publication-hostname"
  default: "localhost"
```

This will prevent the **`setup add`** command from displaying these prompts.

13.7.4 Generate a new configuration

To create a new configuration run **`setup add`**:

```
cd cue-front-path/setup/
docker-compose run setup add configuration-set
```

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

If you want to modify an existing configuration, run **`setup edit`** instead.

13.7.5 Reconfigure nginx

Copy *cue-front-path*`/service/waiter/docker/nginx.conf` to *publication-path*`/service/waiter/docker/`:

```
cp cue-front-path/service/waiter/docker/nginx.conf publication-path/service/waiter/
docker/
```

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

```
server {
  listen       8100;
  server_name  localhost;

  root /srv/templates/_base;

  location / {
    try_files $uri @waiter;
  }

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

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

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

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

  root /srv/templates/myotherpub;

  location / {
    try_files $uri @waiter;
  }

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

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

13.7.6 Restart the Waiter

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

```
docker-compose restart waiter
```

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

14 Setting up Tomorrow Sport

This chapter tells you how to:

- Set up Tomorrow Sport as a sister publication to Tomorrow Online
- Enable cross-publishing between the two publications

The **tomorrow-online-1.13.9-3.zip** package contains everything you need to set up a sister publication called Tomorrow Sport alongside Tomorrow Online. This publication mostly uses the same templates as Tomorrow Online, with just a few overlay templates to distinguish it from the main publication. The package also contains everything needed to allow Tomorrow Online to cross-publish content from Tomorrow Sport.

14.1 Create Tomorrow Sport

> The procedure described in this section is a short cut to setting up Tomorrow Sport as a sister publication to Tomorrow Online. For a general description of how to serve multiple publications based on a single publication definition, see section 13.7.

To create and publish Tomorrow Sport:

1. Create a new publication in the Content Store called **tomorrow-sport**, using the publication definition called **tomorrow-sport-with-content.zip** which you will find in the **tomorrow-online/publication/dist** folder. This file is identical to the **tomorrow-online-with-content.zip** file that you used to create Tomorrow Online, except for its content, which is exclusively sport-related. For instructions on how to create new publications in the Content Store, see http://docs.escenic.com/ece-install-guide/7.1/create_a_publication.html.
2. Give your CUE Front user (that is, the user the Cook uses to log in to the Content Store) a minimum of read access to the new publication. If you just accepted defaults when installing CUE Front, then the Tomorrow Online admin user (**tomorrow-online_admin**) is your CUE Front user. If you also use this user for working in CUE, then you will probably want to give this user read/write access to Tomorrow Sport. If you use a different user for working in CUE, then you can just give read access to the CUE Front user, and read/write access to your editing user. For general information on how to create and manage users and user access rights, see the CUE Content Store Publication Administrator Guide.
3. Copy **waiter-config.yml** from your *publication-path*/**contrib/tomorrow-sport/** folder to *publication-path*/**setup/defaults/**. For example:

   ```
   mkdir -p publication-path/setup/defaults
   cp publication-path/contrib/tomorrow-sport/waiter-config.yml publication-path/
   setup/defaults/
   ```

   The contents of **waiter-config.yml** are based on the assumption that your publications are called **tomorrow-online** and **tomorrow-sport**. If this is not the case, then you will need to open the file and edit it.
4. Open the **blueprint.yml** file in your config folder (*cue-front-path*/**myconfig/blueprint.yml**, for example) in an editor, and remove the following lines:

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

```
    publications-hostNames: localhost
```

From the **`waiter:`** section of the file.

5. Copy **`nginx.conf`** from the *publication-path*`/contrib/tomorrow-sport/` folder to *publication-path*`/service/waiter/docker/`. For example:

```
cp publication-path/contrib/tomorrow-sport/nginx.conf publication-path/service/
waiter/docker/nginx.conf
```

6. Open the copied file for editing and change this line:

```
        server_name  sister.publication.hostname;
```

Replace **`sister.publication.hostname`** with whatever host name you want to use for the Tomorrow Sport site. For example:

```
        server_name  tomorrow-sport;
```

7. Remake your config by running **`setup generate`**. For example:

```
cd cue-front-path/setup
docker-compose run setup generate myconfig
```

8. Open your **`hosts`** file (**`/etc/hosts`** on MacOS and Linux, **`c:\Windows\System32\Drivers\etc\hosts`** on Windows), for editing and add your selected host name as an alias for the IP address **`127.0.0.0`**. For example:

```
127.0.0.1 localhost tomorrow-sport
```

9. Rebuild and restart the Waiter:

```
docker-compose build waiter
docker-compose restart waiter
```

You should now be able to access Tomorrow Online on **`http://localhost:8100`** as before, and Tomorrow Sport on **`http://tomorrow-sport:8100`**. Tomorrow Sport should have different content and a different appearance than Tomorrow Online.

If you now overwrite your **`myconfig`** setup by running **`setup add`** then you will break this configuration. To prevent the risk of this occurring, you can open the default **`blueprint.yml`** file (*cue-front-path*`/setup/defaults/blueprint.yml`) in an editor and remove these lines:

```
publications-name:
  message: "Publication-name"
  default: "tomorrow-online"

publications-hostNames:
  message: "Publication-hostname"
  default: "localhost"
```

from the **`waiter:`** section of the file. This will prevent **`setup add`** from prompting for a publication name and host name.

14.2 Cross-Publishing from Tomorrow Sport

If you have set up your user permissions correctly as described in step 2 of section 14.1, then you should now be able to cross-publish content from Tomorrow Sport to Tomorrow Online. That is, when editing Tomorrow Online in CUE, you should be able to find content belonging to Tomorrow Sport

and desk it in Tomorrow Online. It will then appear on the Tomorrow Online website as if it were Tomorrow Online content.

The Tomorrow Online front page has a "Latest in Sports" section at the bottom that by default displays teasers for content in the Tomorrow Online Sport section. This content is selected by a data source called **`latest_sport.graphql`**, which you can find in the *publication-path*`/recipe/datasources` folder:

```
{
  and {
    publication
    field(name:"home_section_name", value:"Sport")
    or {
      type(names: ["story","legacystory"])
    }
  }
}
```

The *publication-path*`/recipe/datasources` folder also contains an alternative version of this data source called **`latest_sport_tomorrow_sport.graphql`**, which selects content from Tomorrow Sport instead of from the Tomorrow Online Sport section:

```
{
  and {
    publication(name: "tomorrow-sport")
    field(name:"home_section_name", value:"News")
    or {
      type(names: ["story","legacystory"])
    }
  }
}
```

You can therefore switch from displaying local sports content to cross-published sports content on the front page by making the following change to line 2 of the the GraphQL query that assembles content for the Tomorrow Online front page (*publication-path*`/recipe/queries/index-page-ece_frontpage.graphql`):

```
  latest_sport: datasource(name:"latest_sport_tomorrow_sport") {
```

15 Publication Extensions

A publication extension is a bundle of functionality that can be added to any standard CUE Front publication. It typically consists of one or more new content types plus everything needed for CUE Front to render them: recipe extensions, GraphQL code, datasources, Twig templates and Waiter extensions. In order to ensure that it can be easily applied to any publication, an extension must be organized in a standard folder structure and packaged in a zip file.

Extending a publication with a correctly packaged extension doesn't involve much more than unzipping the extension into the publication folder. It is, however, also necessary to be careful with the naming of files: if an extension file has the same name and location in the structure as a file in the original publication, then it will overwrite it, so you should take care to avoid this. It is also the case that sometimes you may need to manually edit some publication files in order to make an extension work properly. This is typically the case with GraphQL code, since CUE Front does not provide any extension hooks or plug-in mechanism for GraphQL code.

15.1 Publication Extension Structure

A publication extension must be a zip file that adheres to the following structure:

publication/
  escenic/
    content-type/
      content-types
recipe/
  datasources/
    data-sources
  queries/
    graphql-queries
  extensions/
    recipe-extensions
templates/
  _annotations
  -data
  _layouts
  _meta
  _patterns/
  _twig-components
  placeholders
  theme
setup/
  defaults/
     setup-defaults
service/
  waiter/
    waiter-extensions/
      Extensions/
        waiter-extensions

where:

content-types
Are one or more modular content type resource files containing definitions of content types you want to add to a publication. For information about modular content type files, see The content-type Resource.

data-sources
Are one or more GraphQL files containing data source definitions you want to add to a publication. See chapter 7 for more information about data sources.

graphql-queries
Are one or more GraphQL files containing GraphQL queries you want to add to a publication. See section 4.2 for more information about GraphQL.

recipe-extensions
Are one or more Javascript files containing recipe extensions you want to add to a publication. See chapter 8 for more information about recipe extensions.

setup-defaults
Is a YAML file containing setup defaults you want to add to a publication. Any settings in this file will override the publication's existing settings. See section 13.6 for more information about setup defaults.

waiter-extensions
Are one or more PHP files containing waiter extensions you want to add to a publication. See chapter 5.

The **`templates`** folder must be a standard publication template folder containing the template patterns, layouts, CSS, Javascript and so on that you want to add to a publication.

15.2 Applying a Publication Extension

To apply a publication extension to a publication:

1. Copy the extension **`.zip`** file into the root folder of the publication. For example:
   ```
   cp myextension.zip ~/publications/mypublication/
   ```
2. Unzip the extension. For example:
   ```
   cd ~/publications/mypublication/
   unzip myextension.zip
   ```
3. Make any additional adjustments that are necessary (editing GraphQL queries, for example).
4. If the extension includes any new content types, then you will need to update the publication's GraphQL schema. For details of how to do this, see section 4.1.