Puli Documentation

Release 1.0.0-beta4

Bernhard Schussek

August 10, 2016

CHAPTER 1

Installation

1.1 Requirements

Puli requires PHP 5.3.9 or higher.

1.2 Installing the Puli CLI

To use Puli, you first need to install the Puli Command Line Interface (CLI). This is usually done only once on a system.

You can install the Puli CLI in one of three ways:

1.2.1 As a Phar (Recommended)

Download the `puli.phar` from the latest release page on GitHub. You can test whether the Phar works by running:

```
$ php puli.phar
```

Note: The leading `$` in the code examples is a convention that tells you that the example contains shell code. Type the command that follows the dollar sign into your terminal, but don't type the `$` itself.

You can either place the Phar in your project directory or somewhere in your path (such as `/usr/local/bin`) and chmod it to `755`. You can rename it to just `puli` to avoid typing `puli.phar` for every command.

The Phar can be updated with the `self-update` command:

```
$ php puli.phar self-update
```

Note: For this command to work you need PHP 5.6 or higher.

1.2.2 As a Global Composer Dependency

If you prefer to install tools like PHPUnit as global Composer dependencies, you can do the same for Puli. Install Composer and enter the following command in a terminal:

```
$ composer global require puli/cli:^1.0
```

1.2.3 As a Composer Dependency

If you want to explicitly document which version of the Puli CLI is required by your project, you can add it to your Composer dependencies. Install Composer and enter the following command in a terminal:

```
$ composer require --dev puli/cli:^1.0
```

1.2.4 OS X Only: Using Homebrew

If you are on OS X, you can also install Puli through Homebrew:

```
$ brew install puli
```

Note: This command requires Homebrew PHP to be installed.

1.2.5 Unix Only: Disable Glob Expansion

By default, Unix shells like Bash expand glob arguments before passing them to the called command. Look at this short example for a demonstration:

```
# What you type
$ command *.js

# What the command receives by the shell
$ command script1.js script2.js ...
```

If you use Puli on a Unix system, you should disable glob expansion for the `puli` command. If you use Bash, add the following lines to `~/.bashrc`:

```
# Disable glob expansion for Puli
alias puli='set -f;puli';puli(){ command puli "$@";set +f;}
```

Apply the changes with the `source` command:

```
$ source ~/.bashrc
```

If you use a different shell than Bash, see this answer on StackOverflow for instructions.

1.3 Next Steps

Read The Repository Component to learn how to register your resources with Puli.

CHAPTER 2

Glossary

Binding Assigns one or more resources to a binding type.

Binding Parameter Additional, named data stored with a binding. Needs to be defined by the binding type. Can be optional or required.

Binding Type A name for a semantic type of resources, like "thor/translations". Should start with the vendor namespace of the package that defines the name.

Child Resource A resource nested in another resource.

Package A logical grouping of files in a directory. Typically installed with Composer.

Path Mapping Maps a path prefix to a file or directory in a Composer package.

Path Prefix A prefix of a Puli path, like `/batman/blog`.

Public Resource A resource that is published in the document root of a web server and accessible in the internet.

Puli-Enabled Package A package that contains a `puli.json` file in its root directory.

Puli Path The path of a resource in the resource repository, like `/batman/blog/views/index.html.twig`.

Resource A file or directory, typically XML, YAML, HTML, CSS, an image etc.

Resource Consumer A package that defines a binding type and uses resources bound to this type.

Resource Provider A package that contains resources bound to a binding type of a resource Consumer.

Root Package The package for the current project. Any Puli project has exactly one root package and any number of package dependencies (non-root packages).

URL Format A printf string used to generate URLs of public resources, like `https://example.com/%s`.

UUID Short for Universally Unique Identifier. A random 128-bit value used to identify bindings and other distributed objects managed by Puli.

Web Server Represents a physical web server in Puli. Typically has a URL format and a document root.

CHAPTER 3

The Repository Component

Puli's Repository Component provides a framework-agnostic naming convention for accessing resources in your project and your installed Composer packages. *Resources*, in Puli, are all files that are not PHP, such as XML, YAML, CSS, JavaScript, images and so on.

3.1 The Problem

At the moment, you are probably accessing files through the `__DIR__` constant:

```
echo file_get_contents(__DIR__.'/../../res/views/index.html.twig');
```

Paths constructed like this become very long. Also, these paths break when you move the PHP file and make refactoring harder.

Many frameworks solve this problem by looking for files in a fixed directory:

```
// res/views/index.html.twig
echo load_file('views/index/html.twig');
```

This solution works well for simple applications, but breaks if you want to load resources from *different* resource directories, like your installed Composer packages. Frameworks solve this problem by supporting aliases for the resource directories of different packages:

```
// vendor/acme/blog/res/views/index.html.twig
echo load_file('acme-blog:views/index/html.twig');
```

Unfortunately, every framework has a different naming convention for these aliases. Puli solves this by introducing a universal naming convention that works with any framework – and even without.

3.2 How It Works

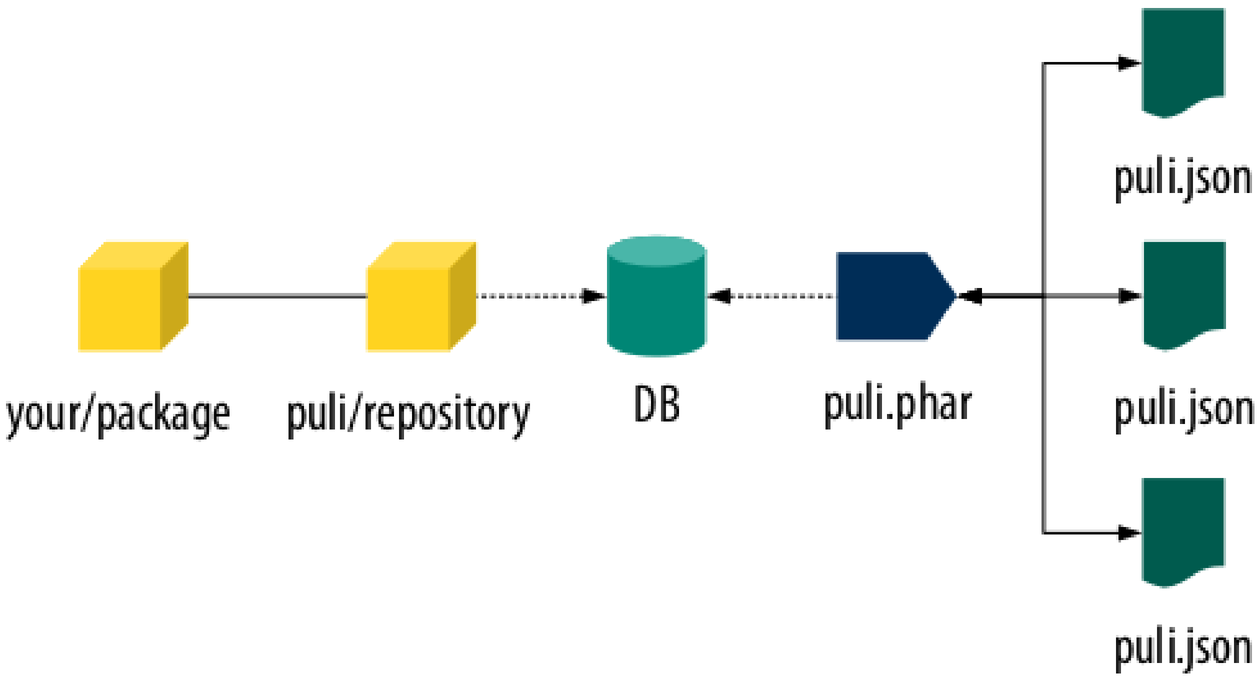
The Repository Component manages resources in a repository that looks very much like a Unix filesystem. Each resource is accessible through a *Puli path*:

```
// res/views/index.html.twig
echo $repo->get('/app/views/index.html.twig')->getBody();
```

Puli finds this file by loading *path mappings* from the `puli.json` files in your application and all installed Composer package. Such a mapping can be added with the Puli CLI:

```
$ php puli.phar map /app res
```

In this example, you mapped the *path prefix* `/app` to the directory `res` in your application. Puli now knows that any file with the prefix `/app` should be looked for in the `res` directory.



Behind the scenes, the Puli CLI builds a database from the mappings in the different `puli.json` files of your project. This database can be accessed from your PHP code through a `ResourceRepository` instance. The format of the database depends on the concrete `ResourceRepository` implementation.

3.3 Getting Started

Read Getting Started with the Repository Component to learn how to install and use the Repository Component in your project.

CHAPTER 4

Getting Started with the Repository Component

The way you install and use the Repository Component depends on the type of project you are working on:

- *In a Symfony Project*, the Repository Component is installed through Puli's Symfony Bundle.
- *In a PHP Application* that will never be a dependency of other Composer packages, the Repository Component is installed through Puli's Composer Plugin.
- *In a Composer Package* that is a dependency of an application or other packages, the Repository Component needs to be installed manually.

4.1 In a Symfony Project

Important: Before you continue, *install the Puli CLI* on your system.

In a Symfony application, the Repository Component is installed through Puli's Symfony Bundle. Before you install the bundle, set "minimum-stability" to "beta" by entering the following command in a terminal:

```
$ composer config minimum-stability beta
```

The bundle can be installed with Composer. Install Composer and enter the following command in a terminal:

```
$ composer require puli/symfony-bundle:^1.0
```

As with every bundle, add the `PuliBundle` class to your `AppKernel`:

```
class AppKernel extends Kernel
{
    public function registerBundles()
    {
        return array(
            // ...
            new Puli\SymfonyBundle\PuliBundle(),
        );
    }

    // ...
}
```

Now that the bundle is installed, let's add our first *resource* to the Puli repository. We will map the *path prefix* `/app` to the `app/Resources` directory of our application:

```
$ php puli.phar map /app app/Resources
```

All resources stored in the `app/Resources` directory can now be accessed through the path prefix `/app`. As example, we will add an `index.html.twig` file:

```
$ mkdir app/Resources/views
$ echo "Success" > app/Resources/views/index.html.twig
```

Puli's `ResourceRepository` can be used to access all files found through these *path mappings*. Let's print the contents of the `index.html.twig` file in a controller:

```
class PostController
{
    public function indexController()
    {
        $repo = $this->get('puli.repository');

        echo $repo->get('/app/views/index.html.twig')->getBody();
    }
}
```

4.2 In a PHP Application

Important: Before you continue, *install the Puli CLI* on your system.

In a PHP application, the Repository Component is installed through Puli's Composer Plugin. Before you install the plugin, set "minimum-stability" to "beta" by entering the following command in a terminal:

```
$ composer config minimum-stability beta
```

The plugin can be installed with Composer. Install Composer and enter the following command in a terminal:

```
$ composer require puli/composer-plugin:^1.0
```

Now that the plugin is installed, let's add our first *resource* to the Puli repository. We will map the *path prefix* `/app` to the `res` directory of our application:

```
$ php puli.phar map /app res
```

All resources stored in the `res` directory can now be accessed through the path prefix `/app`. As example, we will add an `index.html.twig` file:

```
$ mkdir res/views
$ echo "Success" > res/views/index.html.twig
```

Puli's `ResourceRepository` can be used to access all files found through these *path mappings*. Use the Puli factory to create the `ResourceRepository` instance:

```
$factoryClass = PULI_FACTORY_CLASS;
$factory = new $factoryClass();

$repo = $factory->createRepository();
```

Note: For performance reasons, Puli services such as `$factory` or `$repo` should be created only once per application. Instead of storing them in global variables, it is usually nicer to use a Dependency Injection Container for creating the services on demand. A simple Dependency Injection Container for small projects is Pimple.

Let's print the contents of the `index.html.twig` file in our PHP code:

```
echo $repo->get('/app/views/index.html.twig')->getBody();
```

4.3 In a Composer Package

Important: Before you continue, *install the Puli CLI* on your system.

In a Composer package, the Repository Component is installed manually. Before you install the component, set "minimum-stability" to "beta" by entering the following command in a terminal:

```
$ composer config minimum-stability beta
```

Install Composer and enter the following command in a terminal:

```
$ composer require puli/repository:^1.0
```

Now that the component is installed, let's add our first *resource* to the Puli repository. We will map the *path prefix* `/batman/blog` to the `res` directory of our package:

```
$ php puli.phar map /batman/blog res
```

Note: By convention, path prefixes match the name of their Composer package. If your Composer package is called `my/package`, use `/my/package` as Path Prefix.

All resources stored in the `res` directory can now be accessed through the path prefix `/batman/blog`. As example, we will add an `index.html.twig` file:

```
$ mkdir res/views
$ echo "Success" > res/views/index.html.twig
```

Puli's `ResourceRepository` can be used to access all files found through these *path mappings*. Let's print the contents of the `index.html.twig` file in our PHP code:

```
use Puli\Repository\Api\ResourceRepository;

class ResourcePrinter
{
    private $repo;

    public function __construct(ResourceRepository $repo)
    {
        $this->repo = $repo;
    }

    public function printResources()
    {
        echo $this->repo->get('/batman/blog/views/index.html.twig')->getBody();
    }
}
```

You should never create `ResourceRepository` instances in a Composer package. Instead, let the application that uses your package create the repository and pass it to your code. This way, every part of the application uses the same instance and benefits of caching and other optimizations done internally.

4.4 Further Reading

- Read Directory Layout to learn about the recommended directory layout for Puli projects.
- Read Mapping Resources to learn about creating and modifying path mappings with the Puli CLI.
- Read Working with Resources to learn how to use the `ResourceRepository` API to access and work with resources in your PHP code.
- Read Stream Wrappers to learn how to register a PHP stream wrapper for a Puli repository.

CHAPTER 5

Directory Layout

We recommend to follow a certain directory layout in your project. This is by no means mandatory, but it will improve your experience when working with Puli.

Most importantly, we recommend to separate PHP code and non-PHP resources into two separate top-level directories:

```
src/
    MyService.php
    ...
res/
    config/
        config.yml
    ...
```

The names of these directories don't matter – you can name them `source`, `resources` or whatever else you prefer. The important point is that the two directories do not overlap. If the directories overlap, both the class autoloader and the resource repository need to process unnecessary files.

Second, we recommend to use the following names for the sub-directories of the resource directory:

```
config/
    ... configuration files ...
public/
    css/
        ... CSS files ...
    js/
        ... Javascript files ...
    images/
        ... images ...
trans/
    ... translation files ...
views/
    ... templates ...
```

Using common names ensures a consistent user experience when referencing resources in your project and any other Puli-enabled package:

```
// Rendering an application template with Twig
$twig->render('/app/views/index.html');

// Rendering a package template with Twig
$twig->render('/acme/blog/views/post/show.html.twig');
```

The public resources are bundled in a directory `public` because this way these resources can be easily copied to sub-directories of your public directory:

```
/app/public/* -> /public_html/
/acme/blog/public/* -> /public_html/blog/
...
```

Mapping Resources

Puli paths can be mapped to files or directories with the `map` command of the Puli CLI:

```
$ php puli.phar map /acme/blog res
```

The first argument is a *path prefix*, followed by one or more paths in your project. By convention, the path prefix should equal the name of your Composer package with an additional leading slash ("/"). The resulting *path mapping* is added to the `puli.json` file of your package.

Tip: If you develop an application that is not used as Composer dependency by other packages, use `/app` as path prefix.

You cannot just map directories, but also individual files. This is helpful if you need to cherry-pick files from specific locations:

```
$ php puli.phar map /acme/blog/css/reset.css shared/reset.css
```

6.1 Listing Mappings

The current path mappings can be displayed by calling the `map` command without arguments:

```
$ php puli.phar map
The following path mappings are currently enabled:

    Package: puli/acme-blog

        Puli Path   Real Path(s)
        /acme/blog  res
```

The path displays all path mappings currently found in your project. If you want to display just the path mappings of your own package (the *root package*), pass the `--root` option. If you want to display just the path mappings of a package with a specific name, use the `--package` option. With `map -h`, you can learn more about the `map` command:

```
$ php puli.phar map -h
```

6.2 Listing Mapped Files

Now that we mapped paths to the repository, it would be nice to know which *resources* the repository actually contains. You can use `ls` – just like the `ls` command on UNIX – to list the resources in the repository:

```
$ php puli.phar ls
acme
$ php puli.phar ls /acme/blog/config
config.yml  config-dev.yml  doctrine
```

You can also print the resources as tree with `tree`:

```
$ php puli.phar tree
/
-- acme
|   -- blog
|       -- config
|           -- ...
-- app
    -- ...
```

This command prints the whole repository by default. You can also pass the Puli path of an individual resource if you want to print just a part of the repository:

```
$ php puli.phar tree /acme/blog/config
/acme/blog/config
-- config.yml
-- config-dev.yml
-- doctrine
|   -- Acme.Blog.Post.dcm.xml
|   -- ...
-- ...
```

At last, use `find` to list resources according to different criteria:

```
$ php puli.phar find --name *.yml
FileResource /acme/blog/config/config.yml
FileResource /acme/blog/config/config-dev.yml
```

Pass `-h` to find out more about each command's arguments and options:

```
$ php puli.phar find -h
```

6.3 Changing a Mapping

You can add paths to a path mapping with the `map -u` (or `--update`) command:

```
$ php puli.phar map -u /acme/blog --add assets
$ php puli.phar map
The following path mappings are currently enabled:

    Package: puli/acme-blog

        Puli Path   Real Path(s)
        /acme/blog  res, assets
```

When a path mapping contains multiple files or directories, later mappings *override* earlier mappings. That means, the file `/acme/blog/css/style.css` will be looked for first in `assets/css/style.css`, then in `res/css/style.css`.

Likewise, paths can be removed from a path mapping with the `--remove` option:

```
$ php puli.phar map -u /acme/blog --remove assets
$ php puli.phar map
The following path mappings are currently enabled:

    Package: puli/acme-blog

        Puli Path   Real Path(s)
        /acme/blog  res
```

6.4 Deleting a Mapping

Path mappings can be removed completely with `map -d` (or `--delete`):

```
$ php puli.phar map -d /acme/blog
```

6.5 Referencing Other Packages

Sometimes it is necessary to map path prefixes to files or directories in other Composer packages. A typical use case is when you use packages that don't contain a `puli.json` file.

Use the prefix `@<vendor/package>:` to reference the install path of another package:

```
$ php puli.phar map /acme/theme @acme/theme:res
```

The example above maps the Puli path `/acme/theme` to the `res` directory of the "acme/theme" package.

6.6 Further Reading

- Read Working with Resources to learn how to use the resources returned by the resource repository.

CHAPTER 7

Working with Resources

You can retrieve *resources* from Puli's `ResourceRepository` with the `get()` method:

```
$resource = $repo->get('/css/style.css');
```

The `get()` method accepts the *Puli path* of a resource and returns a `Resource`.

If you want to retrieve multiple resources at once, use `find()`. This method accepts a glob pattern and returns a `ResourceCollection`:

```
foreach ($repo->find('/css/*')->getPaths() as $path) {
    echo $path;
}

// => /css/reset.css
// => /css/style.css
```

You can check whether a resource exists by passing its path to `contains()`:

```
if ($repo->contains('/css/style.css')) {
    // ...
}
```

Like `find()`, this method also accepts glob patterns. If you pass a glob, the method will return `true` only if at least one resource matched the pattern.

7.1 Resources

The `get()` method returns `Resource` instances. This interface provides access to the name and the Puli path of the resource:

```
$resource = $repo->get('/css/style.css');

echo $resource->getName();
// => style.css

echo $resource->getPath();
// => /css/style.css
```

Resources don't necessarily have to be located on the filesystem. But those that do implement `FilesystemResource`, which lets you access the filesystem path with `getFilesystemPath()`:

```
$resource = $repo->get('/css/style.css');

echo $resource->getFilesystemPath();
// => /path/to/res/assets/css/style.css
```

Resources that have a body - such as files - implement `BodyResource`. This interface lets you access the body with `getBody()`:

```
$resource = $repo->get('/css/style.css');

$css = $resource->getBody();
```

7.2 Child Resources

Resources support nested resources. In Puli, these are called *child resources*. One prime example is a filesystem directory which may contain other directories and files.

You can access the children of a resource with the methods `getChild()`, `hasChild()` and `listChildren()`:

```
$resource = $directory->getChild('style.css');

if ($directory->hasChild('style.css')) {
    // ...
}

foreach ($directory->listChildren() as $name => $resource) {
    // ...
}
```

7.3 Metadata

Resources support the method `getMetadata()` which returns a `ResourceMetadata` instance. This interface gives access to additional data about a resource. For example, you can use `getModificationTime()` to access the UNIX timestamp of the resource's last modification. This is useful for caching:

```
$resource = $repo->get('/css/style.css');

if ($resource->getMetadata()->getModificationTime() > $cacheTimestamp) {
    // refresh cache
}
```

7.4 Resource Collections

When you fetch multiple resources from the repository, they are returned within a `ResourceCollection` instance. Resource collections offer convenience methods for accessing the names and the Puli paths of all contained resources at once:

```
$resources = $repo->get('/css/*.css');

print_r($resources->getNames());
// Array
// (
//     [0] => reset.css
//     [1] => style.css
// )

print_r($resources->getPaths());
// Array
// (
//     [0] => /css/reset.css
//     [1] => /css/style.css
// )
```

Resource collections are traversable, countable and support `ArrayAccess`. When you still need the collection as array, call `toArray()`:

```
$array = $resources->toArray();
```

CHAPTER 8

Stream Wrappers

Puli supports a stream wrapper that lets you access the *resources* in the repository transparently through PHP's file functions. To register the wrapper, call the `register()` method and pass the name of a URI schema and a `ResourceRepository` instance:

```
use Puli\Repository\StreamWrapper\ResourceStreamWrapper;

ResourceStreamWrapper::register('puli', $repo);
```

After registering the stream wrapper, you can pass *Puli paths* to regular PHP functions, prefixed by the registered URI scheme:

```
$contents = file_get_contents('puli:///acme/blog/css/style.css');

foreach (scandir('puli:///acme/blog') as $entry) {
    // ...
}
```

8.1 Performance Tweaks

The method `register()` needs to be called in every request even if the stream wrapper is not used. This means that also the repository needs to be loaded in every request, which can have a negative performance impact on your application. To mitigate this performance impact, you can pass a callable to `register()` that loads the repository on demand:

```
class ServiceRegistry
{
    private $repo;

    public function getRepository()
    {
        if (!$this->repo) {
            $factoryClass = PULI_FACTORY_CLASS;
            $factory = new $factoryClass();
            $this->repo = $factory->createRepository();
        }

        return $this->repo;
    }
}
```

```php
$registry = new ServiceRegistry();

ResourceStreamWrapper::register('puli', array($registry, 'getRepository'));
```

CHAPTER 9

The URL Generator Component

The URL Generator Component generates URLs for the *resources* in your Puli repository.

9.1 The Problem

When writing HTML, CSS or JavaScript code, you frequently need to refer to other resources of the web server. A simple example is a HTML `<img>` tag:

```
<img src="/images/logo.png" />
```

Hard-coded URLs, however, have a few issues:

Changing the Deployment Target

If you decide to host all images on another server – like a Content Delivery Network (CDN) – you need to manually add the domain name of the CDN to all image paths.

Changing the Version

If you version files by appending `?v1` query parameters, you need to manually update the versions whenever you publish a new release.

Creating Reusable Packages

If your code is part of a Composer package, you force the users of the package to publish your resources at exactly the location that you have hardcoded in your code. In the above example, for instance, the user can't choose to move the file to `/blog/images/logo.png` instead.

9.2 How It Works

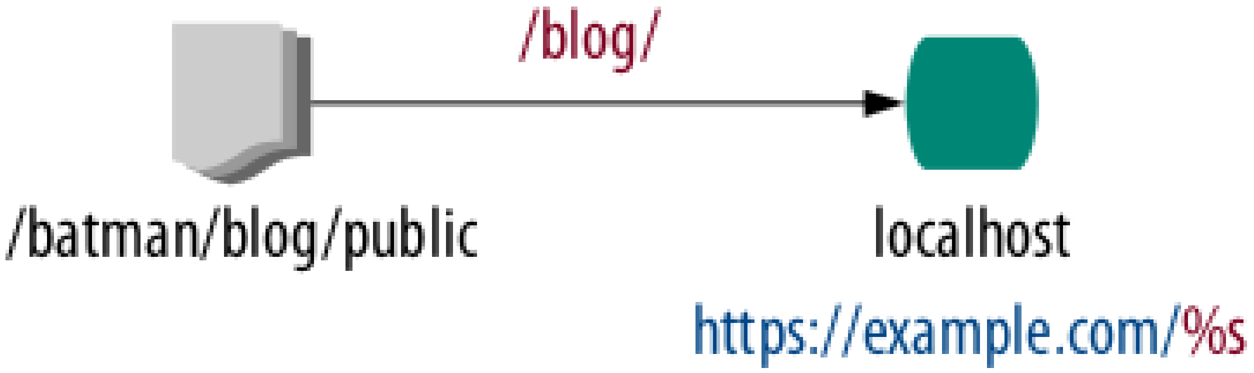
Puli solves this problem by automating the URL generation of your resources. Instead of hardcoding the URLs, you pass the *Puli path* of the resource to the URL Generator Component:

```
<img src="{{ resource_url("/batman/blog/public/images/logo.png") }}" />
```

You can also use relative paths to shorten your code:

```
<img src="{{ resource_url("../public/images/logo.png") }}" />
```

The end user of your package finally configures how the URLs should be generated:



At first, the end user registers at least one *web server*. A web server in Puli has a name and a *URL format*. The URL format tells Puli how the generated URLs should look like. In this example we use "localhost" as the name and `https://example.com/%s` as the URL format of our web server.

Next, Puli resources are mapped to the web server. Such resources are called *public resources*. In this example we map the Puli path `/batman/blog/public` to the `/blog/` directory in the document root of the web server.

The generated URL looks like this:

```
<img src="https://example.com/blog/images/logo.png" />
```

Puli finally automates the deployment of your public resources to the Web Server. By telling Puli where your web server is located and how the public resources should be moved there (symlink, copy, rsync, ...), you can install them with a single CLI command:

```
$ php puli.phar publish --install
Installing /batman/blog/public into public_html/blog via symlink...
```

9.3 Getting Started

Read Getting Started with the URL Generator Component to learn how to install and use public resources in your project.

CHAPTER 10

Getting Started with the URL Generator Component

The way you install and use the URL Generator Component depends on the type of project you are working on:

- *In a Symfony Project*, the URL Generator Component is installed through Puli's Symfony Bundle.
- *In a PHP Application* that will never be a dependency of other Composer packages, the URL Generator Component is installed through Puli's Composer Plugin.
- *In a Composer Package* that is a dependency of an application or other packages, the URL Generator Component needs to be installed manually.

10.1 In a Symfony Project

Important: Before you continue, *install the Puli CLI* and the Repository Component in your project.

In a Symfony application, the URL Generator Component is installed through Puli's Symfony Bundle. If you followed Getting Started with the Repository Component, this bundle is already installed in your application.

With Puli's `UrlGenerator` class you can generate URLs for a *Puli path*. Let's print the URL of the `app/Resources/public/images/logo.png` file in a controller:

```
class PostController
{
    public function indexController()
    {
        $generator = $this->get('puli.url_generator');

        echo $generator->generateUrl('/app/public/images/logo.png');
    }
}
```

Tip: In Twig templates you can use the `resource_url()` function of Puli's Twig Extension. See the documentation of the extension for more information.

Before the above code actually works, you need to tell Puli how to generate your URLs. Add a *web server* for your `web` directory with the Puli CLI:

```
$ php puli.phar server --add localhost web
```

In the example, the server is named “localhost”, but you can choose any name you like.

Next, publish the resources under /app/public to the server:

```
$ php puli.phar publish /app/public localhost
```

Now the URL is generated correctly in the template:

```
<img src="/images/logo.png" />
```

You can install all public resources in the web directory with the publish --install command:

```
$ php puli.phar publish --install
Installing /app/public into web via symlink...
```

10.2 In a PHP Application

Important: Before you continue, *install the Puli CLI* and the Repository Component in your project.

In a PHP application, the URL Generator Component is installed through Puli’s Composer Plugin. If you followed Getting Started with the Repository Component, this bundle is already installed in your application.

With Puli’s UrlGenerator class you can generate URLs for a *Puli path*. Use the Puli factory to create the UrlGenerator instance:

```
$factoryClass = PULI_FACTORY_CLASS;
$factory = new $factoryClass();

$repo = $factory->createRepository();
$discovery = $factory->createDiscovery($repo);
$generator = $factory->createUrlGenerator($discovery);
```

Note: For performance reasons, Puli services such as $factory or $repo should be created only once per application. Instead of storing them in global variables, it is usually nicer to use a Dependency Injection Container for creating the services on demand. A simple Dependency Injection Container for small projects is Pimple.

Use the generateUrl() method to generate URLs for a *resource*:

```
echo $generator->generateUrl('/app/public/images/logo.png');
```

Tip: Install Puli’s Twig Extension to generate URLs in Twig templates.

This code will echo the URL for the res/public/images/logo.png file. Before the code actually works, you need to tell Puli how to generate your URLs. Add a *web server* for the document root of your web server with the Puli CLI:

```
$ php puli.phar server --add localhost public_html
```

In the example, the server is named “localhost”, but you can choose any name you like. We assume that the document root of the server is the public_html directory in your project.

Next, publish the resources under /app/public to the server:

```
$ php puli.phar publish /app/public localhost
```

Now the URL is generated correctly:

```
echo $generator->generateUrl('/app/public/images/logo.png');
// => /images/logo.png
```

You can install all public resources in the `public_html` directory with the `publish --install` command:

```
$ php puli.phar publish --install
Installing /app/public into public_html via symlink...
```

10.3 In a Composer Package

Important: Before you continue, *install the Puli CLI* and the Repository Component in your project.

In a Composer package, the URL Generator Component is installed manually. Before you install the component, set "minimum-stability" to "beta" in `composer.json`:

```
{
    "minimum-stability": "beta"
}
```

Install the component with Composer:

```
$ composer require puli/url-generator:^1.0
```

With Puli's `UrlGenerator` class you can generate URLs for a *Puli path*. Use the `generateUrl()` method to generate URLs for a *resource*:

```
echo $generator->generateUrl('/app/public/images/logo.png');
```

Tip: Install Puli's Twig Extension to generate URLs in Twig templates.

This code will echo the URL for the `res/public/images/logo.png` file. However, before this code actually works, the application that uses your package must register a *web server* and publish your resources there.

You should never create `UrlGenerator` instances in a Composer package. Instead, let the application that uses your package create the URL generator and pass it to your code. This way, every part of the application uses the same instance and benefits of caching and other optimizations done internally.

10.4 Further Reading

- Read Web Server Configuration to learn more about configuring web servers.

Web Server Configuration

Before Puli can generate URLs for your *public resources*, you need to add a *web server* to the Puli configuration and publish your resources there.

11.1 Adding a Web Server

A web server can be added with the `server --add` command of the Puli CLI:

```
$ php puli.phar server --add localhost public_html
```

This command adds a new server to your `puli.json`. The command receives a name for the server as first argument. Here we chose “localhost”, but you can use any name you like. The second argument is the path to the document root of the server.

11.2 Listing Web Servers

You can list all configured web servers with the `server` command:

```
$ php puli.phar server
Server Name  Installer  Document Root  URL Format
localhost    symlink    public_html    /%s
```

The command returns a list of servers and their current configuration. In this example, our configuration contains one server named “localhost”. The document root of the server is the `public_html` directory. Resources are installed there using symbolic links. Finally, the format string `/%s` is used to generate URLs for the resources published to this server. You will learn more about *URL formats* in *Custom URL Formats*.

11.3 Changing a Web Server

The configuration of a web server can be changed with the `server -u` (or `--update`) command:

```
$ php puli.phar server -u localhost --document-root public_html
```

The first argument is the name of the server you want to change. You can pass additional options with the updated options. Run `server -h` to learn more about the supported options and their usage:

```
$ php puli.phar server -h
```

11.4 Deleting a Web Server

A web server can be deleted with the `server -d` (or `--delete`) command:

```
$ php puli.phar server -d localhost
```

11.5 Publishing Resources

To publish resources to a server, use the `publish` command:

```
$ php puli.phar publish /app/public localhost
```

The command stores the created mapping in your `puli.json` file. The first argument is the Puli path of the resource(s) you want to publish. The second is the name of the server you want to publish the resources in.

By default, the resources are published in the document root of the web server. If you want to publish the resources in a sub-directory instead, pass the path to the directory in the third argument:

```
$ php puli.phar publish /batman/blog/public localhost /blog
```

11.6 Listing Public Resources

You can display all public resources with the `publish` command:

```
$ php puli.phar publish
The following resources are published:

    Server localhost
    Location:   public_html
    Installer:  symlink
    URL Format: /%s

        4f1f14 /app/public         /
        d1a9d5 /batman/blog/public /blog

Use "puli publish --install" to install the resources on your servers.
```

Every public resource is identified by a *UUID*. This UUID can be used to update or delete the resource.

11.7 Unpublishing a Resource

To unpublish a resource, call `publish -d` (or `--delete`) with the UUID of the resource in question:

```
$ php puli.phar publish -d d1a9d5
```

11.8 Installing Resources

The `publish` command does not actually move your public resources to the web server. This is done by `publish --install`:

```
$ php puli.phar publish --install
Installing /app/public into public_html via symlink...
Installing /batman/blog/public into public_html/blog via symlink...
```

By default, Puli tries to create symbolic links in the document of your web server. If that's not possible, it will fall back to copying your files.

11.9 Custom URL Formats

By default, web servers use the URL format `/%s` to generate URLs for resources published to that server. The format is a simple sprintf-string: The `%s` in the format is replaced by the path of the resource relative to the document root of the server.

For example, if you publish your `/app/public` directory to the document root of your server, Puli will generate the URL `/images/logo.png` for the Puli path `/app/public/images/logo.png`.

You can change the URL format with the `--url-format` option when adding or changing a server:

```
$ php puli.phar server -u localhost --url-format http://example.com/%s
```

There are two major use cases for changing the default URL format:

- When you install your resources on a different server, you need to include the domain name in every URL.
- By appending a query string (like `?v2`) to your generated URLs you can implement a very simple cache invalidation mechanism for your resources.

The Discovery Component

The Discovery Component connects Composer packages that **consume** *resources* with Composer packages that **provide** resources.

12.1 The Problem

At the moment, you need to write a lot of boilerplate code if you use different packages that use each other's files. Suppose you install the following two packages in your application:

- The package `thor/translator`, which uses `*.yml` files to translate text to some language.
- The package `batman/blog`, which contains the files `blog.en.yml` and `blog.fr.yml` with translations for the strings in the package.

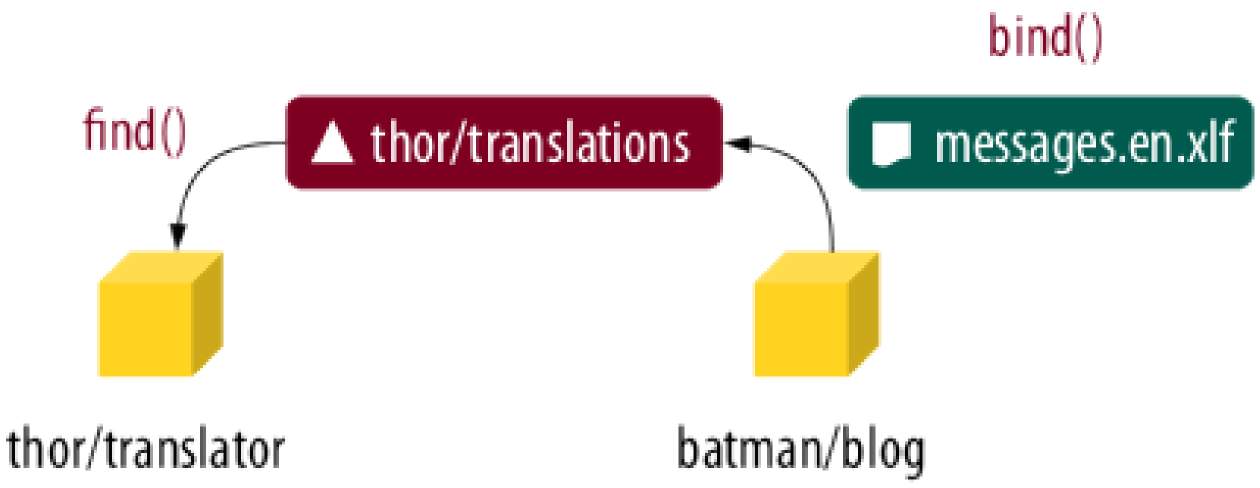
As user of these packages, you have to register the `*.yml` files with the `Translator` instance of the `thor/translator` package:

```
$translator = new Translator(array(
    __DIR__.'/../vendor/batman/blog/res/trans/blog.en.yml',
    __DIR__.'/../vendor/batman/blog/res/trans/blog.fr.yml',
));
```

If you're lucky, your framework does this job for you. However, offloading this task to framework developers doesn't scale, since for any combination of package and framework, *someone* needs to write and maintain such integration code. That's a lot of duplicated work.

12.2 How It Works

Puli's Discovery Component solves this problem by decentralizing the resource registration process. Composer packages are divided into *resource consumers* and *resource providers*.



Resource consumers, like our translator, register a name for the resources they want to use. This name is called a *binding type*. In this example, the developer of the translator defines the binding type "thor/translations" and publishes that type in their documentation.

Other packages (the resource providers) assign their translation files to this binding type. This is called a *binding*. In the end, the translator fetches all translation files for its binding type from a `ResourceDiscovery` instance.

By communicating only through Puli's Discovery Component and a common binding type, consumers and providers are completely decoupled from each other. As user of these packages, you only need to pass the `ResourceDiscovery` instance to the translator:

```
$translator = new Translator($discovery);
```

Puli takes care of everything else.

12.3 Getting Started

Read Getting Started with the Discovery Component to learn how to install and use the Discovery Component in your project.

CHAPTER 13

Getting Started with the Discovery Component

The way you install and use the Discovery Component depends on the type of project you are working on:

- *In a Symfony Project*, the Discovery Component is installed through Puli's Symfony Bundle.
- *In a PHP Application* that will never be a dependency of other Composer packages, the Discovery Component is installed through Puli's Composer Plugin.
- *In a Composer Package* that is a dependency of an application or other packages, the Discovery Component needs to be installed manually.

13.1 In a Symfony Project

Important: Before you continue, *install the Puli CLI* and the Repository Component in your project.

In a Symfony application, the Discovery Component is installed through Puli's Symfony Bundle. If you followed Getting Started with the Repository Component, this bundle is already installed in your application.

As a *resource consumer*, you can define a new *binding type* with the `type --define` command of the Puli CLI:

```
$ php puli.phar type --define thor/translations
```

Note: The name of a binding type must always start with the vendor name of your Composer package. This prevents naming collisions and makes sure that you *own* the type.

The `type --define` command adds the new binding type to your `puli.json`.

Use Puli's `ResourceDiscovery` to access all *bindings* bound to your binding type. Each `ResourceBinding` instance may refer to more than one `Resource`, for example if the bound path contains the wildcard "*". Consequently you need a double loop to access the resources in your code:

```
class PostController
{
    public function indexController()
    {
        $bindings = $this->get('puli.discovery')->findByType('thor/translations');

        foreach ($bindings as $binding) {
            foreach ($binding->getResources() as $resource) {
                echo $resource->getPath();
            }
        }
    }
}
```

As a *resource provider*, you can bind resources to a binding type with the `bind` command of the Puli CLI:

```
$ php puli.phar bind /app/trans/*.yml thor/translations
```

The `bind` command adds the new binding to your `puli.json`.

If a resource consumer and a resource provider are installed at the same time, Puli makes sure that the consumer receives all resources bound to its binding type by the provider.

13.2 In a PHP Application

Important: Before you continue, *install the Puli CLI* and the Repository Component in your project.

In a PHP application, the Discovery Component is installed through Puli's Composer Plugin. If you followed Getting Started with the Repository Component, this bundle is already installed in your application.

As a *resource consumer*, you can define a new *binding type* with the `type --define` command of the Puli CLI:

```
$ php puli.phar type --define thor/translations
```

Note: The name of a binding type must always start with the vendor name of your Composer package. This prevents naming collisions and makes sure that you *own* the type.

The `type --define` command adds the new binding type to your `puli.json`.

With Puli's `ResourceDiscovery` class you can access all *bindings* bound to your binding type. Use the Puli factory to create the `ResourceDiscovery` instance:

```
$factoryClass = PULI_FACTORY_CLASS;
$factory = new $factoryClass();

$repo = $factory->createRepository();
$discovery = $factory->createDiscovery($repo);
```

Note: For performance reasons, Puli services such as `$factory` or `$repo` should be created only once per application. Instead of storing them in global variables, it is usually nicer to use a Dependency Injection Container for creating the services on demand. A simple Dependency Injection Container for small projects is Pimple.

Each `ResourceBinding` instance may refer to more than one `Resource`, for example if the bound path contains the wildcard "*". Consequently you need a double loop to access the resources in your code:

```
$bindings = $discovery->findByType('thor/translations');

foreach ($bindings as $binding) {
    foreach ($binding->getResources() as $resource) {
        echo $resource->getPath();
    }
}
```

As a *resource provider*, you can bind resources to a binding type with the `bind` command of the Puli CLI:

```
$ php puli.phar bind /app/trans/*.yml thor/translations
```

The `bind` command adds the new binding to your `puli.json`.

If a resource consumer and a resource provider are installed at the same time, Puli makes sure that the consumer receives all resources bound to its binding type by the provider.

13.3 In a Composer Package

Important: Before you continue, *install the Puli CLI* and the Repository Component in your project.

In a Composer package, the Discovery Component is installed manually. Before you install the component, set "minimum-stability" to "beta" in `composer.json`:

```
{
    "minimum-stability": "beta"
}
```

Install the component with Composer:

```
$ composer require puli/discovery:^1.0
```

As a *resource consumer*, you can define a new *binding type* with the `type --define` command of the Puli CLI:

```
$ php puli.phar type --define thor/translations
```

Note: The name of a binding type must always start with the vendor name of your Composer package. This prevents naming collisions and makes sure that you *own* the type.

The `type --define` command adds the new binding type to your `puli.json`.

With Puli's `ResourceDiscovery` class you can access all *bindings* bound to your binding type. Each `ResourceBinding` instance may refer to more than one `Resource`, for example if the bound path contains the wildcard "*". Consequently you need a double loop to access the resources in your code:

```
use Puli\Discovery\Api\ResourceDiscovery;

class TranslationLoader
{
    private $discovery;

    public function __construct(ResourceDiscovery $discovery)
    {
        $this->discovery = $discovery;
    }

    public function loadTranslations($catalog, $language)
    {
        $bindings = $this->discovery->findByType('thor/translations');
        $resourceName = $catalog.'.'.$language.'.yml';

        foreach ($bindings as $binding) {
            foreach ($binding->getResources() as $resource) {
                if ($resourceName === $resource->getName()) {
                    return Yaml::parse($resource->getBody());
                }
            }
        }

        return array();
    }
}
```

You should never create `ResourceDiscovery` instances in a Composer package. Instead, let the application that uses your package create the discovery and pass it to your code. This way, every part of the application uses the same instance and benefits of caching and other optimizations done internally.

As a *resource provider*, you can bind resources to a binding type with the `bind` command of the Puli CLI:

```
$ php puli.phar bind /batman/blog/trans/*.yml thor/translations
```

The `bind` command adds the new binding to your `puli.json`.

If a resource consumer and a resource provider are installed at the same time, Puli makes sure that the consumer receives all resources bound to its binding type by the provider.

13.4 Further Reading

- Read Consuming Resources to learn more about resource consumers.
- Read Providing Resources to learn more about resource providers.

CHAPTER 14

Consuming Resources

Resource consumers are packages that load *resources* of a specific type or format from other packages. Imagine a translation package that contains a `Translator` class. Translations used by the `Translator` can be provided by other packages, the *resource providers*, in the form of `*.yml` files.

Puli supplies the infrastructure to connect resource consumers with their providers. The consumer defines a *binding type* for the type of resource it is interested in. The provider binds resources to this binding type. At last, the consumer is able to load all bound resources through Puli's `ResourceDiscovery`.

14.1 Defining a Binding Type

A binding type can be defined with the `type --define` command of the Puli CLI:

```
$ php puli.phar type --define thor/translations
```

This command adds the new binding type to the `puli.json` file of your package.

Note: The name of a binding type must always start with the vendor name of your Composer package. This prevents naming collisions and makes sure that you *own* the type.

You should publish the name of your binding type in your documentation so that resource providers for your package know which type to use.

14.2 Listing Binding Types

You can list all defined binding types with the `type` command:

```
$ php puli.phar type
The following binding types are currently enabled:

    Package: thor/translator

        Type               Description  Parameters
        thor/translations
```

14.3 Type Descriptions

You should add a short description for your binding type that explains what kind of resource the binding type expects to be bound to. A description can be set with the `--description` option when adding the type:

```
$ php puli.phar type --define thor/translations \
    --description "A Yaml file with translations for the Translator class."
```

The description is shown in the output of the `type` command:

```
$ php puli.phar type
The following binding types are currently enabled:

    Package: thor/translator

        Type                Description                                     Parameters
        thor/translations   A Yaml file with translations for the
                            Translator class.
```

14.4 Changing a Binding Type

Binding types can be changed with the `type -u` (or `--update`) command. The command takes the name of the binding type as argument and one or more options for the value that you want to change:

```
$ php puli.phar type -u thor/translations \
    --description "A Yaml file with translations for the Translator class."
```

Call `type -h` to learn about the different options supported by the `-u` command:

```
$ php puli.phar type -h
```

14.5 Deleting a Binding Type

Binding types can be removed with the `type -d` (or `--delete`) command:

```
$ php puli.phar type -d thor/translations
```

This command will remove the given binding type from your `puli.json`. Note that you can only remove binding types defined in your own package. If you try to remove a binding type defined by a different package installed through Composer, the `type -d` command will fail.

14.6 Loading Bound Resources

You can load the *bindings* for your binding type with the `findByType()` method of the `ResourceDiscovery`. Pass the name of your binding Type as first and only argument:

```
$bindings = $discovery->findByType('thor/translations');
```

The method returns an array of `ResourceBinding` instances. With `getResources()`, you can load the resources matched by the binding:

```
foreach ($bindings as $binding) {
    foreach ($binding->getResources() as $resource) {
        // do something with $resource...
    }
}
```

14.7 Loading Bindings for a Resource

You can load the bindings for a specific resource with the `findByPath()` method. This method accepts a *Puli path* as first argument:

```
$bindings = $discovery->findByPath('/batman/blog/trans/messages.en.yml');
```

If you are only interested in bindings of one binding type, pass the name of the binding type as second argument:

```
$bindings = $discovery->findByPath('/batman/blog/trans/messages.en.yml', 'thor/translations');
```

14.8 Binding Parameters

You can define custom parameters for a binding type. These parameters can be used to store additional data with a binding. For example, consider that the name of the translation catalog (here: "messages") is not taken from the file name, but from the *binding parameter* "catalog".

You can add the parameter with the `--param` option when defining the binding type:

```
$ php puli.phar type --define thor/translations --param catalog
```

Each binding *must* now specify a value for this parameter:

```
$ php puli.phar bind /batman/blog/trans/messages.*.yml --param catalog="blog"
```

Use `getParameterValue()` on the `ResourceBinding` to check the value of your parameter:

```
$bindings = $discovery->findByType('thor/translations');

foreach ($bindings as $binding) {
    if ('blog' === $binding->getParameterValue('catalog')) {
        // do something with $binding...
    }
}
```

14.9 Optional Parameters

Often it makes sense to define a default value for a binding parameter. You can set a default value by setting the `--param` option in the form `<param>="<default>"`:

```
$ php puli.phar type --define thor/translations --param catalog="messages"
```

Note: The quotes (") are optional if the parameter value does not contain spaces.

Binding parameters with a default value are *optional*. If no parameter value is set for a binding, the default value is used.

CHAPTER 15

Providing Resources

Resource providers are packages that contain *resources* that should be loaded by a specific *resource consumer*. Imagine a package with a list of translations stored in a `messages.en.yml` file that is supposed to be loaded by a specific translator. If the translation package, the resource consumer, defines a *binding type*, you can pass your translation file to that type. Puli takes care of supplying your file to the translator.

15.1 Adding a Binding

A *binding* can be added with the `bind` command of the Puli CLI:

```
$ php puli.phar bind /batman/blog/trans/*.yml thor/translations
```

The command takes a *Puli path* or a glob as first argument. The second argument is the name of the binding type you want to bind your resources to. The created binding is added to the `puli.json` file of your package.

The `bind` command fails if the binding type is not found. You should *always* install the packages defining the binding types you are using ("require" or "require-dev" in `composer.json`) so that Puli can validate whether your binding is specified correctly. If, for some reason, that's not possible, use `-f` (or `--force` when adding your binding:

```
$ php puli.phar bind -f /batman/blog/trans/*.yml thor/translations
```

15.2 Listing Bindings

Bindings can be listed with the `bind` command without arguments:

```
$ php puli.phar bind
The following bindings are currently enabled:

Package: batman/blog

    UUID    Glob                         Type
    bb5a07  /batman/blog/trans/*.yml  thor/translations
```

Each binding has a *UUID* that is used to identify the binding.

15.3 Changing a Binding

Bindings can be changed with the `bind -u` (or `--update`) command. The command takes the UUID of the binding together with one or more options with the values that you want to change:

```
$ php puli.phar bind -u bb5a07 --query /batman/blog/trans/messages.*.yml
```

Call `bind -h` to learn about the different options supported by the `-h` command:

```
$ php puli.phar bind -h
```

15.4 Deleting a Binding

Bindings can be removed with the `bind -d` (or `--delete`) command:

```
$ php puli.phar bind -d bb5a07
```

This command removes the given binding from your `puli.json`. You can only remove bindings defined by your package. If you try to remove a binding of a different package, the command will fail.

15.5 Disabling a Binding

You can't delete bindings defined by other packages, but you can disable them with the `bind --disable` command:

```
$ php puli.phar bind --disable bb5a07
```

This command marks the matching binding as disabled in your `puli.json`. Disabled bindings are treated like deleted bindings, except that they are not physically removed.

Disabled bindings only work in the `puli.json` of the *root package*. For any other installed package, disabled bindings are ignored.

15.6 Enabling a Binding

A disabled binding can be enabled with the `bind --enable` command:

```
$ php puli.phar bind --enable bb5a07
```

This command simply reverses the effects of `bind --disable` in your `puli.json` file.

15.7 Binding Parameters

If the binding type you are binding to specifies *binding parameters*, you can set values for these parameters with the `--param` option of the `bind` command. The option must be passed in the format `<param>="<value>"`:

```
$ php puli.phar bind /batman/blog/trans/*.yml thor/translations --param catalog="blog"
```

Note: The quotes (”) are optional if the parameter value does not contain spaces.

You can learn which parameters are supported/required by the binding type by calling the `type` command:

```
$ php puli.phar type
The following binding types are currently enabled:

    Package: thor/translator

        Type               Description                            Parameters
        thor/translations  A Yaml file with translations for the  catalog
                           Translator class.
```

CHAPTER 16

The Puli Bridge for Symfony

Puli supports a bridge for the Symfony components. The bridge provides a file locator for the Symfony Config component that locates configuration files through a Puli repository. With this locator, you can refer from one configuration file to another by its *Puli path*:

```
# routing.yml
_acme_demo:
    resource: /acme/demo-bundle/config/routing.yml
```

16.1 Installation

Important: Before you continue, *install the Puli CLI* and the Repository Component in your project.

Install the Puli Bridge with Composer:

```
$ composer require puli/symfony-bridge:^1.0
```

16.2 Configuration

To locate configuration files with Puli, create a new `PuliFileLocator` and pass it to your file loaders:

```
use Puli\SymfonyBridge\Config\PuliFileLocator;
use Symfony\Component\Routing\Loader\YamlFileLoader;

$loader = new YamlFileLoader(new PuliFileLocator($repo));

// Locates the file through Puli's repository
$routes = $loader->load('/acme/blog/config/routing.yml');
```

The `PuliFileLocator` receives Puli's `ResourceRepository` as only argument.

16.3 Chained Locators

If you want to use the `PuliFileLocator` and Symfony's conventional `FileLocator` side by side, you can use them both by wrapping them into a `FileLocatorChain`:

```
use Puli\SymfonyBridge\Config\PuliFileLocator;
use Puli\SymfonyBridge\Config\FileLocatorChain;
use Puli\SymfonyBridge\Config\ChainableFileLocator;
use Symfony\Component\Routing\Loader\YamlFileLoader;

$locatorChain = new FileLocatorChain(array(
    new PuliFileLocator($repo),
    // Symfony's FileLocator expects a list of paths
    new ChainableFileLocator(array(__DIR__)),
));

$loader = new YamlFileLoader($locatorChain);

// Loads the file from __DIR__/config/routing.yml
$routes = $loader->load('config/routing.yml');
```

`ChainableFileLocator` is a simple extension of Symfony's `FileLocator` that supports the interface required by the locator chain. Note that this locator must come **after** the `PuliFileLocator` in the chain.

Puli also provides a chainable version of the file locator bundled with the Symfony HttpKernel component: Use the `ChainableKernelFileLocator` if you want to load configuration files from Symfony bundles:

```
use Puli\SymfonyBridge\Config\PuliFileLocator;
use Puli\SymfonyBridge\Config\FileLocatorChain;
use Puli\SymfonyBridge\Config\ChainableFileLocator;
use Puli\SymfonyBridge\HttpKernel\ChainableKernelFileLocator;

$locatorChain = new FileLocatorChain(array(
    new PuliFileLocator($repo),
    new ChainableKernelFileLocator($httpKernel),
    new ChainableFileLocator(array(__DIR__)),
));

$loader = new YamlUserLoader($locatorChain);

// Loads the file from AcmeBlogBundle
$routes = $loader->load('@AcmeBlogBundle/Resources/config/routing.yml');
```

Take care again that the `ChainableKernelFileLocator` comes last in the chain.

16.4 Limitations

Due to limitations with Symfony's `FileLocatorInterface`, relative file references are not properly supported. Let's load some routes for example:

```
$routes = $loader->load('/acme/blog/config/routing-dev.yml');
```

Assume that this file contains the following import:

```
# routing-dev.yml
_main:
    resource: routing.yml
```

What happens if we override this file in the Puli repository?

```
// Load files from /path/to/blog
$repo->add('/acme/blog', '/path/to/blog');
```

```
// Override just routing.yml with a custom file
$repo->add('/acme/blog/config/routing.yml', '/path/to/routing.yml');

// Load the routes
$routes = $loader->load('/acme/blog/config/routing-dev.yml');

// Expected: Routes loaded from
//   - /path/to/blog/config/routing-dev.yml
//   - /path/to/routing.yml

// Actual: Routes loaded from
//   - /path/to/blog/config/routing-dev.yml
//   - /path/to/blog/config/routing.yml
```

This is a limitation in Symfony and cannot be worked around. For this reason, `PuliFileLocator` does not support relative file paths.

CHAPTER 17

The Puli Bundle for Symfony Projects

There are two ways of using Puli with the Symfony framework:

- New projects can be started based on the Symfony Puli Edition.
- Existing projects can be extended with the Puli Bundle.

Both ways are described in detail below.

17.1 Starting a Project from the Symfony Puli Edition

A new project can be started based on the Symfony Puli Edition with Composer. Install Composer and enter the following command in a terminal:

```
$ composer create-project puli/symfony-puli-edition /path/to/project "~2.5@dev"
```

Tip: To download the vendor files faster, add the `--prefer-dist` option at the end of any Composer command.

Composer will create a new project based on the Symfony Puli Edition in */path/to/project* with Symfony 2.5.

Read the Installing and Configuring Symfony to learn more about installing Symfony distributions.

17.2 Installing the Puli Bundle

If you cannot or don't want to start off the Symfony Puli Edition, you need to install the Puli Bundle with Composer. Install Composer and enter the following command in a terminal:

```
$ composer require "puli/symfony-bundle:~1.0"
```

Note: Make sure that the "minimum-stability" setting is set to "beta" in composer.json, otherwise the installation will fail:

```
{
    ...,
    "minimum-stability": "beta"
}
```

This will download the bundle to your project.

When this command completes, run `composer install` to initialize the Composer plugin for Puli:

```
$ composer install
```

Now, enable the bundle by modifying `AppKernel`:

```
// app/AppKernel.php

// ...
class AppKernel extends Kernel
{
    // ...

    public function registerBundles()
    {
        $bundles = array(
            // ...,
            new Puli\SymfonyBundle\PuliBundle(),
        );

        // ...
    }
}
```

The bundle is now installed in your project.

17.3 Bundle Usage

17.3.1 Configuration Files

With the bundle, you can load configuration files by Puli paths. This is mostly needed when loading bundle routes in routing.yml or routing_dev.yml:

```
# routing_dev.yml
_wdt:
    resource: /symfony/web-profiler-bundle/config/routing/wdt.xml
    prefix:   /_wdt
```

This entry will load all routes found under the Puli path `/symfony/web-profiler-bundle/config/routing/wdt.xml`. Usually, the first two directories of a Puli path correspond to the name of a Composer package. In this example, the file `config/routing/wdt.xml` is loaded from the `Resources` directory in the package "symfony/web-profiler".

Read The Puli Bridge for Symfony if you want to learn more about using Puli with Symfony configuration files.

17.3.2 Twig Templates

With the bundle, it is possible to refer to Twig templates by Puli paths. This is typically done in the controller when rendering a template:

```
// DemoController.php

// ...
```

```
class DemoController extends Controller
{
    /**
     * @Route("/hello/{name}", name="_demo_hello")
     */
    public function helloAction($name)
    {
        return $this->render('/acme/demo-bundle/views/demo/hello.html.twig', array(
            'name' => $name,
        ));
    }

    // ...
}
```

In this example, the template at the Puli path `/acme/demo-bundle/views/demo/hello.html.twig` is rendered.

Within Twig templates, you can also refer to other templates by Puli paths:

```
{# views/demo/hello.html.twig #}

{% extends "/acme/demo-bundle/views/layout.html.twig" %}

...
```

This will let the `hello.html.twig` template extend the template `/acme/demo-bundle/views/layout.html.twig`. Instead of passing the absolute Puli path, it is usually more comfortable to pass relative paths instead:

```
{# views/demo/hello.html.twig #}

{% extends "../layout.html.twig" %}

...
```

Read The Puli Extension for Twig to learn more about the Puli extension for Twig.

CHAPTER 18

The Puli Extension for Twig

Puli provides an extension for the Twig templating engine. With this extension, you can refer to template files through *Puli paths*:

```
echo $twig->render('/acme/blog/views/show.html.twig');
```

The extension also adds a `resource_url()` function for generating URLs for *public resources*:

```
<img src="{{ resource_url('/app/public/images/logo.png') }}" />
```

18.1 Installation

Important: Before you continue, *install the Puli CLI* and the Repository Component in your project.

Install the extension with Composer:

```
$ composer require puli/twig-extension:^1.0
```

18.2 Configuration

To activate the extension, create a new `PuliTemplateLoader` and register it with Twig. The loader enables Twig to load templates through Puli paths:

```
use Puli\TwigExtension\PuliTemplateLoader;

$twig = new \Twig_Environment(new PuliTemplateLoader($repo));
```

The loader receives Puli's `ResourceRepository` as only argument.

Next, create a new `PuliExtension` and add it to Twig. The extension adds the `resource_url()` function and does a few more tweaks to properly support Puli in Twig:

```
use Puli\TwigExtension\PuliExtension;

// The $urlGenerator is only needed if you use the resource_url() function
$twig->addExtension(new PuliExtension($repo, $urlGenerator));
```

18.3 Usage in Twig

Using Puli in Twig is straight-forward: Use Puli paths wherever you would usually use a file path. For example:

```
{% extends '/acme/blog/views/layout.html.twig' %}

{% block content %}
    {# ... #}
{% endblock %}
```

Contrary to Twig's default behavior, you can also refer to templates using relative paths:

```
{% extends 'layout.html.twig' %}

{% block content %}
    {# ... #}
{% endblock %}
```

18.4 Resource URLs

You can generate URLs for public Puli resources with the `resource_url()` function:

```
<img src="{{ resource_url('/app/public/images/logo.png') }}" />
```

The function accepts both absolute and relative paths:

```
<img src="{{ resource_url('../public/images/logo.png') }}" />
```

Note: The resource must have been published with the `publish` command of the Puli CLI, otherwise the URL generator will fail. See the URL generator documentation for more information.

Puli (pronounced "poo-lee") is a universal package system for PHP. Puli aims to replace "bundles", "plugins", "modules" and similar specialized packages of different frameworks with one generic, framework independent solution.

CHAPTER 19

The Puli Package

A Puli package is a directory that contains PHP code, non-PHP files and a `puli.json` file. The `puli.json` file configures how your application and other Puli packages access and load the contents of your package.

```
my-package/
    src/
        ... PHP files ...
    res/
        ... non-PHP files ...
    puli.json
```

Puli packages are typically installed with Composer. Add a `composer.json` to the package and distribute it on Packagist. Then your package can be installed with the `composer` command line utility:

```
$ composer require vendor/my-package
```

Features of Puli Packages

Resource Access

Puli provides a naming convention (so called *Puli paths*) to access non-PHP files (YAML, XML, CSS, HTML, images and more) from a Puli package:

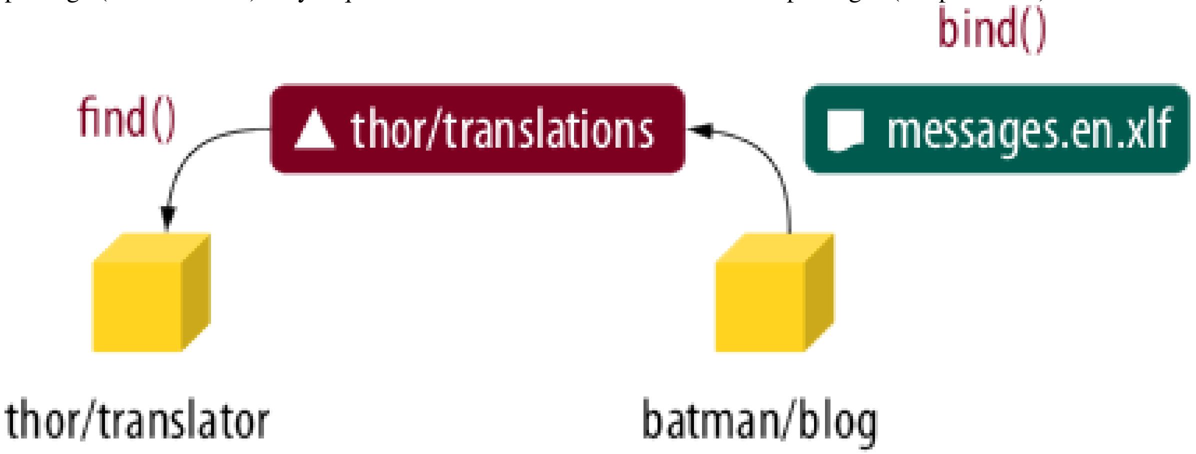
```
// views/index.html.twig in the "batman/blog" package
echo $twig->render('/batman/blog/views/index.html.twig');
```

The path is resolved by reading *path mappings* from the `puli.json` files of your installed Puli packages. Authors of these packages, such as the author of the "batman/blog" package, map prefixes to actual directories with the Puli Command Line Interface (CLI):

```
$ php puli.phar map /batman/blog res
```

Discovery

Puli packages may act as *consumers* and as *providers* of resources and PHP classes. For example, a translator package (the consumer) may request translation files from other installed packages (the providers).



Consumers, like our translator, give a name to the resources they want to load, such as "thor/translations". This name is called a *binding type*. Other packages (the *resource providers*) assign their translation files to this type. This is called *binding*. In the end, you as the user of both packages only need to pass Puli's `Discovery` to the `Translator` class:

```
$translator = new Translator($discovery);
```

The translator then loads all bound translation files from the discovery without any further configuration.

Package Overrides

Puli packages may override other packages. For example, your application or a specialized theme package may replace the `style.css` file provided by a generic package "batman/blog" by a custom version:

```
$ php puli.phar map /batman/blog/css/style.css res/css/blog/style.css
```

Resource URLs

Puli publishes CSS files, JavaScript files and images bundled in your packages and generates their public URLs for you. You can pass Puli paths to simple utility functions to generate URLs in your PHP or HTML code:

```
<img src="{{ resource_url('/batman/blog/public/logo.png') }}" />
```

Puli paths can be marked public with the Puli CLI:

```
$ php puli.phar publish /batman/blog/public localhost /blog/
```

Here, we published the `/batman/blog/public` directory to the sub-directory `/blog/` of the server "localhost". That server must also be defined with the Puli CLI:

```
$ php puli.phar server --add localhost public_html
```

This command registers the server "localhost" with the directory `public_html` used as document root. Now Puli has enough information to generate your URLs:

```
<img src="/blog/logo.png" />
```

Resource Installation

If you want, you can use Puli also to install your public resources on the server. If the server is your localhost, Puli creates simple symbolic links or file copies in your document root:

```
$ php puli.phar publish --install
Installing /batman/blog/public into public_html/blog via symlink...
```

For the final release of Puli, this functionality will be moved to plugins for Gulp, Brunch or similar asset management tools.

CHAPTER 21

Getting Started

Read Installation to get started with Puli.

CHAPTER 22

Infrastructure

Puli's core packages maintain Puli's infrastructure for you.

Package	Source	Current Version	Next Stable Release
The Puli Manager	GitHub	1.0.0-beta10	late 2016
The Command Line Interface	GitHub	1.0.0-beta10	late 2016
The Composer Plugin	GitHub	1.0.0-beta10	late 2016

CHAPTER 23

Components

The Puli components provide the interface between your PHP code and Puli.

Component	Source	Current Version	Next Stable Release
The Repository Component	GitHub	1.0.0-beta10	late 2016
The Discovery Component	GitHub	1.0.0-beta9	late 2016
The URL Generator Component	GitHub	1.0.0-beta4	late 2016

Extensions

Puli's extensions provide integration with different third-party tools.

Extension	Source	Current Version	Next Stable Release
The Symfony Bridge	GitHub	1.0.0-beta4	late 2016
The Symfony Bundle	GitHub	1.0.0-beta10	late 2016
The Twig Extension	GitHub	1.0.0-beta8	late 2016

CHAPTER 25

Authors

- Bernhard Schussek a.k.a. @webmozart
- The Community Contributors

CHAPTER 26

Contribute

Contributions to Puli are very welcome!

- Report any bugs or issues you find on the issue tracker.
- You can grab the source code at Puli's Git organization.

If you find issues in the documentation, please let us know:

- The documentation has a dedicated docs issue tracker.
- The source of the documentation is hosted at the docs Git repository.

CHAPTER 27

Support

If you are having problems, send a mail to bschussek@gmail.com or shout out to @PuliPHP on Twitter.

CHAPTER 28

License

Puli, its extensions and this documentation are licensed under the MIT license.

CHAPTER 29

Logo

The Puli logo was kindly sponsored by Resonanz Online Marketing.