Dependency Injection with Bread::Board

jesse.luehrs@iinteractive.com

A Motivating Example

2

Before getting started with dependency injection itself, I want to start with an example of the kinds of problems it is going to be solving.

```
package MyApp;
use MyFramework;

sub call {
    my $self = shift;

    my $logger = Logger->new(log_file => 'logs/myapp.log');
    $logger->log("connecting to database");

    my $dbh = DBI->connect('dbi:mysql:myapp_db');
    my $hello = $dbh->selectall_arrayref('SELECT * FROM my_table')->[0][0];

    $logger->log("rendering template");
    my $template = Template->new(INCLUDE_PATH => 'root/template');
    $template->process('hello.tt', { hello => $hello }, \(my $output));

    return $output;
}
```


So you're writing a basic web app - it connects to the database, pulls some information out, and renders it into a template.

```
package MyApp;
use MyFramework;

has model => (is => 'ro', isa => 'Model', default => sub { Model->new });
has view  => (is => 'ro', isa => 'View',  default => sub { View->new });

sub call {
    my $self = shift;
    my $hello = $self->model->get_hello;
    return $self->view->render($hello);
}
```


Now you want to start adding new features, but it's hard when everything is hardcoded directly in the code, so you start refactoring. This lets you do things like swap out the model object for a mock during testing. Instead of making the logger a global (globals are bad), you move it into a builder in the Model and View classes.

```
package MyApp;
use MyFramework;

has logger => (
    is => 'ro', isa => 'Logger',
    default => sub { Logger->new }
);

has model => (
    is => 'ro', isa => 'Model', lazy => 1,
    default => sub { Model->new(logger => $_[0]->logger) },
);

has view  => (
    is => 'ro', isa => 'View', lazy => 1,
    default => sub { View->new(logger => $_[0]->logger) },
);

sub call {
    my $self = shift;
    my $hello = $self->model->get_hello;
    return $self->view->render($hello);
}
```


But that required initializing the logger in multiple places, so you pull the logger initialization out into the main app, and just pass it down into where it's needed.

```
package MyApp;
use MyFramework;

has dsn     => (is => 'ro', isa => 'Str',    default => 'dbi:mysql:myapp_db');
has tt_root => (is => 'ro', isa => 'Str',    default => 'root/template');
has logger  => (is => 'ro', isa => 'Logger', default => sub { Logger->new });

has model => (
    is => 'ro', isa => 'Model', lazy => 1,
    default => sub {
        my $m = Model->connect($_[0]->dsn);
        $m->set_logger($_[0]->logger);
        return $m;
    },
);

has view  => (
    is => 'ro', isa => 'View', lazy => 1,
    default => sub {
        View->new(logger => $_[0]->logger, tt_root => $_[0]->tt_root);
    },
);

sub call { ... }
```


But you're still duplicating code in your tests (creating the test model object), so you pull some more things out to allow just specifying a dsn, instead of having to create the entire model object. But look at how complicated this is now!

Dependency Injection

7

So dependency injection allows you to get all of those benefits without it being so much of a mess. You just describe the relationships between your classes, and the framework handles all of the actual object construction.

Dependency Injection

- a form of inversion of control

7

So dependency injection allows you to get all of those benefits without it being so much of a mess. You just describe the relationships between your classes, and the framework handles all of the actual object construction.

Dependency Injection

- a form of inversion of control
- "the inverse of garbage collection"

7

So dependency injection allows you to get all of those benefits without it being so much of a mess. You just describe the relationships between your classes, and the framework handles all of the actual object construction.

Dependency Injection

- a form of inversion of control
- "the inverse of garbage collection"
- manages object construction

7

So dependency injection allows you to get all of those benefits without it being so much of a mess. You just describe the relationships between your classes, and the framework handles all of the actual object construction.

Benefits to Dependency Injection

8

This makes things like testing much easier, because you can just ask the framework for a model object, and it will create one in the exact same way that it does when your application asks for one.

Benefits to Dependency Injection

- provides access to the same object creation code that your app will actually use

8

This makes things like testing much easier, because you can just ask the framework for a model object, and it will create one in the exact same way that it does when your application asks for one.

Benefits to Dependency Injection

- provides access to the same object creation code that your app will actually use
- removes need for globals

8

This makes things like testing much easier, because you can just ask the framework for a model object, and it will create one in the exact same way that it does when your application asks for one.

Benefits to Dependency Injection

- provides access to the same object creation code that your app will actually use
- removes need for globals
- testing and reuse

8

This makes things like testing much easier, because you can just ask the framework for a model object, and it will create one in the exact same way that it does when your application asks for one.

Catalyst

9

Catalyst contains just enough of a dependency injection system to handle its specific application structure, and nothing more. This typically works fine, except when you want to do something outside of that structure (reuse a model class in a helper script, for instance).

Catalyst

- contains a simplistic dependency injection system

9

Catalyst contains just enough of a dependency injection system to handle its specific application structure, and nothing more. This typically works fine, except when you want to do something outside of that structure (reuse a model class in a helper script, for instance).

Catalyst

- contains a simplistic dependency injection system
  - `$c->model('DBIC')` looks up the class `MyApp::Model::DBIC`, and instantiates it as necessary using the data in the app's configuration

9

Catalyst contains just enough of a dependency injection system to handle its specific application structure, and nothing more. This typically works fine, except when you want to do something outside of that structure (reuse a model class in a helper script, for instance).

Catalyst

Catalyst contains just enough of a dependency injection system to handle its specific application structure, and nothing more. This typically works fine, except when you want to do something outside of that structure (reuse a model class in a helper script, for instance).

Catalyst

- it is, however, pretty limited

Catalyst contains just enough of a dependency injection system to handle its specific application structure, and nothing more. This typically works fine, except when you want to do something outside of that structure (reuse a model class in a helper script, for instance).

Catalyst

- it is, however, pretty limited
  - can only create objects, and these objects must be either models or views (typically with specific class names)

10

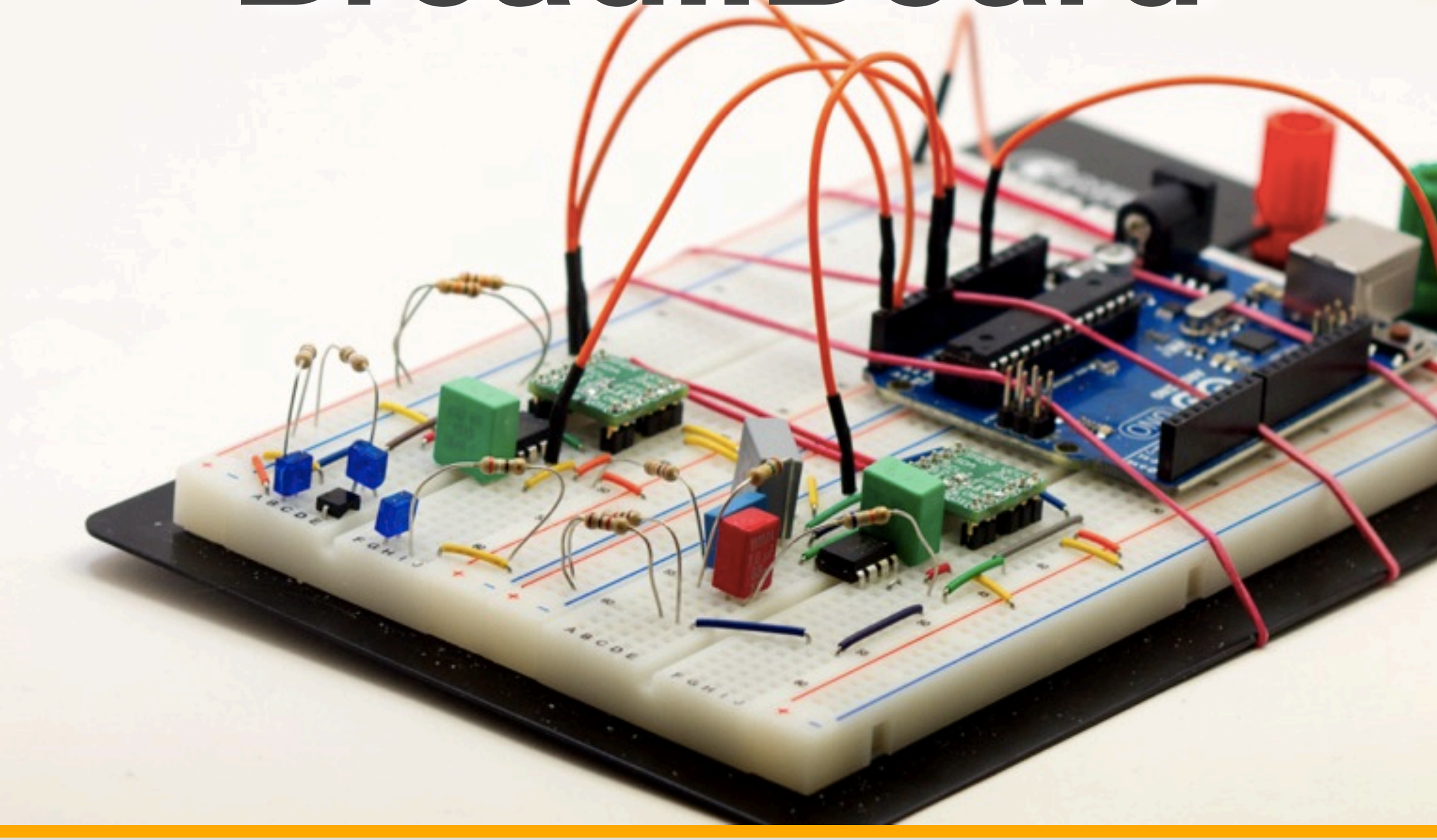
Catalyst contains just enough of a dependency injection system to handle its specific application structure, and nothing more. This typically works fine, except when you want to do something outside of that structure (reuse a model class in a helper script, for instance).

Catalyst

- it is, however, pretty limited
  - can only create objects, and these objects must be either models or views (typically with specific class names)
  - all data to create the objects must be specified in the configuration, objects can't take other objects as constructor parameters

Catalyst contains just enough of a dependency injection system to handle its specific application structure, and nothing more. This typically works fine, except when you want to do something outside of that structure (reuse a model class in a helper script, for instance).

Bread::Board



INFINITY INTERACTIVE

Bread::Board is a fully-featured, general purpose dependency injection system. It provides a common set of functionality so that applications can be built in a consistent way, and interoperate cleanly.

```
my $c = container MyApp => as {
    service dsn    => 'dbi:mysql:myapp_db';
    service logger => (class => 'Logger', lifecycle    => 'Singleton');
    service view   => (class => 'View',   dependencies => ['logger']);

    service model => (
        class        => 'Model',
        dependencies => ['logger', 'dsn'],
        block        => sub {
            my $m = Model->connect($_[0]->param('dsn'));
            $m->set_logger($_[0]->param('logger'));
            return $m;
        },
    );

    service app => (
        class        => 'MyApp',
        dependencies => ['model', 'view'],
    );
};

$c->resolve(service => 'app');
```


This is an example of a Bread::Board container. It will be explained in the following slides.

Services

The services are the representation of the data that you want to create.

Services

- represent the data you're storing

The services are the representation of the data that you want to create.

Services

- represent the data you're storing
- access contents via the `->get` method

The services are the representation of the data that you want to create.

Services

- represent the data you're storing
- access contents via the `->get` method
- three built-in types:

The services are the representation of the data that you want to create.

Bread::Board::ConstructorInjection

```
service view => (
    class => 'View',
);
```


This is a service which will create a new instance of the class View for you.

Bread::Board::BlockInjection

```
service model => (
    class => 'Model', # optional
    block => sub {
        my $m = Model->new
        $m->initialize;
        return $m;
    },
);
```


This is a service which will run the given block to create the object.

Bread::Board::Literal

```
service dsn => 'dbi:mysql:myapp_db';
```


This is a service which just returns the literal data given every time.

Containers

Containers are used to organize groups of services. Services within a given container can be used to help build each other.

Containers

- hold services and other containers

Containers are used to organize groups of services. Services within a given container can be used to help build each other.

Containers

- hold services and other containers
- access contents via the `->fetch` method

Containers are used to organize groups of services. Services within a given container can be used to help build each other.

Containers

- hold services and other containers
- access contents via the `->fetch` method
- `->resolve` is a shortcut method for `->fetch(...)->get`

Containers are used to organize groups of services. Services within a given container can be used to help build each other.

Containers

18

Services and containers within a container are referred to via service paths, which should be a familiar metaphor. This is used when calling ->fetch or ->resolve, and is also used when specifying dependencies.

Containers

- services within a container are referred to via paths

18

Services and containers within a container are referred to via service paths, which should be a familiar metaphor. This is used when calling ->fetch or ->resolve, and is also used when specifying dependencies.

Containers

- services within a container are referred to via paths
  - like UNIX paths

18

Services and containers within a container are referred to via service paths, which should be a familiar metaphor. This is used when calling ->fetch or ->resolve, and is also used when specifying dependencies.

Containers

- services within a container are referred to via paths
  - like UNIX paths
  - relative paths are resolved relative to the container that `->fetch` is called on

18

Services and containers within a container are referred to via service paths, which should be a familiar metaphor. This is used when calling ->fetch or ->resolve, and is also used when specifying dependencies.

Containers

- services within a container are referred to via paths
  - like UNIX paths
  - relative paths are resolved relative to the container that `->fetch` is called on
  - absolute paths are resolved by going up the tree of parent containers until the root is found

18

Services and containers within a container are referred to via service paths, which should be a familiar metaphor. This is used when calling ->fetch or ->resolve, and is also used when specifying dependencies.

Containers

- services within a container are referred to via paths
  - like UNIX paths
  - relative paths are resolved relative to the container that `->fetch` is called on
  - absolute paths are resolved by going up the tree of parent containers until the root is found
  - path components of '..' are allowed to mean "the parent container"

Services and containers within a container are referred to via service paths, which should be a familiar metaphor. This is used when calling ->fetch or ->resolve, and is also used when specifying dependencies.

Dependencies

19

Dependencies are how you tell the services how to help build each other. Any objects that are necessary to build a given object must be specified as dependencies on a service which provides the required object.

Dependencies

- tells `Bread::Board` how your classes are related

19

Dependencies are how you tell the services how to help build each other. Any objects that are necessary to build a given object must be specified as dependencies on a service which provides the required object.

Dependencies

- tells `Bread::Board` how your classes are related
- specified as a map of names to service paths (there are some shortcuts)

Dependencies are how you tell the services how to help build each other. Any objects that are necessary to build a given object must be specified as dependencies on a service which provides the required object.

Dependencies

- tells `Bread::Board` how your classes are related
- specified as a map of names to service paths (there are some shortcuts)
- relative paths are relative to the container that directly contains the service, in this case

Dependencies are how you tell the services how to help build each other. Any objects that are necessary to build a given object must be specified as dependencies on a service which provides the required object.

Dependencies

```
service logger => (class => 'Logger');
service view => (
    class        => 'View',
    dependencies => ['logger'],
);
```


In this example, a logger object will be created via Logger->new, and then the result of that will be passed as View->new(logger => $logger). Note the shortcut for the dependency specification - ['logger'] is equivalent to { logger => 'logger' }.

Dependencies

```
service dsn => 'dbi:mysql:myapp_db';
service model => (
    class        => 'Model',
    dependencies => ['dsn'],
    block        => sub {
        my $service = shift;
        return Model->connect($service->param('dsn'));
    },
);
```


Here, a dependency is specified for a block injector service. In this case, the service itself is passed as an argument to the block, after having all of its dependencies resolved. The resolved dependencies can be accessed via the 'param' method.

Dependencies

```
container MyApp => as {
    container Model => as {
        service dsn => 'dbi:mysql:myapp_db';
        service model => (
            class        => 'Model',
            dependencies => ['dsn'],
            block        => sub {
                my $service = shift;
                return Model->connect($service->param('dsn'));
            },
        );
    };
    service app => (
        class        => 'MyApp',
        dependencies => ['Model/model'],
    );
};
```


In this more complicated example, we have the model being held in a subcontainer, which means that the path to the service has to include components for both the container ('Model') and service ('model'). The dependency specification of ['Model/model'] is equivalent to { model => 'Model/model' }.

Parameters

23

In particular, any place where I previously talked about dependencies in a context other than how to supply them (being passed to constructors, being accessed via the ->param method, etc) can also use parameters.

Parameters

- like dependencies, but supplied when calling `->get` or `->resolve`

23

In particular, any place where I previously talked about dependencies in a context other than how to supply them (being passed to constructors, being accessed via the ->param method, etc) can also use parameters.

Parameters

```
my $c = container MyApp => as {
    service user => (
        class      => 'User',
        parameters => {
            name => { isa => 'Str' },
        },
    );
};

$c->resolve(
    service    => 'user',
    parameters => { name => 'doy' }
);
```


Parameters are specified with a map of parameter names to parameter options. Valid options are 'isa', 'default', and 'optional'.

Parameters

```
my $c = container MyApp => as {
    service user => (
        class      => 'User',
        parameters => {
            name => { isa => 'Str' },
        },
    );
    service superusers => (
        block        => sub { [ $_[0]->param('root') ] },
        dependencies => {
            root => { user => { name => 'root' } },
        },
    );
};
```


Here, a service has a dependency on a service with a required parameter. The parameter can also be supplied as part of the dependency specification.

Parameters

```
my $c = container MyApp => as {
    service user => (
        class => 'User',
        parameters => {
            name => { isa => 'Str' },
        },
    );
    service superusers => (
        block => sub {
            [ $_[0]->param('user')->inflate(name => 'root') ]
        },
        dependencies => ['user'],
    );
};
```


Often though, the parameter value isn't known at the time that the container is being constructed. If a dependency on a parameterized service is specified without the parameter being provided, then the result of resolving the service is a special object, whose purpose is only to accept the parameters (via the 'inflate' method) and return the actual resolved value from the service.

Parameters

```
my $c = container MyApp => as {
    service user => (
        class      => 'User',
        parameters => {
            name => {
                isa      => 'Str',
                optional => 1,
                default  => 'guest'
                },
            },
    );
};
# user with name 'guest'
$c->resolve(service => 'user');

# user with name 'doy'
$c->resolve(service => 'user', parameters => { name => 'doy' });
```


Parameters can also be given defaults. In this case, you typically also want to make them optional.

Parameters

```
my $c = container MyApp => as {
    service default_username => 'guest';
    service user => (
        class => 'User',
        parameters => {
            name => { isa => 'Str', optional => 1 },
        },
        dependencies => {
            name => 'default_username',
        },
    );
};
# user with name 'guest'
$c->resolve(service => 'user');

# user with name 'doy'
$c->resolve(service => 'user', parameters => { name => 'doy' });
```


Parameters and dependencies can both be specified for a service. If an optional parameter is given with the same name as a dependency, it can be used to provide a more powerful mechanism for specifying a default. The parameter will be used if provided when the service is resolved, otherwise the dependency will be resolved as normal and used.

Lifecycles

29

Lifecycles affect how the service is resolved, on a per-service basis. Creating singleton services rather than singleton classes is much better for reuse.

Lifecycles

- this is what determines what happens when `->get` is called

Lifecycles affect how the service is resolved, on a per-service basis. Creating singleton services rather than singleton classes is much better for reuse.

Lifecycles

- this is what determines what happens when `->get` is called
- by default, each call to `->get` creates a new object

Lifecycles affect how the service is resolved, on a per-service basis. Creating singleton services rather than singleton classes is much better for reuse.

Lifecycles

- this is what determines what happens when `->get` is called
- by default, each call to `->get` creates a new object
- by specifying `lifecycle => 'Singleton'` when creating the service, the same object will be returned each time

Lifecycles affect how the service is resolved, on a per-service basis. Creating singleton services rather than singleton classes is much better for reuse.

```
my $c = container MyApp => as {
    service dsn    => 'dbi:mysql:myapp_db';
    service logger => (class => 'Logger', lifecycle    => 'Singleton');
    service view   => (class => 'View',   dependencies => ['logger']);

    service model => (
        class        => 'Model',
        dependencies => ['logger', 'dsn'],
        block        => sub {
            my $m = Model->connect($_[0]->param('dsn'));
            $m->set_logger($_[0]->param('logger'));
            return $m;
        },
    );

    service app => (
        class        => 'MyApp',
        dependencies => ['model', 'view'],
    );
};

$c->resolve(service => 'app');
```


So now this example from earlier should make more sense.

Typemaps

31

Having to manually specify dependencies for every service does get tedious after a while as well. You're already listing these things once when writing your classes (via attributes and type constraints), so it seems a bit redundant to also have to do it again in the Bread::Board configuration. This is what typemaps solve.

Typemaps

- defines a mapping from a `class_type` to a service

Having to manually specify dependencies for every service does get tedious after a while as well. You're already listing these things once when writing your classes (via attributes and type constraints), so it seems a bit redundant to also have to do it again in the Bread::Board configuration. This is what typemaps solve.

Typemaps

- defines a mapping from a `class_type` to a service
- instead of requesting a particular service, you can request an object of a particular type:

```
$c->resolve(type => 'Model');
```


Having to manually specify dependencies for every service does get tedious after a while as well. You're already listing these things once when writing your classes (via attributes and type constraints), so it seems a bit redundant to also have to do it again in the Bread::Board configuration. This is what typemaps solve.

Typemaps

- defines a mapping from a `class_type` to a service
- instead of requesting a particular service, you can request an object of a particular type:
  `$c->resolve(type => 'Model');`
- with this, we can (mostly) infer the dependencies for a given class

31

Having to manually specify dependencies for every service does get tedious after a while as well. You're already listing these things once when writing your classes (via attributes and type constraints), so it seems a bit redundant to also have to do it again in the Bread::Board configuration. This is what typemaps solve.

Typemaps

```
package Model
use Moose;
has logger => (is => 'ro', isa => 'Logger', required => 1);

package Logger;
use Moose;
```

```
my $c = container MyApp => as {
    typemap Logger => infer;
    typemap Model  => infer;
};

$c->resolve(type => 'Model')->logger; # a valid logger object
```


Here, we just tell Bread::Board to 'infer' the dependencies for a given service. It automatically creates a constructor injection service for the specified class, and populates the dependencies by introspecting the class.

Inferred Services

33

Generally when writing your classes, you'll want to make as many things required as possible, moving all of the logic involved in constructing each of the attributes outside of the class. If you do this, inference works very smoothly.

Inferred Services

- required attributes are automatically inferred, becoming either dependencies (on types) or parameters (if the type doesn't exist in the typemap)

33

Generally when writing your classes, you'll want to make as many things required as possible, moving all of the logic involved in constructing each of the attributes outside of the class. If you do this, inference works very smoothly.

Inferred Services

- required attributes are automatically inferred, becoming either dependencies (on types) or parameters (if the type doesn't exist in the typemap)
- non-required attributes can still be satisfied by parameters, or specified manually as dependencies

Generally when writing your classes, you'll want to make as many things required as possible, moving all of the logic involved in constructing each of the attributes outside of the class. If you do this, inference works very smoothly.

```
my $c = container MyApp => as {
    service dsn    => 'dbi:mysql:myapp_db';
    typemap Logger => infer(lifecycle => 'Singleton');
    typemap View   => infer;

    service model => (
        class        => 'Model',
        dependencies => ['type:Logger', 'dsn'],
        block        => sub {
            my $m = Model->connect($_[0]->param('dsn'));
            $m->set_logger($_[0]->param('type:Logger'));
            return $m;
        },
    );

    typemap Model => 'model';

    typemap MyApp => infer;
};

$c->resolve(type => 'MyApp');
```


Using typemaps, the previous example could be rewritten like this. Note that typemaps can be specified for non-inferred services too. Manual dependencies on types are indicated by the 'type:' prefix.

Catalyst and Bread::Board

```
package MyApp;
use Catalyst 'Bread::Board';

__PACKAGE__->config(
    'Plugin::Bread::Board' => {
        container => MyApp::Container->new,
        }
);
```


As mentioned earlier, Catalyst uses its own dependency injection system, which isn't very featureful. Catalyst::Plugin::Bread::Board lets you replace that with Bread::Board. There is also currently work being done to merge this into Catalyst itself.

Best Practices

Containers are for initialization

36

A container that you can access at runtime is effectively the service locator pattern, rather than dependency injection. This loses the benefit of separating out which parts of your code need which other parts, since you can get any service you want out of the container. Also, factories aren't as bad as they sometimes have to be in other languages - closures make perfectly good factories for simple cases.

Best Practices

Containers are for initialization

- passing around containers in order to create objects later makes everything in the container effectively global again

36

A container that you can access at runtime is effectively the service locator pattern, rather than dependency injection. This loses the benefit of separating out which parts of your code need which other parts, since you can get any service you want out of the container. Also, factories aren't as bad as they sometimes have to be in other languages - closures make perfectly good factories for simple cases.

Best Practices

Containers are for initialization

- passing around containers in order to create objects later makes everything in the container effectively global again
- if you need this, you can have your container create factories

A container that you can access at runtime is effectively the service locator pattern, rather than dependency injection. This loses the benefit of separating out which parts of your code need which other parts, since you can get any service you want out of the container. Also, factories aren't as bad as they sometimes have to be in other languages - closures make perfectly good factories for simple cases.

Best Practices

Containers are for initialization

```
package MyApp::Container;
use Moose;
extends 'Bread::Board::Container';

sub BUILD {
    container $self => as {
        # ...
    };
}
```


The container sugar not only works to create new containers, but also to append to existing containers. Just pass a container instance as the first parameter instead of a name.

Best Practices

Containers are for initialization

```
container SomethingElse => as {
    container MyApp::Container->new;
};
```


Putting this configuration in BUILD means that you can create new fully-configured container instances by calling ->new.

Bread::Board::Declare

```
package MyApp::Container;
use Moose;
use Bread::Board::Declare;

has dsn    => (is => 'ro', isa => 'Str', value => 'dbi:mysql:myapp_db');
has logger => (is => 'ro', isa => 'Logger');
has view   => (is => 'ro', isa => 'View', infer => 1);

has model => (
    is           => 'ro',
    isa          => 'Model',
    infer        => 1,
    dependencies => ['dsn'],
    block        => sub {
        my $m = Model->connect($_[0]->param('dsn'));
        $m->set_logger($_[0]->param('logger'));
        return $m;
    },
);

has app => (is => 'ro', isa => 'MyApp', infer => 1);
```


If you need to create more complicated container classes, doing everything in BUILD may still be uglier than necessary. Bread::Board::Declare provides a nicer sugar layer for doing that, by creating services from attributes. This is what the original example would look like in Bread::Board::Declare.

Bread::Board::Declare

This solves the problem of having to modify the container definition itself whenever you want to swap out individual services, which tends to be something that comes up a lot as you use a container in more than one place (including during testing).

Bread::Board::Declare

- services are declared just by defining attributes

40

This solves the problem of having to modify the container definition itself whenever you want to swap out individual services, which tends to be something that comes up a lot as you use a container in more than one place (including during testing).

Bread::Board::Declare

- services are declared just by defining attributes
- attribute accessors resolve the service if no value is set

40

This solves the problem of having to modify the container definition itself whenever you want to swap out individual services, which tends to be something that comes up a lot as you use a container in more than one place (including during testing).

Bread::Board::Declare

- services are declared just by defining attributes
- attribute accessors resolve the service if no value is set
- if the attribute has a value, it is used in dependency resolution

This solves the problem of having to modify the container definition itself whenever you want to swap out individual services, which tends to be something that comes up a lot as you use a container in more than one place (including during testing).

Bread::Board::Declare

- services are declared just by defining attributes
- attribute accessors resolve the service if no value is set
- if the attribute has a value, it is used in dependency resolution

```
MyApp::Container->new(dsn => 'dbi:mysql:other_db')->model
```


This solves the problem of having to modify the container definition itself whenever you want to swap out individual services, which tends to be something that comes up a lot as you use a container in more than one place (including during testing).

Bread::Board::Declare

Unlike previously where you had to define services separately and add them to the typemap manually if you didn't want to infer all dependencies, here you can just specify «infer => 1» and any inferred dependencies are appended to any dependencies you declared explicitly, and all services with a class_type constraint are added to the typemap automatically.

Bread::Board::Declare

- typemaps are much simplified

Unlike previously where you had to define services separately and add them to the typemap manually if you didn't want to infer all dependencies, here you can just specify «infer => 1» and any inferred dependencies are appended to any dependencies you declared explicitly, and all services with a class_type constraint are added to the typemap automatically.

Bread::Board::Declare

- typemaps are much simplified
- attributes with `class_type` constraints automatically get a typemap

Unlike previously where you had to define services separately and add them to the typemap manually if you didn't want to infer all dependencies, here you can just specify «infer => 1» and any inferred dependencies are appended to any dependencies you declared explicitly, and all services with a class_type constraint are added to the typemap automatically.

Bread::Board::Declare

- typemaps are much simplified
- attributes with `class_type` constraints automatically get a typemap
- `infer => 1` infers as many dependencies as possible

Unlike previously where you had to define services separately and add them to the typemap manually if you didn't want to infer all dependencies, here you can just specify «infer => 1» and any inferred dependencies are appended to any dependencies you declared explicitly, and all services with a class_type constraint are added to the typemap automatically.

OX

```
package MyApp;
use OX;

has model => (is => 'ro', isa => 'Model');
has view  => (is => 'ro', isa => 'View');

has controller => (
    is    => 'ro',
    isa   => 'Controller',
    infer => 1,
);

router as {
    route '/' => 'controller.index';
};
```


OX is a new web framework that is built around Bread::Board, that uses Bread::Board::Declare as a sugar layer and Path::Router for dispatching. In this example, a request for '/' would cause a Controller object to be resolved (probably with the help of new Model and View objects), and then the 'index' method would be called on that object to handle the request.

OX

```
# equivalent to MyApp->new->resolve(service => 'App');
my $psgi_app = MyApp->new->to_app;

# attributes work the same way they do in Bread::Board::Declare
my $model = MyApp->new->model;
```


43

The app itself is just a Bread::Board container, so it can be used as a subcontainer in other apps, and resolving services works as expected. Bread::Board::Declare handles resolving attributes, so reusing parts of the container in tests or external scripts becomes trivial.

OX

```
package MyApp;
use OX;

has logger => (is => 'ro', isa => 'Logger');

has model => (is => 'ro', isa => 'Model', infer => 1);
has view  => (is => 'ro', isa => 'View',  infer => 1);

has controller => (
    is    => 'ro',
    isa   => 'Controller',
    infer => 1,
);

router as {
    route '/' => 'controller.index';
};
```


The Bread::Board framework that OX is based on can be used to create objects that have nothing to do with the typical Model/View/Controller layout - in fact, the names used by these attributes are just convention, there is nothing special going on here at all.

Questions??

https://metacpan.org/module/Bread::Board
https://metacpan.org/module/Bread::Board::Declare
https://github.com/stevan/OX