Hopscotch

Paul Everitt

Mar 14, 2022

CONTENTS

Writing a decoupled application – a "pluggable app" – in Python is a common practice. Looking for a modern registry that scales from simple use, up to rich dependency injection (DI)? `hopscotch` is a registry and DI package for Python 3.9+, written to support research into component-driven development for Python's web story.

Let's Be Real

I expect a lot of skepticism. In fact, I don't expect a lot of adoption. Instead, I'm using this to learn and write articles.

CHAPTER

ONE

FEATURES

- *Simple to complex*. The easy stuff for a simple registry is easy, but rich, replaceable systems are in scope also.
- *Better DX*. Improve developer experience through deep embrace of static analysis and usage of symbols instead of magic names.
- *Hierarchical*. A cascade of parent registries helps model request lifecycles.
- *Tested and documented*. High test coverage and quality docs with lots of (tested) examples.- *Extensible*.
- *Great with components*. When used with `viewdom`, everything is wired up and you can just work in templates.

Hopscotch takes its history from `wired`, which came from `Pyramid`, which came from `Zope`.

CHAPTER

TWO

REQUIREMENTS

- Python 3.9+.
- venusian (for decorators)

CHAPTER

THREE

INSTALLATION

You can install `Hopscotch` via pip from PyPI:

```
$ pip install hopscotch
```

CHAPTER

FOUR

QUICK EXAMPLES

Let's look at: a hello world, same but with a decorator, replacement, and multiple choice.

Here's a registry with one "kind of thing" in it:

```
# One kind of thing
@dataclass
class Greeter:
    """A simple greeter."""

    greeting: str = "Hello!"


registry = Registry()
registry.register(Greeter)
# Later
greeter = registry.get(Greeter)
# greeter.greeting == "Hello!"
```

That's manual registration – let's try with a decorator:

```
@injectable()
@dataclass
class Greeter:
    """A simple greeter."""

    greeting: str = "Hello!"


registry = Registry()
registry.scan()
# Later
greeter = registry.get(Greeter)
# greeter.greeting == "Hello!"
```

You're building a pluggable app where people can replace builtins:

```
# Some site might want to change a built-in.
@injectable(kind=Greeter)
@dataclass
class CustomGreeter:
    """Provide a different ``Greeter`` in this site."""
```

(continues on next page)

(continued from previous page)

```
    greeting: str = "Howdy!"
```

Sometimes you want a `Greeter` but sometimes you want a `FrenchGreeter` – for example, based on the row of data a request is processing:

```
@injectable(kind=Greeter, context=FrenchCustomer)
@dataclass
class FrenchGreeter:
    """Provide a different ``Greeter`` in this site."""

    greeting: str = "Bonjour!"

# Much later
child_registry = Registry(
    parent=parent_registry,
    context=french_customer
)
greeter2 = child_registry.get(Greeter)
# greeter2.greeting == "Bonjour!"
```

Finally, have your data constructed for you in rich ways, including custom field "operators":

```
@injectable()
@dataclass
class SiteConfig:
    punctuation: str = "!"


@injectable()
@dataclass
class Greeter:
    """A simple greeter."""

    punctuation: str = get(SiteConfig, attr="punctuation")
    greeting: str = "Hello"

    def greet(self) -> str:
        """Provide a greeting."""
        return f"{self.greeting}{self.punctuation}"
```

The full code for these examples are in the docs, with more explanation (and many more examples.)

And don't worry, dataclasses aren't required. Some support is available for plain-old classes, `NamedTuple`, and even functions.

CHAPTER

FIVE

CONTRIBUTING

Contributions are very welcome. To learn more, see the *contributor's guide*.

CHAPTER

SIX

LICENSE

Distributed under the terms of the MIT license, *Hopscotch* is free and open source software.

CHAPTER

SEVEN

ISSUES

If you encounter any problems, please file an issue along with a detailed description.

CHAPTER

EIGHT

CREDITS

This project was generated from @cjolowicz's Hypermodern Python Cookiecutter template.

8.1 Why Hopscotch?

I'm not convinced the world of Python wants registries (though they should.) I'm *really* not convinced Python wants *another* registry package – there are already several, some that come really close to what I wanted. Registry *plus* dependency injection?

C'mon, man.

This document tries to explain what itches are being scratched in Hopscotch. Remember: I don't actually expect this stack of software to get adopted. It's primarily for me to learn and articulate some ideas from the world of frontend development.

8.1.1 Why Not?

I'll start here. Things like registries are an indirection. All frameworks are by definition an indirection – some mysterious force is calling your code and passing in arguments. Have you ever written a pytest test? If so, a registry is calling your code and even doing dependency injection!

Still, registries have a bad rap in Python. Inversion of control, dependency injection – hell, when even type hints are "too much ceremony", you know something like Hopscotch is in left field.

Though times are kind of changing, thanks to FastAPI and its cohort.

8.1.2 Pluggable Apps

My background in Zope has baked into my consciousness a love of pluggable apps. What's that? A "mostly done", out-of-the-box (OOTB) system with pieces that can be extended (add), replaced (overwritten), *and* varied (multiple implementations from which best-fit is chosen.) If you've ever used Pyramid and seen its predicates – that's what I mean.

As an example, imagine a Sphinx (pluggable app) using a theme (plugin) in a site (local customization.) I'd like to change the breadcrumbs, but *only* in one class of thing, or one part of the site.

8.1.3 Fail Faster

I want static analysis to help drive a better developer experience. Sitting in an editor, I want red warnings when I do something wrong.

"Convention over configuration", with its magically-named variables and files, flies in the face of this. I want to see how far I can go with pluggable systems that express the lines you can paint within, via a smart editor.

8.1.4 Tooling

In a related sense, I want CI to tell me – or even better, people extending or using my downstream system – when the rules are broken. If you've ever written a Sphinx extension, you'll know – it's magical names all the way down.

8.1.5 Caller-Callee Decoupling

Again in Sphinx, if I want to extend something and there wasn't a specially-designed facility (e.g. "put your list of sidebars here as strings"), then I have to fork/monkeypatch the caller. If I'm writing to a plug point, and I need more information than what it will pass me, I have to... fork the caller.

Forking the caller is bad.

So instead, pluggable systems pass around a universe object, where you can get everything (the Sphinx app, the request object.)

8.1.6 Small Surface Area

This is also bad. The callee now has a contract that's... kind of big. They might get passed more arguments than they want. It certainly makes test writing a contemplative exercise.

I want my callable arguments to say exactly what I depend on, no more, no less.

And later, I might want to cache/persist the results. In that case, I *really* don't want to depend on the universe. How do you hash the universe?

8.1.7 Opportunities

Registry-driven injection has some opportunities for fun ideas.

Frontend development is very, very rich in innovation. Views driven by immutable-state stores as reactive observers... it's not overkill, it's actually fodder for some real leaps forward.

One of my biggest goals is to have a component (injectable) constructed by "the system" which tracks what you depended on. If anything changes, we regenerate you, immutably. If you're building a Sphinx site, you might render breadcrumbs once for a folder, and other items therein will use it.

And then, persist that rendered component, so when you wake up next time... if nothing changed, you're already built.

There are other places for cool thinking, like... htmx. If the things you depend on are replaceable, you might have a "Layout" component which knows how to write fragments to disk. Then, getting your htmx-driven views let you fetch smaller payloads.

8.1.8 Yeh, Sure, Back to `jinja2`

It's asking too much for people to rethink templating in Python. But, I'll tinker with it anyway.

8.2 Registry

The registry holds implementations of "kinds of things". It then lets you retrieve an implementation, possibly doing injection along the way.

8.2.1 Creating a Registry

It's quite simple to create a registry:

```
>>> from hopscotch import Registry
>>> registry = Registry()
```

You can also create with a parent and with a context. These are both discussed below.

8.2.2 Using a Registry

Hopscotch comes from the Pyramid family, which doesn't like module-level globals for the "app". Usually your registry would become part of your "app" and passed around as needed.

For example, when using `viewdom`, the registry is passed around behind the scenes – one of the benefits of DI.

8.2.3 Registering Things

Once you have a registry, you can put something into it "imperatively", meaning, by calling a method on that object. For example, imagine you had this code:

```
class Greeting:
    """A dataclass to give a greeting."""

    salutation: str = "Hello"
```

Maybe `Greeting` ships with your pluggable app. But you want to allow a local site to replace it with a different greeting:

```
@dataclass()
class AnotherGreeting(Greeting):
    """A replacement alternative for the default ``Greeting``."""

    salutation: str = "Another Hello"
```

Easy: they just grab the registry and register their custom `AnotherGreeting`, telling the registry it's a "kind of" `Greeting`:

```
>>> registry.register(AnotherGreeting, kind=Greeting)
```

As a note, there's nothing magical about dataclasses at this point. You could just as easily use a "plain old class."

8.2.4 Kind

What's up with that word "kind"? At this point, it's a hedge because I can't make up my mind if the registry will be about types.

As we see below, you can `.get` something from the registry. What is this "something"? Well, it's the best implementation for the situation, out of the registered implementations. But you should get back something that, type-wise, is a "kind of" the thing you asked for, to get the developer experience benefits of static analysis.

It would be *great* if "kind" didn't have to mean "subclass". You could register things that really had no implementation coupling (inheritance) with the thing they were replacing. In fact, you could use a `NamedTuple` or even a function. (Historical fact: you can actually register a function as an implementation of `Greeting`.)

However, tooling – editors, `mypy` – will complain that the function isn't really a type-of `Greeting`. "A-ha" you say, "that's what PEP 544 protocols are for." That's what I thought too. But protocols can't be used in all the places a type can – for example, a `TypeVar`.

So I'm currently stuck with Frankenkind. For now, it's "subtype."

8.2.5 Retrieving From A Registry

Super, we now have a place to store implementations. How do we get one out?

```
>>> greeting = registry.get(Greeting)
>>> greeting.salutation
'Another Hello'
```

Hmm, I got `AnotherGreeting`, not `Greeting`? Yep. There were two implementations. The most recent one – the second one – out-prioritized the earlier one (which is still in the registry.)

And another "Hmm"…I got back an instance, not the class. Yep. The registry constructs your instances. These dataclasses had default values on the fields, so nothing was needed.

8.2.6 Parent Registries

We all work with web-based systems. There's a startup phase, then when a request comes in, a request-response phase. The startup information should be setup once, then the per-request information stored and discarded.

Hopscotch has a hierarchical registry. When you create a registry, you can pass a parent:

```
>>> child_registry = Registry(parent=registry)
```

If you try to get something from the child, it will find the registration in the parent:

```
>>> greeting = child_registry.get(Greeting)
>>> greeting.salutation
'Another Hello'
```

The injector is aware of parentage. When it goes to get something from the registry, it will walk up until it finds the first match.

> **Warning:** I'm In Over My Head Hierarchical registries will ultimately be awesome. While they work now, it's a "just barely" kind of thing. Getting it really right – high performance, lower complexity, caching, and multiprocess – will be hard. (But will be worth it.)

8.2.7 Context

Hooray, here's kind of the whole point of Hopscotch: picking the "best" implementation.

In our mythical web system, a request comes in for `/customer/mary`. I'm just guessing, but we'll probably get the `mary` row from the `Customer` database, as the primary object for that request. Maybe the `Customer` looks like this:

```
@dataclass()
class Customer:
    """The person to greet, stored as the registry context."""

    first_name: str
```

Pyramid makes this a first-class idea called the request "context." If you put it to use, it's really powerful. For example, you can register a view that is custom to that "kind of thing." `wired` also keeps a similar idea in its registry.

Hopscotch does this by letting you, like `wired`, create a registry with an optional `context`:

```
>>> customer = Customer(first_name="mary")
>>> child_registry = Registry(parent=registry, context=customer)
>>> registry.context is None
True
>>> child_registry.context.first_name
'mary'
```

To really see why this is useful, let's start over with a registry that has two registrations:

- A `Greeting` to be used in the general case
- A `AnotherGreeting` to be used when the customer is a `FrenchCustomer`

```
>>> registry = Registry()
>>> registry.register(Greeting)
>>> registry.register(AnotherGreeting, context=FrenchCustomer)
```

A request comes in with no context (or any context that isn't `FrenchCustomer`):

```
>>> child_registry = Registry(parent=registry, context=None)
>>> greeting = registry.get(Greeting)
>>> greeting.salutation
'Hello'
```

Another request comes in – but it's for a `FrenchCustomer`:

```
>>> customer = FrenchCustomer(first_name="marie")
>>> child_registry = Registry(parent=registry, context=customer)
>>> greeting = child_registry.get(Greeting)
>>> greeting.salutation
'Another Hello'
```

This time when we asked for `Greeting`, we got the registration for `context=FrenchCustomer`. Why? Because the child registry was created in a way that was "bound" to that as the registry context.

As a note, you can also manually provide a context when doing a `get`. Let's use a `FrenchCustomer`, but with the parent registry that was created with `context=None`:

```
>>> greeting = registry.get(Greeting, context=customer)
>>> greeting.salutation
'Another Hello'
```

8.2.8 Precedence

The registry lets you register multiple implementations of a "kind." How does the registry decide which to use?

I'll be honest: the current implementation is sketchy, though it has potential. Basically, it looks through the current registry for matches (before going to the parent.) It eliminates those that don't match the context. It then uses a "policy" to decide the best fit.

This can get better/richer/faster in the future.

8.2.9 Decorator

Imperative registration is definitely not-sexy. Let's show use of the `@injectable` decorator.

Let's again imagine we have a `Greeting`, but let's show the line *before* what we showed previously:

```
@injectable()
@dataclass()  # Start Greeting
class Greeting:
    """A dataclass to give a greeting."""

    salutation: str = "Hello"
```

When we create a registry *this* time, we'll call `.scan()` to go look for decorators:

```
>>> registry = Registry()
>>> registry.scan()
```

As before, we can later get this:

```
>>> greeting = registry.get(Greeting, context=customer)
>>> greeting.salutation
'Hello'
```

`.scan()` can be passed a symbol for a package/module, or a string for a package location. It's based on the scanner in `venusian` which is a really cool way to defer registration until *after* import.

8.2.10 Singletons

Sometimes you don't need an instance constructed. The data is "outside" the system and there's only one implementation and you already have the instance. That's where singletons come in.

In Hopscotch you can register a singleton:

```
>>> greeting = Greeting(salutation="I am a singleton")
>>> registry = Registry()
>>> registry.register(greeting)
>>> greeting = registry.get(Greeting)
>>> greeting.salutation
'I am a singleton'
```

You can register a singleton as a "kind" and it will replace a built-in:

```
>>> greeting = AnotherGreeting()
>>> registry = Registry()
>>> registry.register(Greeting)
>>> registry.register(greeting, kind=Greeting)
>>> greeting = registry.get(Greeting)
>>> greeting.salutation
'Hello'
```

Singletons get higher "precedence" than non-singleton registrations. This helps when you want to say: "Listen, this is the answer in this registry."

> **Warning:** Is This Dumb? I went back and forth on whether singletons should use the same method for registering and getting. I settled on "simpler DX". But it makes the type hinting harder.

8.2.11 Props

We'll cover this more in *injection*, but as a placeholder....when you do a `registry.get()` you can pass in kwargs to use in the construction. These are called "props", to mimic component-driven development.

8.3 Injection

Hopscotch has two faces: a registry for implementations *and* a dependency injector for construction.

What does that even mean?

In Hopscotch, here's the flow:

- Something asks for a "kind of thing"...maybe a component asks for a subcomponent
- The registry gets the best implementation
- The registry then *calls* that implementation, supplying it the arguments it is asking for

- Some of those arguments might be "kinds of things" which also need instances
- Later, when I'm brave, this will all be cached and persistent

If *registries* are an "OMG too much magic" thing in Python, then dependency injection is a "you're trying to make this into Java" kind of thing. In Hopscotch, I want to show we can lower the bar to make the simple case really simple, especially for consumers.

Let's roll.

8.3.1 The Simplest Case

We have a `Greeter` who says hello with a greeting. Actually, a `Greeting` – an instance of a class that is in "the system".

```
@injectable()
@dataclass()
class Greeter:
    """A dataclass to engage a customer."""

    greeting: Greeting
```

We can make a pluggable which has `Greeting` and `Greeter` in its registry:

```
>>> from hopscotch import Registry
>>> registry = Registry()
>>> registry.scan()
```

Now that we're all setup, let's ask for a `Greeter`:

```
>>> greeter = registry.get(Greeter)
>>> greeter.greeting.salutation
'Hello'
```

What happened?

- We asked the registry for `Greeter`
- It found the "best-fit" implementation…in this case, `Greeter` itself
- The registry started constructing an instance by introspecting its fields
- The `Greeter.greeting` field had a type hint of `Greeting`
- The registry had an implementation for `Greeting`
- Since `Greeting` had a default value for its one field, it could be constructed
- The registry used that constructed instance to construct `Greeter`
- Done

"Woah, dataclass magical mumbo jumbo!" you say. Well, here's an example using a plain-old-class:

```
class Greeter:
    """A plain-old-class to engage a customer."""
```

(continues on next page)

(continued from previous page)

```
    greeting: Greeting

    def __init__(self, greeting: Greeting):
        """Construct a greeter."""
        self.greeting = greeting
```

Here's a `NamedTuple`:

```
class Greeter(NamedTuple):
    """A ``NamedTuple`` to engage a customer."""

    greeting: Greeting
```

…but with a caveat: `NamedTuple` and functions (next) have a little sharp edge regarding the "kind" discussion in *Registry*. Here's a function for `Greeter` that can also be dependency injected:

```
class Greeter(NamedTuple):
    """A ``NamedTuple`` to engage a customer."""

    greeting: Greeting
```

Even for the "simple" case, this is pretty valuable. Really de-coupled systems, where you can add things without monkey-patching and the callees get to decide what they need.

8.3.2 Manual Factory

"Too much magic!" It's true that the injector has a good number of policy decisions in the service of "helping." Perhaps you'd like to keep injection, but have manual control over construction? For that, provide a class method named `__hopscotch_factory`:

```
@dataclass()
class GreetingFactory:
    """Use the ``__hopscotch_factory__`` protocol to control creation."""

    salutation: str

    @classmethod
    def __hopscotch_factory__(cls, registry: Registry) -> GreetingFactory:
        """Manually construct this instance, instead of injection."""
        return cls(salutation="Hi From Factory")
```

8.3.3 Generics

Hopscotch injection works by the type hint. Provide a type, Hopscotch tries to go get it and make an instance for you. But those type hints can be…rich. Here's a `Greeter` who can have an optional `Greeting`:

```
@dataclass()
class GreeterOptional:
    """A dataclass to engage a customer with optional greeting."""

    greeting: Optional[Greeting]  # no default
```

Dataclasses especially have some extra generics to cover their fields.

8.3.4 Default Values

When you're constructing or calling something – dataclass, plain old class, NamedTuple, function – the parameters might have default values.

A dataclass might have a field with a default value:

```
class Greeting:
    """A dataclass to give a greeting."""

    salutation: str = "Hello"
```

But so might a function:

```
def GreeterOptional(greeting: Optional[str]) -> Optional[str]:
    """A function to engage a customer with optional greeting."""
    return greeting
```

The default value is the lowest-precedence option. The injector tries to go get a value from the registry based on the field/parameter's type. In these two cases, the type hint says `str` which, obviously, won't be in the registry.

8.3.5 Operators

Now, on to the part where Hopscotch actually adds to the status quo.

In some cases, we want a little transform between what we're asking for and what we're getting. For example, perhaps we have a registry with a context:

```
>>> registry = Registry()
>>> registry.register(Greeting)
>>> registry.register(AnotherGreeting, kind=Greeting)
>>> registry.register(Greeter)
```

We'd like to get `Greeting` out, but we know it's really going to be `AnotherGreeting`. For this we can use an "operator": a simple callable class which is given some inputs and returns an output:

```
@injectable()
@dataclass()
class GreeterGetAnother:
    """Use an operator to change the type hint of what's retrieved."""

    customer_name: AnotherGreeting = get(Greeting)
```

What is `get`? It's an "operator":

```
@dataclass(frozen=True)
class Get:
    """Lookup a kind and optionally pluck an attr."""

    lookup_key: Any
```

(continues on next page)

(continued from previous page)

```
    attr: Optional[str] = None

    def __call__(
        self,
        registry: Registry,
    ) -> object:
        """Use registry to find lookup key and optionally pluck attr."""
        # Can't lookup a string, ever, so bail on this with an error.
        if isinstance(self.lookup_key, str):
            lk = self.lookup_key
            msg = f"Cannot use a string '{lk}' as container lookup value"
            raise ValueError(msg)

        result_value = registry.get(self.lookup_key)

        # Are we plucking an attr?
        if self.attr is not None:
            result_value = getattr(result_value, self.attr)

        return result_value
```

In this case, we're saying: "Sure, go get me a `Greeting`, but actually, I know it is a `AnotherGreeting`."

Here's a super-useful variation: get me a `Greeting` and then pluck the attribute I'm really looking for:

```
@dataclass()
class GreeterFirstName:
    """A dataclass that gets an attribute off a dependency."""

    customer_name: str = get(Customer, attr="first_name")
```

Let's see it in action. I have a registry which registers a `Customer` singleton and `GreeterFirstName`, then gets a `Greeter`:

```
>>> registry = Registry()
>>> customer = Customer(first_name="Mary")
>>> registry.register(customer)  # A singleton
>>> registry.register(GreeterFirstName, kind=Greeter)
>>> greeter = registry.get(Greeter)
>>> greeter.customer_name
'Mary'
```

Operators act like a DSL, giving instructions to the injector. Since you can very easily write your own, it provides a nice way to concentrate your injectables on what they *really* need. Minimizing the surface area with the outside system has benefits.

8.3.6 Annotated

We just discussed operators. I lied a little: `get` isn't strictly an operator. It's actually a `dataclasses.field` which stuffs some expected values – namely, `operator` – in the metadata part of a `field`.

This is actually syntactic sugar over a more verbose form that can be used *outside* dataclasses: plain old classes, `NamedTuple`, and even functions. For example, here's a function that asks the registry to get the `Customer` and pluck the `first_name`:

```
def GreeterAnnotated(
    customer_name: Annotated[str, Get(Customer, attr="first_name")]
) -> str:
    """A function to engage a customer with an ``Annotated``."""
    return customer_name
```

Previously, the `dataclasses.field` metadata communicated with the injector. This uses PEP 593 – Flexible function and variable annotations to give injector instructions. The `NamedTuple` and plain old classes also use this. In fact, dataclasses can use `Annotated` also, though there's no real reason to. As long as the operator has a "field" version – `Get` vs. `get` – it's a lot more convenient to use the latter.

8.3.7 Context

Sometimes you want to inject a field that has an attribute off the context. You can't just say `get(Context)` as there isn't a `Context` class registered on the registry. Instead, it's an attribute.

Instead, there's another operator for this: the `Context` operator with its `context` field:

```
@injectable(context=FrenchCustomer)
@dataclass()
class GreeterFrenchCustomer:
    """A dataclass that depends on a different registry context."""

    customer: FrenchCustomer = context()
```

This does the moral equivalent of grabbing `registry.context`. It also supports passing in `attr=` to pluck just one attribute.

8.3.8 Props

We've seen how Hopscotch can gather the inputs needed to construct an instance: symbols in the registry, operators that return values, default values, etc.

Hopscotch was written to power `ViewDOM` and its software for component-driven development. In frontends, components are usually passed "props" in a particular usage. Hopscotch also allows "props": values passed in during `registry.get()` which have the highest precedence.

Here's an example of a registry with a `Greeting`:

```
>>> registry = Registry()
>>> registry.register(Greeting)
```

I'm now in a request and, for some reason, I want to supply a specific `salutation`, as a "prop":

```
>>> greeting = registry.get(Greeting, salutation="Hello Prop")
>>> greeting.salutation
'Hello Prop'
```

What would that look like in ViewDOM? In a template somewhere, you'd say: `<{Greeting} salutation="Hello Prop" //>`.

8.3.9 Same Ol' Dataclasses

As can be inferred, dataclasses are the "first-class citizen." As such, there's several aspects that are accommodated. For example, fields that should be handled in a `__post_init__` are deferred to let the dataclass handle that field, rather than injecting it.

8.3.10 inject_callable

ViewDOM works well with Hopscotch, but doesn't require it. You can have some utility components as functions that aren't in the registry, because of the whole type/kind thing. Or, you can just skip completely the "replaceable components" thing and just rely in symbols as the only implementation.

For this, Hopscotch lets you use the injector independently of the registry via the `hopscotch.inject_callable` function. It's a bit more work: you have to supply a `Registration` object that is the result of introspecting the target to be constructed. (There's a function for that too.)

8.4 Future Work

Just to reiterate: I don't expect much adoption of this. I'm treating it less as "a maybe big package on PyPI" and more like "topics to write articles about."

With that said, here are some places I'm interested in taking this:

8.4.1 Badass Sphinx "Theme"

Hopscotch is the lowest layer. Above that are a number of other things, resulting in the real target. I'd like to make a really neat "theme" that works with Sphinx, Pelican, Pyramid, and whatever. When I say "theme"…it's a bit of a departure from what's in Sphinx. Let's just say it that way.

8.4.2 Kind

I would really like to crack the nut on protocols and really allow implementations that don't subclass, but still fulfill the contract. I'm skeptical, though: mypy is just pretty overwhelmed with what's on its plate.

One easy first step to improve the developer experience (DX) is to take a page out of Will McGugan's handbook and infer the type. Let's say we had this:

```
@injectable()
@dataclass
class AnotherHeading(Heading):
    title: str
```

We could skip needing `@injectable(kind=Heading)` with this logic:

- Get the base classes
- If that class is registered, register this class as a kind

8.4.3 Precedence

My current scheme for deciding on the best implementation is pretty naive and brittle. I'd like to restructure the datastructure for the registrations – for the hundredth time – to make it more efficient, effective, and simple to get the best match.

8.4.4 Performance

In a similar sense, lookups are going to be grossly less efficient than the standard Python "go get me this function." I need it to be a little less gross and possibly rely on caching.

I've tried to think the entire time around ideas of immutability, making decisions up-front, doing work only once, etc. I can extend it to an idea of: take all the inputs, make a hashable named-tuple, and remember what came out.

8.4.5 Persistence

"A good Gatsby for Python" has been a target of mine. I'd like re-render time to be super-fast, but also startup time. There are ways to remember the introspection results and only update them when software changes.

8.4.6 Reactive

If you're going to compete with Hugo, and you're in Python…you have to do some tricks. The biggest being: do the minimal amount work needed on each operation.

I'd like a component system that remembers injection and scribbles down the observer and observable. Then, when the observable changes, go find everything that injected it, and update it.

Ambitious. Then again, frontend systems are on their 4th generation of these ideas.

8.4.7 Configuration Step

At the moment, you can just keep adding registrations to a registry at any time.

Systems like Pyramid have an explicit configuration step which closes at some point. Hopscotch could benefit from this – in performance, reliability, and simplicity – by using this to re-compute more efficient datastructures in the registry. It could also implement the alternative to "kind=" mentioned above.

8.4.8 Predicates

Ahh, the really big win. Pyramid has a concept of registrations with *predicates*: extra kwargs of registration information. These are then used to find really specific best-fit registrations. For example, "use this kind of Heading in this section of the site."

I've written this before, for Decate. It's kind of fun and certainly useful.

8.5 Reference

`Hopscotch` only has a few public symbols to be used by other packages. Here's the API.

8.5.1 Registry

The registry is the central part of Hopscotch. It mimics the registry in `wired`, `Pyramid`, and `Zope` (all 3 of which use Zope's registry.)

class hopscotch.**Registry**(*parent=None*, *context=None*)

Type-oriented registry with special features.

Parameters

- **parent** (*Optional[hopscotch.registry.Registry]*) –
- **context** (*Optional[Any]*) –

Return type None

get(*kind*, *context=None*, ***kwargs*)

Find an appropriate kind class and construct an implementation.

The passed-in keyword args act as "props" which have highest-precedence as arguments used in construction.

Parameters

- **kind** (*Type[hopscotch.registry.T]*) –
- **context** (*Optional[Any]*) –
- **kwargs** (*dict[str, Any]*) –

Return type hopscotch.registry.T

get_best_match(*kind*, *context_class=None*, *allow_singletons=True*)

Find the best-match registration, if any.

Using the registry is a two-step process: lookup an implementation, then if needed, construct and return. This is the first part.

Parameters

- **kind** (*Type[hopscotch.registry.T]*) –
- **context_class** (*Optional[Any]*) –
- **allow_singletons** (*bool*) –

Return type *Optional*[*hopscotch.registry.Registration*]

inject(*registration*, *props=None*)

Use injection to construct and return an instance.

Parameters

- **registration** (hopscotch.registry.Registration) –
- **props** (*Optional[dict[str, Any]]*) –

Return type hopscotch.registry.T

register(*implementation*, *, *kind=None*, *context=None*)
Use a LIFO list for all the possible implementations.

Note that the implementation must be a subclass of the kind.

Parameters

- **implementation** (*hopscotch.registry.T*) –
- **kind** (*Optional[Type[hopscotch.registry.T]]*) –
- **context** (*Optional[Any]*) –

Return type None

scan(*pkg=None*)
Look for decorators that need to be registered.

Parameters **pkg** (*Optional[Union[module, str]]*) –

Return type None

setup(*pkg=None*)
Pass the registry to a package which has a setup function.

Parameters **pkg** (*Optional[Union[module, str]]*) –

Return type None

8.5.2 injectable

This decorator provides a convenient way for the `venusian`-based scanner in the registry to recursively look for registrations.

class hopscotch.**injectable**(*kind=None*, *, *context=None*)
`venusian` decorator to register an injectable factory .

Parameters

- **kind** (*Optional[Type[hopscotch.registry.T]]*) –
- **context** (*Optional[Optional[Any]]*) –

8.5.3 inject_callable

Sometimes you want injection without a registry. As an example, `viewdom` works both with and without a registry. For the latter, it does a simpler form of injection, but with many of the same rules and machinery.

class hopscotch.**inject_callable**(*registration*, *props=None*, *registry=None*)
Construct target with or without a registry.

Parameters

- **registration** (hopscotch.registry.Registration) –
- **props** (*Optional[dict[str, Any]]*) –

- **registry** (*Optional[hopscotch.registry.Registry]*) –

Return type hopscotch.registry.T

8.5.4 Registration

When using `inject_callable` directly, you need to make an object with the introspected registration information. This is the object to use.

class hopscotch.**Registration**(*implementation*, *kind=None*, *context=None*, *field_infos=<factory>*, *is_singleton=False*)

Collect registration and introspection info of a target.

Parameters

- **implementation** (*Union[Callable[[...], object], object]*) –
- **kind** (*Optional[Callable[[...], object]]*) –
- **context** (*Optional[Callable[[...], object]]*) –
- **field_infos** (*list[hopscotch.field_infos.FieldInfo]*) –
- **is_singleton** (*bool*) –

Return type None

8.5.5 hopscotch.fixtures

Hopscotch provides some fixtures for use in tests and examples.

DummyOperator

Example objects for tests, examples, and docs.

class hopscotch.fixtures.**DummyOperator**(*arg*)

Simulate an operator that looks something up.

Parameters **arg** (*str*) –

Return type None

Dataclass Examples

Example objects and kinds implemented as dataclasses.

class hopscotch.fixtures.dataklasses.**AnotherGreeting**(*salutation='Another Hello'*)

A replacement alternative for the default `Greeting`.

Parameters **salutation** (*str*) –

Return type None

class hopscotch.fixtures.dataklasses.**Customer**(*first_name*)

The person to greet, stored as the registry context.

Parameters **first_name** (*str*) –

Return type None

class hopscotch.fixtures.dataklasses.**FrenchCustomer**(*first_name*)
A different kind of person to greet, stored as the registry context.

Parameters **first_name** (*str*) –

Return type None

class hopscotch.fixtures.dataklasses.**Greeter**(*greeting*)
A dataclass to engage a customer.

Parameters **greeting** (hopscotch.fixtures.dataklasses.Greeting) –

Return type None

class hopscotch.fixtures.dataklasses.**GreeterAnnotated**(*greeting*)
A dataclass to engage a customer with an annotation.

Parameters **greeting** (hopscotch.fixtures.dataklasses.Greeting) –

Return type None

class hopscotch.fixtures.dataklasses.**GreeterChildren**(*children*)
A dataclass that is passed a tree of VDOM nodes.

Parameters **children** (*tuple[str]*) –

Return type None

class hopscotch.fixtures.dataklasses.**GreeterCustomer**(*customer*)
A dataclass that depends on the registry context.

Parameters **customer** (hopscotch.fixtures.dataklasses.Customer) –

Return type None

class hopscotch.fixtures.dataklasses.**GreeterFirstName**(*customer_name*)
A dataclass that gets an attribute off a dependency.

Parameters **customer_name** (*str*) –

Return type None

class hopscotch.fixtures.dataklasses.**GreeterFrenchCustomer**(*customer*)
A dataclass that depends on a different registry context.

Parameters **customer** (hopscotch.fixtures.dataklasses.FrenchCustomer) –

Return type None

class hopscotch.fixtures.dataklasses.**GreeterGetAnother**(*customer_name*)
Use an operator to change the type hint of what's retrieved.

Parameters **customer_name** (hopscotch.fixtures.dataklasses.AnotherGreeting) –

Return type None

class hopscotch.fixtures.dataklasses.**GreeterKind**(*greeting*)
A dataclass Kind to engage a customer.

Parameters **greeting** (hopscotch.fixtures.dataklasses.Greeting) –

Return type None

class hopscotch.fixtures.dataklasses.**GreeterOptional**(*greeting*)
A dataclass to engage a customer with optional greeting.

Parameters **greeting** (*Optional[*hopscotch.fixtures.dataklasses.Greeting*]*) –

Return type None

class hopscotch.fixtures.dataklasses.**GreeterRegistry**(*registry*)
A dataclass that depends on the registry.

Parameters **registry** (hopscotch.registry.Registry) –

Return type None

class hopscotch.fixtures.dataklasses.**Greeting**(*salutation='Hello'*)
A dataclass to give a greeting.

Parameters **salutation** (*str*) –

Return type None

class hopscotch.fixtures.dataklasses.**GreetingFactory**(*salutation*)
Use the __hopscotch_factory__ protocol to control creation.

Parameters **salutation** (*str*) –

Return type None

class hopscotch.fixtures.dataklasses.**GreetingFieldDefault**(*salutation='Default Argument'*)
A dataclass with a field using a default argument.

Parameters **salutation** (*str*) –

Return type None

class hopscotch.fixtures.dataklasses.**GreetingFieldDefaultFactory**(*salutation=<factory>*)
A dataclass with a field using a default factory.

Parameters **salutation** (*list*) –

Return type None

class hopscotch.fixtures.dataklasses.**GreetingInitFalse**
A dataclass with a field that inits to false.

Return type None

class hopscotch.fixtures.dataklasses.**GreetingNoDefault**(*salutation*)
A dataclass to give a greeting with no default value.

Parameters **salutation** (*str*) –

Return type None

class hopscotch.fixtures.dataklasses.**GreetingOperator**(*greeter*)
A dataclass to give a greeting via an operator.

Parameters **greeter** (hopscotch.fixtures.dataklasses.Greeting) –

Return type None

class hopscotch.fixtures.dataklasses.**GreetingPath**(*location*)
A dataclass to give a builtin Path.

Parameters **location** (*pathlib.Path*) –

Return type None

class hopscotch.fixtures.dataklasses.**GreetingTuple**(*salutation*)
A dataclass to give a sequence of greetings.

Parameters **salutation** (*tuple[str, ...]*) –

Return type None

Function Examples

Example objects and kinds implemented as functions.

hopscotch.fixtures.functions.**Greeter**(*greeting*)
A function to engage a customer.

Parameters **greeting** (*str*) –

Return type str

hopscotch.fixtures.functions.**GreeterAnnotated**(*customer_name*)
A function to engage a customer with an Annotated.

Parameters **customer_name** (*str*) –

Return type str

hopscotch.fixtures.functions.**GreeterChildren**(*children*)
A function that is passed a tree of VDOM nodes.

Parameters **children** (*tuple[str]*) –

Return type tuple[str]

hopscotch.fixtures.functions.**GreeterOptional**(*greeting*)
A function to engage a customer with optional greeting.

Parameters **greeting** (*Optional[str]*) –

Return type *Optional*[str]

hopscotch.fixtures.functions.**Greeting**(*salutation='Hello'*)
A function to give a greeting.

Parameters **salutation** (*str*) –

Return type str

hopscotch.fixtures.functions.**GreetingDefaultNoHint**(*salutation='Hello'*)
A function to with a parameter having no hint.

Return type str

hopscotch.fixtures.functions.**GreetingNoDefault**(*salutation*)
A function to give a greeting without a default.

Parameters **salutation** (*str*) –

Return type str

`NamedTuple` Examples

Example objects and kinds implemented as `NamedTuple`.

Note that, since `typing.NamedTuple` doesn't really do inheritance, we can't implement a `Kind` as a `NamedTuple`.

class hopscotch.fixtures.named_tuples.**Greeter**(*greeting*)
A `NamedTuple` to engage a customer.

Parameters **greeting** (hopscotch.fixtures.named_tuples.Greeting) –

greeting: *hopscotch.fixtures.named_tuples.Greeting*
Alias for field number 0

class hopscotch.fixtures.named_tuples.**GreeterAnnotated**(*greeting*)
A `NamedTuple` to engage a customer with an annotation.

Parameters **greeting** (hopscotch.fixtures.named_tuples.Greeting) –

greeting: *hopscotch.fixtures.named_tuples.Greeting*
Alias for field number 0

class hopscotch.fixtures.named_tuples.**GreeterChildren**(*children*)
A `NamedTuple` that is passed a tree of VDOM nodes.

Parameters **children** (*tuple[str]*) –

children: **tuple[str]**
Alias for field number 0

class hopscotch.fixtures.named_tuples.**GreeterOptional**(*greeting*)
A `NamedTuple` to engage a customer with optional greeting.

Parameters **greeting** (*Optional[*hopscotch.fixtures.named_tuples.Greeting*]*) –

greeting: **Optional[***hopscotch.fixtures.named_tuples.Greeting***]**
Alias for field number 0

class hopscotch.fixtures.named_tuples.**Greeting**(*salutation='Hello'*)
A `NamedTuple` to give a greeting.

Parameters **salutation** (*str*) –

salutation: **str**
Alias for field number 0

class hopscotch.fixtures.named_tuples.**GreetingNoDefault**(*salutation*)
A `NamedTuple` to give a greeting without a default.

Parameters **salutation** (*str*) –

salutation: **str**
Alias for field number 0

Plain Old Class Examples

Example objects and kinds implemented as plain classes.

class hopscotch.fixtures.plain_classes.**Greeter**(*greeting*)
A plain-old-class to engage a customer.

Parameters **greeting** (hopscotch.fixtures.plain_classes.Greeting) –

class hopscotch.fixtures.plain_classes.**GreeterAnnotated**(*greeting*)
A plain-old-class to engage a customer with an annotation.

Parameters **greeting** (hopscotch.fixtures.plain_classes.Greeting) –

class hopscotch.fixtures.plain_classes.**GreeterChildren**(*children*)
A plain-old-class that is passed a tree of VDOM nodes.

Parameters **children** (*tuple[str]*) –

class hopscotch.fixtures.plain_classes.**GreeterKind**(*greeting*)
A plain-old-class Kind to engage a customer.

Parameters **greeting** (hopscotch.fixtures.plain_classes.Greeting) –

class hopscotch.fixtures.plain_classes.**GreeterOptional**(*greeting*)
A plain-old-class to engage a customer with optional greeting.

Parameters **greeting** (*Optional[hopscotch.fixtures.plain_classes.Greeting]*) –

class hopscotch.fixtures.plain_classes.**Greeting**
A plain-old-class to give a greeting.

class hopscotch.fixtures.plain_classes.**GreetingNoDefault**(*salutation*)
A plain-old-class to give a greeting without a default.

Parameters **salutation** (*str*) –

8.6 Contributor Guide

Thank you for your interest in improving this project. This project is open-source under the MIT license and welcomes contributions in the form of bug reports, feature requests, and pull requests.

Here is a list of important resources for contributors:

- Source Code
- Documentation
- Issue Tracker
- *Code of Conduct*

8.6.1 How to report a bug

Report bugs on the Issue Tracker.

When filing an issue, make sure to answer these questions:

- Which operating system and Python version are you using?
- Which version of this project are you using?
- What did you do?
- What did you expect to see?
- What did you see instead?

The best way to get your bug fixed is to provide a test case, and/or steps to reproduce the issue.

8.6.2 How to request a feature

Request features on the Issue Tracker.

8.6.3 How to set up your development environment

You need Python 3.6+ and the following tools:

- Poetry
- Nox
- nox-poetry

Install the package with development requirements:

```
$ poetry install
```

You can now run an interactive Python session, or the command-line interface:

```
$ poetry run python
$ poetry run hopscotch
```

8.6.4 How to test the project

Run the full test suite:

```
$ nox
```

List the available Nox sessions:

```
$ nox --list-sessions
```

You can also run a specific Nox session. For example, invoke the unit test suite like this:

```
$ nox --session=tests
```

Unit tests are located in the `tests` directory, and are written using the pytest testing framework.

8.6.5 How to submit changes

Open a pull request to submit changes to this project.

Your pull request needs to meet the following guidelines for acceptance:

- The Nox test suite must pass without errors and warnings.
- Include unit tests. This project maintains 100% code coverage.
- If your changes add functionality, update the documentation accordingly.

Feel free to submit early, though—we can always iterate on this.

To run linting and code formatting checks before commiting your change, you can install pre-commit as a Git hook by running the following command:

```
$ nox --session=pre-commit -- install
```

It is recommended to open an issue before starting work on anything. This will allow a chance to talk it over with the owners and validate your approach.

8.7 Contributor Covenant Code of Conduct

8.7.1 Our Pledge

We as members, contributors, and leaders pledge to make participation in our community a harassment-free experience for everyone, regardless of age, body size, visible or invisible disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

We pledge to act and interact in ways that contribute to an open, welcoming, diverse, inclusive, and healthy community.

8.7.2 Our Standards

Examples of behavior that contributes to a positive environment for our community include:

- Demonstrating empathy and kindness toward other people
- Being respectful of differing opinions, viewpoints, and experiences
- Giving and gracefully accepting constructive feedback
- Accepting responsibility and apologizing to those affected by our mistakes, and learning from the experience
- Focusing on what is best not just for us as individuals, but for the overall community

Examples of unacceptable behavior include:

- The use of sexualized language or imagery, and sexual attention or advances of any kind
- Trolling, insulting or derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or email address, without their explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

8.7.3 Enforcement Responsibilities

Community leaders are responsible for clarifying and enforcing our standards of acceptable behavior and will take appropriate and fair corrective action in response to any behavior that they deem inappropriate, threatening, offensive, or harmful.

Community leaders have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

8.7.4 Scope

This Code of Conduct applies within all community spaces, and also applies when an individual is officially representing the community in public spaces. Examples of representing our community include using an official e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event.

8.7.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the community leaders responsible for enforcement at pauleveritt@me.com. All complaints will be reviewed and investigated promptly and fairly.

All community leaders are obligated to respect the privacy and security of the reporter of any incident.

8.7.6 Enforcement Guidelines

Community leaders will follow these Community Impact Guidelines in determining the consequences for any action they deem in violation of this Code of Conduct:

1. Correction

Community Impact: Use of inappropriate language or other behavior deemed unprofessional or unwelcome in the community.

Consequence: A private, written warning from community leaders, providing clarity around the nature of the violation and an explanation of why the behavior was inappropriate. A public apology may be requested.

2. Warning

Community Impact: A violation through a single incident or series of actions.

Consequence: A warning with consequences for continued behavior. No interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, for a specified period of time. This includes avoiding interactions in community spaces as well as external channels like social media. Violating these terms may lead to a temporary or permanent ban.

3. Temporary Ban

Community Impact: A serious violation of community standards, including sustained inappropriate behavior.

Consequence: A temporary ban from any sort of interaction or public communication with the community for a specified period of time. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

4. Permanent Ban

Community Impact: Demonstrating a pattern of violation of community standards, including sustained inappropriate behavior, harassment of an individual, or aggression toward or disparagement of classes of individuals.

Consequence: A permanent ban from any sort of public interaction within the community.

8.7.7 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 2.0, available at https://www.contributor-covenant.org/version/2/0/code_of_conduct.html.

Community Impact Guidelines were inspired by Mozilla's code of conduct enforcement ladder.

For answers to common questions about this code of conduct, see the FAQ at https://www.contributor-covenant.org/faq. Translations are available at https://www.contributor-covenant.org/translations.

8.8 MIT License

PYTHON MODULE INDEX

h

INDEX

H

I

M

R

S