Findig Documentation

Release 0.1.0

Te-je Rodgers

July 18, 2015

Contents

Findig is a micro-framework for developing HTTP applications in Python. It is built on Werkzeug and is intended as an alternative to Flask when developing RESTful APIs.

The documentation is split into two parts: the user guide (which includes installation and quickstart instructions) and detailed API documentation.

Note: The documentation here isn't complete yet. If you stumble across something that you think should be documented, but isn't, please feel free to let us know about it on our issue tracker.

You may also clone the Github Repository and submit pull requests with documentation fixes for review.

Findig uses Werkzeug internally, and so some understanding of how Werkzeug works and its data types, while not necessary, will almost certainly be helpful:

- Werkzeug documentation.

CHAPTER 1

User Guide

This part of the documentation is a comprehensive guide for getting things done with Findig, as well as some notes on its design.

1.1 Introduction

This section gives a brief introduction to what Findig is, its design philosophy, and what to expect when using it.

1.1.1 What Findig is

Findig is a micro-framework for building HTTP applications. It's perhaps comparable to Flask and Bottle in terms of size and ease of use. Continuing in their tradition, it makes it incredibly easy to set up a WSGI application, and avoids forcing specific choices on you (such as a database layer or templating engine). However, Findig is geared specifically toward building RESTful web applications, so much so that serving HTML for web browsers with Findig would be incredibly counter-intuitive.

Where traditional frameworks typically describe web applications in terms of views (or pages) and routes, Findig applications are described in terms of resources and CRUD actions, and the actual generation of views is done behind the scenes. Here's an example of what an application that serves a simple JSON API looks like in Findig:

```
from findig.json import App
from dbstack.users import get, save, delete

app = JSONApp()

@app.route("/users/<int:id>")
def user(id):
    user = get(id)
    return user.as_dict()

@user.model("write")
def user(data, id):
    save(id, data)
    return get(id).as_dict()

@user.model("delete")
def user(id):
    delete(id)
```

```
# app is a WSGI callable that can be run by your
# WSGI server of choice
```

This code will accept and respond to application/json `GET|PUT|DELETE` requests sent to `/users/:id` on the server, as long as `id` looks like an integer. Notice how rather than describing the resource in terms of how it looks (i.e., a view), the resource is described in terms of how to read, write and delete its data. It's implied that Findig will use this data to organize the views and construct the responses. Behind the scenes the *findig.json.App* converts the resource data into a JSON formatted response.

1.1.2 A word on customization

As seen above, Findig lets you describe how to manipulate resource data and then runs along and does everything else to handle requests (including parsing input and formatting output). If that sounds a little heavy on the fascism, don't despair; it's actually really customizable.

Input parsing, output formatting and error handling can be globally or selectively overridden in all Findig applications. For example, if we wanted to also output to XML based on an accept header, we could add the following code:

```
@user.formatter.register("application/xml")
def format_xml_response(user_dict):
        # This registers an xml formatter for the user resource
        xml = generate_user_xml_by_some_means(user_dict)
        return xml
```

In addition, a lot of Findig's internals are based on abstract classes, allowing you to plug in custom components that do things just the way you want them done (let's hope it never comes to that, though).

1.1.3 When to use (and not to use) Findig

Note: Findig is plain and simple pre-release software. This section right now is largely irrelevant for production applications: *you simply don't*.

Findig is an API framework that's intended for building APIs, and it aims to be great at doing only that. As a result, support for applications that talk to traditional web browsers is virtually non-existent. Instead, use Findig if you want to build a backend API that powers your web apps and/or mobile applications.

Note: Findig can be used to back angularjs apps.

1.2 Installation

Note: Findig is pre-release software and isn't currently installable by PyPI. Currently the only way to install Findig is using its development version.

Installing Findig is easy, as long as you have the right tool for the job. Make sure you have pip installed before proceeding.

1.2.1 Supported Python versions

Findig currently supports Python 3.4.

1.2.2 Installing the development version

You can install the development version of Findig straight from Github. Just run the following pip command (make sure you have git installed on your system):

```
$ pip install git+https://github.com/geniphi/findig.git#egg=Findig
```

This will install the latest version of Findig on your system.

1.2.3 Getting extra features

Findig has some extra features that are enabled when certain packages are installed on the system. They are defined as extras that can be specified when installing Findig. To install an extra feature, just include its name inside square brackets immediately after 'Findig' in your pip install command. Multiple extra features can be installed by listing them inside square brackets, separated by commas. For example, to install the development version of Findig with redis cache support, run this command:

```
$ pip install git+https://github.com/geniphi/findig.git#egg=Findig[redis]
```

And pip will install Findig along with all the requirements necessary for the extra feature to work.

Here's a list of all the supported extra features:

Feature name	Description	Installed Requirements
redis	Redis data set support	redis

1.2.4 Getting the source code

The source code for Findig is hosted on Github. If you have Git installed, you can clone the repository straight to your hard drive from a command shell:

```
$ git clone git://github.com/geniphi/findig.git
```

Alternatively, you can download a source tarball or zipball, both of which will contain the latest source code from the repository.

1.3 Quickstart

Ready to get started? This section gives you a quick introduction to using Findig and it's basic patterns.

1.3.1 The tiniest JSON application

Here's the smallest app you can write in Findig:

```
from findig.json import App
from werkzeug.serving import run_simple

app = App(indent=4, autolist=True)

if __name__ == '__main__':
    run_simple('localhost', 5000, app)
```

This barebones app exposes a list of resources that are exposed by your app, and the HTTP methods that they support. Save it as *listr.py*, and run as follows:

```
$ python listr.py
 * Running on http://localhost:5000/ (Press CTRL+C to quit)
```

Now, send a GET request to the root of your app:

```
$ curl http://localhost:5000/
[
    {
        "methods": [
            "HEAD",
            "GET"
        ],
        "url": "/",
        "is_strict_collection": false
    }
]
```

We see that the response is a singleton list with an autogenerated JSON resource. That single item is the same one we just queried, and it was added by the `autolist=True` argument to our application. That tells the application to create a resource at the server root that lists all of the resources available on the API. Since we haven't added any other resources, this is the only one available.

1.3.2 Adding resources

To add a resource, tell the app how to get the resource's data, and what URLs route to it:

```
@app.route("/resource")
def resource():
    return { "msg": "Hello World!" }
```

What we've defined here is a pretty useless static resource, but it's okay because it's just for illustrative purposes. Typically, each resource is defined in terms of a function that gets the data associated with it, here we've aptly called ours `resource`. The `app.route()` decorator factory takes a URL rule specification and registers our resource at any URL that matches.

Hint: So what type should your resource function return? Well, Findig doesn't actually have any specific restrictions. If you're working with a JSON app though, you should probably stick to Python mappings and iterables.

Other types can by used seamlessly with custom formatters.

The code above is actually shorthand for the following:

```
@app.route("/resource")
@app.resource
def resource():
    return { "msg": "Hello World!" }
```

`app.resource` takes a resource function and turns it into a *Resource*, which can still be called as though they are resource functions.

You can use arguments to customize resource creation:

```
@app.route("/resource")
@app.resource(name="my-super-special-resource")
```

```
def resource():
    return {"msg": "Hello, from {}".format(resource.name)}
```

Resource names are unique strings that identify the resource somehow. By default, Findig will try to generate one for you, but that can be overridden if you want your resources to follow a particular naming scheme (or if you want to treat two resources as the same, by giving them the same name).

See `finding.resource.Resource` for a full listing of arguments that you can pass to resources.

Note: While you can omit `@app.resource` if your resource doesn't need any arguments, you shouldn't ever omit `@app.route` unless you really intend to have a resource that can never be reached by the outside world!

1.3.3 Collections

Some resources can be designated as collections of other resources. Resource instances have a special decorator to help you set this up:

```
@app.route("/people/<int:idx>")
def person(idx):
    return people()[idx-1]

@app.route("/people/")
@person.collection
def people():
    return [
        {"name": "John Doe", "age": 40},
        {"name": "Jane Smith", "age": 34}
    ]
```

What's going on here? Well, we've defined a `person` resource that routes to a strange looking URL. Actually, `/people/<int:idx>` is a URL rule specification; any URL that matches it will route to this resource. The angle brackets indicate a variable part of the URL. It includes an optional converter, and the name of the variable part. This spec will match the URLs `/people/0`, `/people/1`, `people/2` etc, but not `/people/anthony` (because we specfied an int converter; to match ordinary strings, just omit the converter: `people/<idx>`). A URL spec can have as many variable parts as needed, however the resource function must take a named parameter matching **each** of the variable parts.

Next, we define a `people` resource that's a collection of `person` using `person.collection` as a decorator. The resource functions for collections are expected to return iterables that contain representations of the contained items.

Like resources, collections can take arguments too:

```
@app.route("/people/")
@person.collection(include_urls=True)
def people():
    return [
        {"name": "John Doe", "age": 40, "idx": 1},
        {"name": "Jane Smith", "age": 34, "idx": 2}
    ]
```

The `include_urls=True` instructs the collection to insert a URL field in the generated items that points to that specific item on the API server. The only caveat is that the item data that we return from the collection's resource function has to have enough information contained to build a URL for the item (that's why we added the idx field here).

See `finding.resource.Collection` for a full listing of arguments that you can pass to collections.

1.3.4 Data operations

The HTTP methods that Findig will expose depends on the data operations that you've defined for your resource. By default, `GET` operations are exposed for every resource, since we have to define resource functions that get the resources's data. Makes sense right?

But what about the other HTTP methods? We can support `PUT` requests by telling Findig how to write new resource data:

```
@resource.model("write")
def write_new_data(data):
    # Er, we don't have a database set up, so let's just complain
    # that it's poorly formatted.
    from werkzeug.exceptions import BadRequest
    raise BadRequest
```

That `.model()` function is actually a function available by default on *`Resource`* instances that lets you provide functions that manipulate resource data. Here, we're specifying a function that writes new data for the resource, and its only argument is the new data that should be written, taken from the request body. Here's a complete list of data operations that you can add, and what they should do:

Operation	Arguments	Description
write	data	Replaces completely the data for the resource, and enables `PUT` requests on the resource.
make	data	Creates a new child resource with the input data. It should return a mapping of values that can be used to route to the resource. If present, it enables `POST` requests.
delete		Delete's the resource's data and enables `DELETE` requests on the resource.

1.3.5 Restricting HTTP Methods

Sometimes, your might define more data operations for a resource than you want directly exposed on the API. You can restrict the HTTP methods for a resource through it's route:

```
@app.route("/resource", methods=['GET', 'DELETE'])
def resource():
    # return some resource data
    pass

@resource.model('write')
def write_resource(data):
    # save the resource data
    pass

@resource.model('delete')
def delete_resource():
    # delete the resource
    pass
```

`PUT` requests to this resource will fail with status `405: METHOD NOT ALLOWED`, even though we have a *write* operation defined.

1.3.6 Custom applications

Suppose you wanted to build an API that *wasn't* JSON (hey, I'm not here to judge)? That's entirely possible. You just have to tell Findig how to convert to and from the content-types that you plan to use.

```
from findig import App

app = App()

@app.formatter.register("text/xml")
def convert_to_xml(data):
    s = to_xml(data)
    return s # always return a string

@app.parser.register("text/xml")
def convert_from_xml(s)
    obj = from_xml(s)
    return obj
```

Pretty straightforward stuff; `convert_to_xml` is a function that takes resource data and converts it to an xml string. We register it as the data formatter for the `text/xml` content-type using the `@app.formatter.register("text/xml")` decorator. Whenever a client sends an `Accept` header with the `text/xml` content-type, this formatter will be used. Similarly, `convert_from_xml` converts an xml string to resource data, and is called when a request with a `text/xml` content-type is received.

That's great, but what happens if the client doesn't send an `Accept` header, or if it sends request content without a content-type? Well, Findig will send a `text/plain` response (it calls *str* on the resource data; hardly elegant) in the first case, and send back an `UNSUPPORTED MEDIA TYPE` error in the second case. To avoid this, you can set a default content-type that is assumed if the client doesn't specify one. Here's the same example from above setting `text/xml` as the default:

```
from findig import App

app = App()

@app.formatter.register("text/xml", default=True)
def convert_to_xml(data):
    s = to_xml(data)
    return s # always return a string

@app.parser.register("text/xml", default=True)
def convert_from_xml(s)
    obj = from_xml(s)
    return obj
```

Tip: *findig.json.App* does this for the `application/json` content-type.

An application can register as many parsers and formatters as it needs, and can even register them on specific resources. Here's how:

```
from pickle import dumps
from findig import App

app = App()
app.formatter.register("x-application/python-pickle", dumps, default=True)

@app.route("/my-resource")
def resource():
    return {
        "name": "Jon",
        "age": 23,
    }
```

```
@resource.formatter.register("text/xml")
def format_resource(data):
    return "<resource><name>{}</name><age>{}</age></resource>".format(
        data['name'],
        data['age']
    )
```

So this app has a global formatter that pickles resources and returns them to the client (look, it's just an example, okay?). However, it has a special resource that can handle `text/xml` responses as well, using the resource-specific formatter that we defined.

1.4 taskman — A tutorial

Our tutorial is a simple API that lets you create tasks, edit them, and mark them as finished.

1.4.1 The basic application

Let's go ahead and create our app and declare our resources. They won't do anything for now, but will help us get a firm idea of how the API is laid out.

Add the following to a file and save it as *taskman.py*

```
from findig.json import App

app = App()

@app.route("/tasks/<id>")
@app.resource
def task(id):
    return {}

@app.route("/tasks/")
@task.collection
def tasks():
    return []
```

We want our API to serve JSON resources, so we've imported and initialized *findig.json.App*. We could've supported a different data format, but we'd have to write the code to convert the resources ourselves, since Findig only includes converters for JSON. For this tutorial we'll just stick to JSON (it's pretty darn good!).

Next, we declare a resource called `task`. This resource is going to represent a single task, and provide an interface for us to delete and update individual tasks. There's quite a bit going on here:

- `@app.route("/tasks/<id>")` assigns a URL rule to our task resource. The URL rule tells findig what sort of URLs should route to our resource. The second part of the rule is interesting; by enclosing `id` in angle brackets, we've created a variable part of the URL rule. When matching a URL, Findig will match the static part of the rule exactly, but try to match other parts of the URL to the variables. So for example, `/tasks/foo` and `/tasks/38` will route to this resource, using `'foo'` and `'38'` as the id (variable part) respectively. This kinda important, because we can use just one resource to group all of our tasks together, but use the variable part URL to identify the exact task that we're referring to. By the way, here's the documentation for *app.route*.
- `@app.resource` declares a resource. *app.resource* wraps a resource function and returns a *findig.resource.Resource*. This means that in our code, `task` will be replaced by a Resource object (Resources act like functions, so you can still call it if you want to).

- `task(id)` is our resource function for a task. Notice that it takes an argument called `id`. This is because of our URL rule; for every variable part in a URL rule, a resource function must accept a named argument (keyword arguments work too) that matches the variable part's name. So when task is called by Findig, it will be called with an id that's extracted from the URL. Resource functions are supposed to return the resource's data (in other words, what should be sent back on a GET request). Our resource function returns an empty dictionary. That's because the JSON converters know how to work with dictionaries; if we were to send a GET request to any task right now (example: `GET /tasks/foo`) we would receive an empty JSON object as the response.

The next block declares our `tasks` collection. We'll use it to get a list of all of our tasks, and to add new tasks to that list. Bit by bit, here's what is going on in that code:

- `@app.route("/tasks/")` should be familiar. We're once again assigning a URL rule, only this time there are no variable parts so it will match only one URL (`/tasks/`).
- `@task.collection` declares tasks as a collection containing `task` resource instances. Remember how `app.resource` turned our resource function into a *`Resource`*? Well that's where the collection function comes from. Here, *`collection()`* wraps a resource function and turns it into a *`Collection`* (which are callable, just like a Resource—in fact, they *are* Resources).
- `tasks()` is our resource function for the tasks collection. We return an empty list because the JSON converters know how to turn an iterable into a JSON list; if we sent `GET /tasks` to our API, we'd get an empty JSON list as the response.

1.4.2 Serving it up

If we were deploying a production application, we'd be using a web server like Apache to serve up our API. But since this is a measly tutorial, we can get away with using Werkzeug's built-in development server (see that 'development' in front of server? It means you should only use it during development ;)).

Go ahead and add this to the bottom of *taskman.py*:

```
if __name__ == '__main__':
    from werkzeug.serving import run_simple
    run_simple("localhost", 5000, app, use_reloader=True, use_debugger=False)
```

This serves up our application on port 5000 on the local machine. `use_reloader=True` is a handy setting that reloads the application anytime your change the source file. You might be tempted to set `use_debugger=True`, but don't; we set it to `False` (the default) deliberately to make the point that since the werkzeug debugger is tailored for HTML, it is almost certainly useless for debugging a Findig app.

1.4.3 Adding our data models

We're going to need to store our tasks somewhere. Findig uses data models to figure out how to interface with stored resource data. This section is a little long-winded, because it presents the roundabout way of declaring models, and then promptly throws all of that away and uses a shorter method instead (don't hate, okay? This is still a tutorial, so it's important for you to grasp the underlying concepts).

Explicit data models

We can declare data model functions to instruct Findig on how to access stored data for *each* resource. Whenever we do that, we're using explicit data models. That's what we'll cover in this section.

We won't use any of the code we add in this section in our final application, but it's important that we go through it anyway so that you grasp the underlying concepts. If you don't care for any of that, you can probably skip ahead to *Data sets* (but don't blame me if you don't understand how they work!).

Let's start with the `task` resource. Remember that we want to use that resource to update and delete individual tasks. Add this code to *taskman.py*

```
TASKS = []

@task.model("write")
def write_task(data):
    TASKS[task.id] = data

@task.model("delete")
def delete_task():
    del TASKS[task.id]
```

`TASKS = []` sets up a global module-level list that tracks all of our tasks. Since this is throwaway code anyway, there's no harm in storing our tasks in memory like this; it'll never really get used! Were this a production application, then you would be fully expected to use a more responsible data storage backend. And if you didn't, well, you'd just have to face the consequences, wouldn't you?

Now, the first interesting thing happening here is the `@task.model("write")` declaration. This is declaring a write_tasks as a function that can write new data for a specific task. It gets passed a mapping of fields, directly converted from data send by the requesting client. The next interesting thing is `task.id`. During a request to our task resource, `task.id` will bind to the value of the `id` URL variable.

Tip: Anytime a URL rule with variable parts is used to route to a resource, Findig binds the values of those variables to the resource for the duration of the request. This binding is completely context safe, meaning that even when requests are running on multiple threads, `{resource}.{var}` will always bind to the correct value.

Similarly, `task.model("delete")` declares delete_task as a function that deletes a task. Delete model functions don't take any arguments.

Whenever we introduce model functions, Findig will usually enable additional request methods which correspond somewhat to the model functions. This table gives the model functions, their signatures, and corresponding request methods:

Model function	Request method	Supported resource type
`write(data:mapping)`	PUT	*`Resource`*
`delete()`	DELETE	*`Resource`*
`make(data:mapping) -> token`	POST	*`Collection`*

You may notice that a "make" model is to be attached to a *`Collection`* rather than a *`Resource`*. It must however, create a resource instance of the *`Resource`* that the collection collects. The token returned from the "make" model function is a special mapping with enough data to identify the resource instance that was created. By default, you should make sure that it has at least the same fields as the arguments to the resource instance's resource function.

Anyway, from the table, you should be able to see that our `task` resource now supports `PUT` and `DELETE` requests. Go ahead and test them out (Remember to send `application/json` content with your PUT requests)!

But wait, we're still not done. Remember that `GET /tasks/<id>` still always returns an empty JSON object, no matter if we've already `PUT` a task there. We need to fix that by updating the resource function to return the appropriate task data; change you definition of *task* to look like this:

```
@app.route("/tasks/<id>")
@app.resource
def task(id):
    return TASKS[id]
```

But what if we get a URL that includes an id that is not in TASKS? That's okay! Findig automatically converts a `LookupError` into an HTTP 404 response. So when the invalid id throws a `KeyError` (a `LookupError` subclass), it won't crash; it'll tell the requesting client that it doesn't know what task it's asking about. Of course, if you're

still not convinced, you can go ahead and catch that `KeyError` and raise a `werkzeug.exceptions.NotFound` error yourself.

Next, we'll add the model for `tasks`:

```
@tasks.model("make")
def make_task(data):
    token = {"id": str(uuid4())}
    TASKS[token['id']] = data
    return token
```

update the resource function for our `tasks` collection:

```
@app.route("/tasks/")
@task.collection
def tasks():
    return TASKS
```

and finally add the following import to the top of the file:

```
from uuid import uuid4
```

Not a whole lot new is going on; In our "make" model, we're using the built-in `uuid.uuid4()` function to generate random ids for our tasks (nobody ever said our ids had to be numeric!), and we're storing the data receive with that id. Finally, we return the id as part of the token (remember that the token needs to contain at least enough data to identify the task instance, and here, all we need is id!).

And that's it! We've built out our explicit data model. Now, let's go throw it away...

Data sets

Data sets are an alternative to explicit data model functions. They have the advantage of being far less verbose, but aren't quite as flexible. However, for most resources, you may find that you don't need that extra bit of flexibility, so a data set is perfectly fine.

Essentially, a data set is a special collection of records, each corresponding to a single resource instance. Instead of returning a straight-up list from a collection resource function, we can return a data set instead. Since a data-set is already an iterable object, we don't actually lose anything by dropping one in where we would normally return a list or a generator. However, with a little coaxing, we can get Findig to inspect the data set and derive a model from it, so you don't have to type one out.

We're going to be using the included `findig.extras.sql.SQLASet` (which requires SQLAlchemy) with an SQLite table for our tasks. There's also a *`findig.extras.redis.RedisSet`*, but it relies on a redis server which you may not have on your system (that's a little beyond the scope of this tutorial). Unlike RedisSet, SQLASet does require a table schema to be declared, so the code is a little more verbose.

Let's dig in! Add this to *taskman.py* right after your app initialization:

```
db = SQLA("sqlite:///tasks.sqlite", app=app)
validator = Validator(app)


class Task(db.Base):
    id = Column(Integer, primary_key=True)
    title = Column(String(150), nullable=False)
    desc = Column(String, nullable=True)
    due = Column(DateTime, nullable=False)
```

and add the following imports:

```
from sqlalchemy.schema import *
from sqlalchemy.types import *
from findig.extras.sql import SQLA, SQLASet
```

Tip: If the above import gives you an `ImportError`, it means that you don't have `SQLAlchemy` installed. You'll need to install it to continue (try: `pip install sqlalchemy` in your shell, if you have pip).

All we've done here is declare an SQLAlchemy orm schema. `findig.extras.sql.SQLA` is a helper class for using SQLAlchemy inside a findig application. The first argument we pass here sets up the database engine (we store them in an SQLite database called 'tasks.sqlite'; you'll need to make sure that your application process has write permission to the working directory so that it can create that file), and we pass our app as a keyword argument.

After that, we declare our tasks table and its ORM mapping. We set up our schema with three columns (id, title and desc).

Next up, let's use that schema to create an SQLA data set. Replace the declaration for your tasks collection with this code:

```
@app.route("/tasks/")
@task.collection(lazy=True)
def tasks():
    return SQLASet(Task)
```

So some interesting changes. First up, we've added the `lazy=True` argument to `task.collection`. This gives Findig the heads-up that this resource function returns a data set (meaning that simply calling it does not make any queries to the database). As a result, Findig is able to inspect the return value when setting things up. Since it is a data set, Findig uses that to add our model functions for us.

To complete our transition, replace the resource declaration for `task` with this code:

```
@app.route("/tasks/<id>")
@app.resource(lazy=True)
def task(id):
    return tasks().fetch(id=id)
```

`findig.extras.sql.SQLASet.fetch()` can be thought of as a query. It returns the first matching item as a *MutableRecord*, which Findig also knows how to extract data model functions from.

As a result, we don't need our data model functions anymore, so you should go ahead and delete them.

1.4.4 Validating data

At this point, we've developed a working web application, but it's still incomplete. Why, you ask? Because we haven't actually put an constraints on the data that we receive. As it stands, we could send any old data to our API and get away with it, without a peep in protest from the application.

Note: Well that's not strictly true; since we've added an SQL schema for the tasks, SQLASet will try to make sure that any data that it receives conforms to the schema at the very least. Still, it doesn't perform any checks on the actual values or do any type conversions for us, and so we need to do that ourselves.

So what sort of constraints are we talking? Let's look at the fields in our task schema again, to get a better idea:

```
validator = Validator(app)


class Task(db.Base):
    id = Column(Integer, primary_key=True)
```

```
    title = Column(String(150), nullable=False)
    desc = Column(String, nullable=True)
    due = Column(DateTime, nullable=False)
```

First, let's look at the `id` field. This field is an integer primary key, so the engine will automatically generate one for us (as it does with all integer primary key fields :D); we don't need to put any constraints here because we won't be asking for one from the client.

Next there is the `title` field. It's a string with a maximum length of 150 characters. It's tempting to have our validation engine enforce this for us, but if we pass a string longer than 150 characters to the database engine, it will truncate it for us. I think that's a reasonable compromise. We also see that the field is marked as `nullable=False`; this meaans that it is required.

Our `desc` field doesn't have much in the way of constraints; it's probably okay to just let our user put any old thing in there.

Finally, our `due` field is meant to store a date/time. We should make sure that whatever we receive from the client for 'due' can be parsed into a date/time. Also, note that this field is also required.

Great! So let's go set all of this up. The first thing we need to do is create a *Validator* for our application. Add this code right after you initialize your app:

```
validator = Validator(app)
```

and add this import:

```
from findig.tools.validator import Validator
```

Next up, let's use the validator to enforce the constraints that we've identified. First up, I think it's a good idea to make sure that we don't get any extra fields. We can do that by adding this decorator at the top of our resource declaration for `task`:

```
@validator.restrict("desc", "*due", "*title")
```

So what's happening? We're telling the validator to only accept the fields `desc`, `due` and `title`. But what's with the `*`? If you guessed that the field is required, you're right! *restrict()* accepts a variable number of field names, so we can restrict our resource input data to any number of fields we want.

Tip: We've only set a validation rule for `task`, but what about `tasks`? Since `tasks` is a collection of `task` instances, the validator will check input for `tasks` with the rules we've defined for `task` by default. If you want to disable this behavior, you can pass `include_collections=False` to the validator constructor.

All we have to do now is check that the due date is date/time string, and parse it into a `datetime` object. With *Validator.enforce*, we can supply a converter for the due field. A converter can by a simple type, a magic string, or an application-defined function that takes a string as input and returns the parsed output. Here's what such a function can look like for a date/time field like `due`:

```
def convert_date(string):
    import datetime
    format = "%Y-%m-%d %H:%M:%S%z"
    return datetime.datetime.strptime(string, format)
```

In fact, *Validator.date* is a static method that simplifies this pattern; it takes a date/time format as its argument and returns a converter function that parses a `datetime` object using that format. That's what we'll use to check our `due` field. Add this decorator to our task declaration:

```
@validator.enforce(due=validator.date("%Y-%m-%d %H:%M:%S%z"))
```

With that, we've set up validation for our request input. You should go ahead and try sending requests to the API we've created.

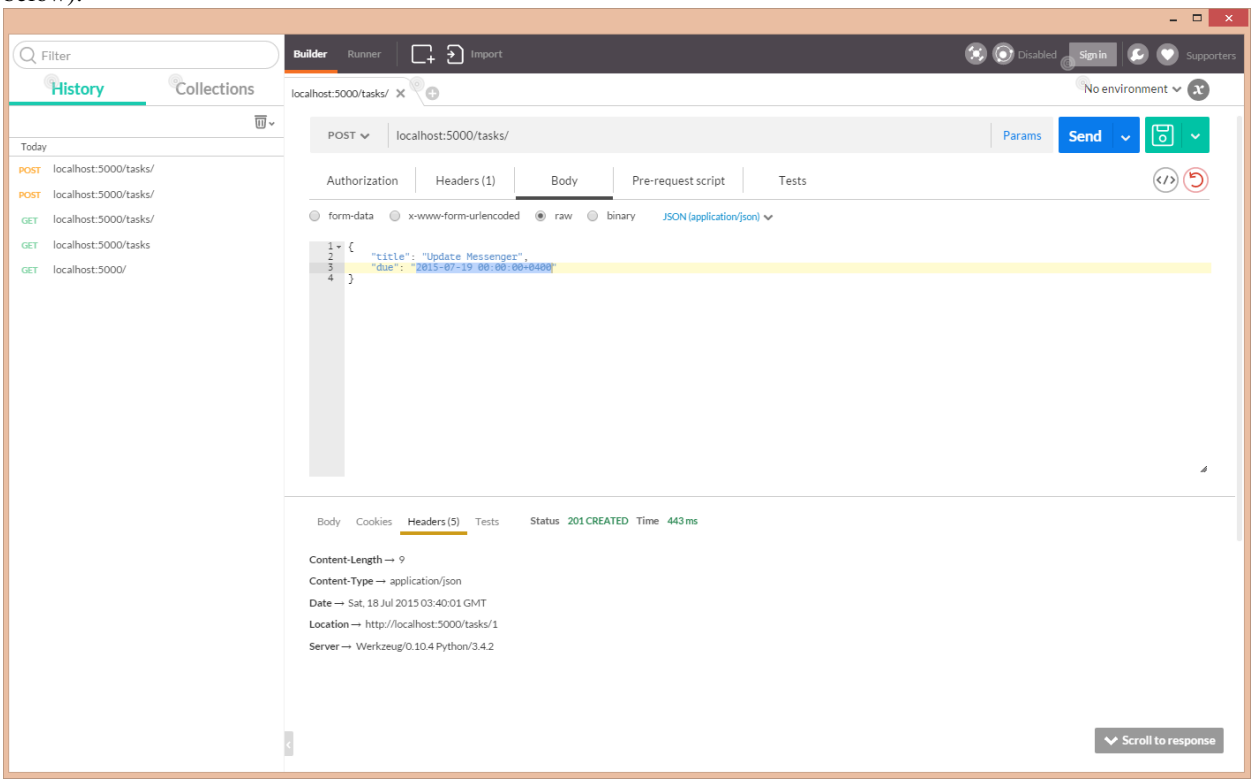
1.4.5 Calling our API

By now, we have a pretty decent API, but how exactly do we use it? First, let's start our development server:

```
$ python taskman.py
 * Running on http://localhost:5000/ (Press CTRL+C to quit)
 * Restarting with stat
```

Our development server is running on port 5000. Calling our API is a matter of sending `application/json` HTTP requests to the server. For testing, you'll need a program that can send `application/json` requests, since your browser probably doesn't provide an interface for this. The examples in this section will use the command-line tool cURL, but they'll include all the details your need to send the requests with any tool you prefer to use.

Tip: If you prefer a graphical interface, you might want to try the Postman Google Chrome extension (pictured below).



Listing our tasks

Send a `GET` request to `/tasks/`. Here's how to do it in cURL:

```
$ curl localhost:5000/tasks/
[]
```

The response is a JSON list containing all of the tasks that we have created. Since we haven't created any yet, we get an empty list.

Creating a new task

To create a new task, we send a `POST` request to `/tasks/`. The request should have a `Content-Type: application/json` header, and the request body must be a JSON object containing the attributes for our new task:

```
$ curl -i -X POST -H "Content-Type: application/json" -d '{"title": "My Task"}' localhost:5000/tasks/
HTTP/1.0 400 BAD REQUEST
Content-Type: application/json
Content-Length: 91
Server: Werkzeug/0.10.4 Python/3.4.2
Date: Sat, 18 Jul 2015 03:33:58 GMT

{"message": "The browser (or proxy) sent a request that this server could not understand."}
```

What's with the error here? Well, remember that we've set up a validator for our `tasks` resource to require a 'due' field with a parseable date/time. Let's modify our request to include one:

```
$ curl -i -X POST -H "Content-Type: application/json" -d '{"title": "My Task", "due": "2015-07-19 00:
HTTP/1.0 201 CREATED
Content-Length: 9
Content-Type: application/json
Date: Sat, 18 Jul 2015 03:40:01 GMT
Location: http://localhost:5000/tasks/1
Server: Werkzeug/0.10.4 Python/3.4.2

{"id": 1}
```

Notably, the status code returned is `201 CREATED` and *not* `200 OK`. Additionally, Findig will try to fill the `Location` header, as long as the data returned from the collection resource function is enough to build a URL for the created resource instance. Our resource function uses `SQLASet`, which returns the primary key fields.

Editing a task

For this one, we send a `PUT` request to the task URL. Just like when creating a task, The request should have a `Content-Type: application/json` header, and the request body must be a JSON object containing the attributes for our updated task. We must send *all* fields, including the ones that we're not updating, since this request type overwrites all of the task's data (unfortunately, Findig doesn't support PATCH yet):

```
$ curl -i -X POST -H "Content-Type: application/json" -d '{"title": "My Task", "due": "2015-07-19 00:
HTTP/1.0 200 OK
Content-Type: text/plain; charset=utf-8
Content-Length: 0
Server: Werkzeug/0.10.4 Python/3.4.2
Date: Sat, 18 Jul 2015 03:47:00 GMT
```

Deleting a task

You can probably guess this one; to do this, we send a DELETE request to the task's URL. Let's delete that task we just created; we're fickle like that:

```
$ curl -i -X DELETE localhost:5000/tasks/1
HTTP/1.0 200 OK
Content-Type: text/plain; charset=utf-8
Content-Length: 0
```

```
Server: Werkzeug/0.10.4 Python/3.4.2
Date: Sat, 18 Jul 2015 03:52:12 GMT
```

It works! It all works!

1.4.6 Customizing error output

Remember when we sent a POST request to `/tasks/` without a required field and it gave us a cryptic error message? We should probably do something about that. We're gonna return a little bit more information to let the client know what exactly has gone wrong.

To do this, we have to override the application's default error handler, which Findig allows us to do for specific exception types by default [1]. The key is realising that *`Validator`* raises specific exceptions when something goes wrong, all resolving to `400 BAD REQUEST`:

- *`findig.tools.validator.MissingFields`* – Raised when the validator expects one or more required fields, but the client does not send them.
- *`findig.tools.validator.UnexpectedFields`* – Raised when the validator receives one or more fields that it does not expect.
- *`findig.tools.validator.InvalidFields`* – Raised when the validator receives one or more fields that can't be converted using the supplied converters.

Knowing this, we can update the app to send a more detailed error whenever a missing field is encountered:

```
@app.error_handler.register(MissingFields)
def on_missing_fields(err):
    output = {
        "error": {
            "type": "missing_fields",
            "fields": err.fields
        },
        "message": "The input is missing one or more parameters.",
    }

    # app.make_response comes from findig.json.App, and is not
    # available on findig.App.
    return app.make_response(output, status=400)
```

You'll also want to import `MissingFields`:

```
from findig.tools.validator import MissingFields
```

Now, let's send another request omitting a field:

```
$ curl -i -X POST -H "Content-Type: application/json" -d '{"title": "My Task"}' localhost:5000/tasks/
HTTP/1.0 400 BAD REQUEST
Content-Type: application/json
Content-Length: 114
Server: Werkzeug/0.10.4 Python/3.4.2
Date: Sat, 18 Jul 2015 05:09:34 GMT

{
  "error": {
    "type": "missing_fields",
```

[1] This can change in very specific circumstances. In particular, if you supply an `error_handler` argument to the application constructor, then this method is no longer available; you would have to check for specific exceptions in the function body of your custom `error_handler` instead.

```
    "fields": [
      "due"
    ]
  },
  "message": "The input is missing one or more parameters."
}
```

As expected, this time we get a more detailed error response.

Here's a little exercise for you; why don't you go ahead and update the app to provide detailed messages for when the client sends an unrecognized field, and for when the client sends badly formed data for the `due` field?

1.4.7 Wrapping up

Whew! Here's the full source code for the app we've built:

```
from datetime import datetime

from findig.extras.sql import SQLA, SQLASet
from findig.json import App
from findig.tools.validator import Validator, MissingFields, UnexpectedFields, InvalidFields
from sqlalchemy.schema import *
from sqlalchemy.types import *


app = App()
db = SQLA("sqlite:///tasks.sqlite", app=app)
validator = Validator(app)


class Task(db.Base):
    id = Column(Integer, primary_key=True)
    title = Column(String(150), nullable=False)
    desc = Column(String, nullable=True)
    due = Column(DateTime, nullable=False)


@validator.restrict("*title", "desc", "*due")
@validator.enforce(due=validator.date("%Y-%m-%d %H:%M:%S%z"))
@app.route("/tasks/<id>")
@app.resource(lazy=True)
def task(id):
    return tasks().fetch(id=id)

@app.route("/tasks/")
@task.collection(lazy=True)
def tasks():
    return SQLASet(Task)


@app.error_handler.register(MissingFields)
def on_missing_fields(err):
    output = {
        "error": {
            "type": "missing_fields",
            "fields": err.fields
        },
```

```
        "message": "The input is missing one or more parameters.",
    }

    # app.make_response comes from findig.json.App, and is not
    # available on findig.App.
    return app.make_response(output, status=400)


if __name__ == '__main__':
    from werkzeug.serving import run_simple
    run_simple("localhost", 5000, app, use_reloader=True, use_debugger=False)
```

We've designed an built a functioning API, but we've only used a subset of what Findig has to offer. Have a look at *Counter* for a tool that counts hits to your resources (this is more useful than it sounds upfront). The *findig.tools.protector* module provides utilies for restrict access to your API to authorized users/clients.

If you're interested in supporting custom content-types, rather than just JSON, have a look at *Custom applications*. The process is very similar to the custom error handler we built in this tutorial.

CHAPTER 2

API Documentation

This section isn't ready yet. Please check back later.

2.1 Findig core modules

Findig provides a number of core modules that each typical application uses in some way (whether explicitly by the application code, or internally by Findig). They are documented in these pages:

2.1.1 App — classes for creating WSGI callables

The core Findig namespace defines the Findig :class:App class, which is essential to building Findig applications. Every :class:App is capable of registering resources as well as URL routes that point to them, and is a WSGI callable that can be passed to any WSGI complaint server.

class `findig.`**App**(*autolist=False*)

Bases: *findig.dispatcher.Dispatcher*

request_class

The class used to wrap WSGI environments by this App instance.

alias of `Request`

build_context(*environ*)

Start a request context.

Parameters **environ** – A WSGI environment.

Returns A context manager for the request. When the context manager exits, the request context variables are destroyed and all cleanup hooks are run.

Note: This method is intended for internal use; Findig will call this method internally on its own. It is *not* re-entrant with a single request.

---

cleanup_hook(*func*)

Register a function that should run after each request in the application.

context(*func*)

Register a request context manager for the application.

A request context manager is a function that yields once, that is used to wrap request contexts. It is called at the beginning of a request context, during which it yields control to Findig, and regains control sometime

after findig processes the request. If the function yields a value, it is made available as an attribute on *findig.context.ctx* with the same name as the function.

Example:

```
>>> from findig.context import ctx
>>> from findig import App
>>>
>>> app = App()
>>> items = []
>>> @app.context
... def meaning():
...     items.extend(["Life", "Universe", "Everything"])
...     yield 42
...     items.clear()
...
>>> with app.test_context(create_route=True):
...     print("The meaning of", end=" ")
...     print(*items, sep=", ", end=": ")
...     print(ctx.meaning)
...
The meaning of Life, Universe, Everything: 42
>>> items
[]
```

startup_hook (*func*)
Register a function to be run before the very first request in the application.

test_context (*create_route=False*, ***args*)
Make a mock request context for testing.

A mock request context is generated using the arguments here. In other words, context variables are set up and callbacks are registered. The returned object is intended to be used as a context manager:

```
app = App()
with app.test_context():
    # This will set up request context variables
    # that are needed by some findig code.
    do_some_stuff_in_the_request_context()

# After the with statement exits, the request context
# variables are cleared.
```

This method is really just a shortcut for creating a fake WSGI environ with `werkzeug.test.EnvironBuilder` and passing that to *build_context()*. It takes the very same keyword parameters as `EnvironBuilder`; the arguments given here are passed directly in.

Parameters **create_route** – Create a URL rule routing to a mock resource, which will match the path of the mock request. This must be set to True if the mock request being generated doesn't already have a route registered for the request path, otherwise this method will raise a `werkzeug.exceptions.NotFound` error.

Returns A context manager for a mock request.

class `findig.json.`**App** (*indent=None*, *encoder_cls=None*, *autolist=False*)
A *findig.App* that works with application/json data.

This app is pre-configured to parse incoming `application/json` data, output `application/json` data by default and convert errors to `application/json` responses.

Parameters

- **indent** – The number of spaces to indent by when outputting JSON. By default, no indentation is used.
- **encoder_cls** – A `json.JSONEncoder` subclass that should be used to serialize data into JSON. By default, an encoder that converts all mappings to JSON objects and all other iterables to JSON lists in addition to the normally supported simplejson types (int, float, str) is used.
- **autolist** – Same as the *autolist* parameter in *findig.App*.

2.1.2 findig.content — Formatters, parsers and error handlers

These are helper implementations of content-handling 'functions' for parsing, formatting and error-handling. The module exposes *Parser*, *Formatter* and *ErrorHandler* respectively, each of which acts like a function but introduces some additional semantics.

Although this is the default behavior, Findig applications are not required to use to tools provided by this module and may use any callable in their place.

Note: Instances of *Formatter* and *Parser* require an active request context to work when called.

class findig.content.**ErrorHandler**

A generic implementation of a error handler 'function'.

A *ErrorHandler* collects handler functions for specific exception types, so that when it is called, it looks up the appropriate handler for the exception that it is called with. The handler used is the closest superclass of the exception's type. If no handler was registered for the exception, then it is raised again.

register(*err_type*, *handler*)

Register a handler function for a particular exception type and its subclasses.

Parameters **err_type** – A type of Exception

Type BaseException or subclass.

Handler A function that will handle errors of the given type.

This method is also usable as a decorator factory:

```
handler = ErrorHandler()
@handler.register(ValueError)
def handle_value_err(e):
    # Handle a value error
    pass
```

class findig.content.**Formatter**

A generic implementation of a formatter 'function'.

A *Formatter* collects handler functions for specific mime-types, so that when it is called, it looks up the the appropriate function to call in turn, according to the mime-type specified by the request's `Accept` header.

register(*mime_type*, *handler*, *default=False*)

Register a handler function for a particular content-type.

Parameters

- **mime_type** – A content type.
- **handler** – A handler function for the given content type.

- **default** – Whether the handler should be used for requests which don't specify a preferred content-type. Only one default content type may be given, so if `default=True` is set multiple times, only the last one takes effect.

Tip: This method can also be used as a generator factory.

class `findig.content.`**Parser**

A generic implementation of a parser 'function'.

A *Parser* collects handler functions for specific mime-types, so that when it is called, it looks up the the appropriate function to call in turn, according to the mime-type specified by the request's `Content-Type` header.

register(*mime_type*, *handler*, *default=False*)

Register a handler function for a particular content-type.

Parameters

- **mime_type** – A content type.
- **handler** – A handler function for the given content type.
- **default** – Whether the handler should be used for requests which don't specify a preferred content-type. Only one default content type may be given, so if `default=True` is set multiple times, only the last one takes effect.

Tip: This method can also be used as a generator factory.

2.1.3 `findig.context` — Request context variables

This module stores request context variables. That is, variables whose values are assigned during the handling of a request and then cleared immediately after the request is done.

Important: With exception of *ctx*, all of the variables documented here are proxies to attributes of *ctx*. For example, *app* is a proxy to *ctx.app*.

`findig.context.`**ctx = <werkzeug.local.Local object>**

A global request context local that can be used by anyone to store data about the current request. Data stored on this object will be cleared automatically at the end of each request and call only be seen on the same thread that set the data. This means that data accessed through this object will only ever be relevant to the current request that is being processed.

`findig.context.`**app = <LocalProxy unbound>**

The *findig.App* instance that responded to the request.

`findig.context.`**request = <LocalProxy unbound>**

An object representing the current request and a subclass of *findig.App.request_class*.

`findig.context.`**url_adapter = <LocalProxy unbound>**

An object representing a `werkzeug.routing.MapAdapter` for the current reques that can be used to build URLs.

`findig.context.`**dispatcher = <LocalProxy unbound>**

The *Dispatcher* that registered the resource that the current request is directed to.

`findig.context.`**resource = <LocalProxy unbound>**

The *AbstractResource* that the current request is directed to.

findig.context.**url_values = <LocalProxy unbound>**
A dictionary of values that have been extracted from the request path by matching it against a URL rule.

2.1.4 findig.data_model — Data access for Findig applications

This module defines data models, which implement data access for a particular resource. In a typical Findig application, each resource has an *AbstractDataModel* attached which has functions defined implementing the data operations that are supported by that resource.

By default, *Resources* have a *DataModel* attached, but this can be replaced with any concrete *AbstractDataModel*.

class findig.data_model.**AbstractDataModel**
Bases: collections.abc.Mapping

An object responsible for managing the data for a specific resource. Essentially, it is a mapping of data operations to the functions that perform.

The following data operations (and their signatures) are supported:

- **read**()
  Retrieve the data for the resource.
- **write**(*data*)
  Replace the resource's existing data with the new data. If the resource doesn't exist yet, create it.
- **delete**()
  Completely obliterate a resource's data; in general the resource should be thought to no longer exist after this occurs.
- **make**(*data*)
  Create a child resource.
  Returns A mapping that can identify the created child (i.e., a key).

To implement this abstract base class, do *either* of the following:

- Implement methods on your subclass with the names of the data operations you want your model to support. For example, the following model implements read and write actions:

```
class ReadWriteModel(AbstractDataModel):
    def read():
        '''Perform backend read.'''

    def write(new_data):
        '''Perform backend write.'''
```

- Re-implement the mapping interface on your subclasses, such that instances will map from a data operation (str) to a function that implements it. This requires implementing __iter__, __len__ and __getitem__ at a minimum. For an example, take a look at the source code for this class.

class findig.data_model.**DataModel**
Bases: *findig.data_model.AbstractDataModel*, collections.abc.MutableMapping

A generic, concrete implementation of *AbstractDataModel*

This class is implemented as a mutable mapping, so implementation functions for data operations can be set, accessed and deleted using the mapping interface:

```
>>> dm = DataModel()
>>> dm['read'] = lambda: ()
```

```
>>> 'read' in dm
True
```

Also, be aware that data model instances can be called to return a decorator for a specific data operation:

```
>>> @dm('write')
... def write_some_data(data):
...     pass
...
>>> dm['write'] == write_some_data
True
```

2.1.5 `findig.dispatcher` – Low-level dispatchers for Findig applications

This low-level module defines the *Dispatcher* class, from which *findig.App* derives.

class findig.dispatcher.**Dispatcher**(*formatter=None*, *parser=None*, *error_handler=None*, *pre_processor=None*, *post_processor=None*)

A *Dispatcher* creates resources and routes requests to them.

Parameters

- **formatter** – A function that converts resource data to a string string suitable for output. It returns a 2-tuple: *(mime_type, output)*. If not given, a generic *findig.content.Formatter* is used.
- **parser** – A function that parses request input and returns a 2-tuple: *(mime_type, data)*. If not given, a generic *findig.content.Parser*.
- **error_handler** – A function that converts an exception to a Response. If not given, a generic *findig.content.ErrorHandler* is used.
- **pre_processor** – A function that is called on request data just after is is parsed.
- **post_processor** – A function that is called on resource data just before it is formatted.

This class is fairly low-level and shouldn't be instantiated directly in application code. It does however serve as a base for *findig.App*.

formatter
If a *formatter* function was given to the constructor, then that is used. Otherwise, a generic *findig.content.Formatter* is used.

parser
The value that was passed for *parser* to the constructor. If no argument for *parser* was given to the constructor, then a generic *findig.content.Parser* is used.

error_handler
The value that was passed for error_handler to the constructor, or if None was given, then a generic *findig.content.ErrorHandler*.

response_class
A class that is used to construct responses after they're returned from formatters.

alias of Response

build_rules()
Return a generator for all of the url rules collected by the *Dispatcher*.

Return type Iterable of werkzeug.routing.Rule

Note: This method will 'freeze' resource names; do not change resource names after this function is invoked.

dispatch()
Dispatch the current request to the appropriate resource, based on which resource the rule applies to.

This function requires an active request context in order to work.

resource(*wrapped*, ***args*)
Create a *findig.resource.Resource* instance.

Parameters **wrapped** – A wrapped function for the resource. In most cases, this should be a function that takes named route arguments for the resource and returns a dictionary with the resource's data.

The keyword arguments are passed on directly to the constructor for `Resource`, with the exception that *name* will default to {module}.{name} of the wrapped function if not given.

This method may also be used as a decorator factory:

```
@dispatcher.resource(name='my-very-special-resource')
def my_resource(route, param):
    return {'id': 10, ... }
```

In this case the decorated function will be replaced by a `Resource` instance that wraps it. Any keyword arguments passed to the decorator factory will be handed over to the `Resource` constructor. If no keyword arguments are required, then `@resource` may be used instead of `@resource()`.

Note: If this function is used as a decorator factory, then a keyword parameter for *wrapped* must not be used.

route(*resource*, *rulestr*, ***ruleargs*)
Add a route to a resource.

Adding a URL route to a resource allows Findig to dispatch incoming requests to it.

Parameters

- **resource** (`Resource` or function) – The resource that the route will be created for.
- **rulestr** (*str*) – A URL rule, according to werkzeug's specification.

See `werkzeug.routing.Rule` for valid rule parameters.

This method can also be used as a decorator factory to assign route to resources using declarative syntax:

```
@route("/index")
@resource(name='index')
def index_generator():
    return ( ... )
```

unrouted_resources
A list of resources created by the dispatcher which have no routes to them.

2.1.6 findig.resource — Classes for representing API resources

class findig.resource.**Resource**(*wrapped=None*, *lazy=None*, *name=None*, *model=None*, *formatter=None*, *parser=None*, *error_handler=None*)
Bases: *findig.resource.AbstractResource*

A concrete implementation of *AbstractResource*.

This accepts keyword arguments only.

Parameters

- **wrapped** – A function which the resource wraps; it typically returns the data for that particular resource.
- **lazy** – Indicates whether the wrapped resource function returns lazy resource data; i.e. data is not retrieved when the function is called, but at some later point when the data is accessed. Setting this allows Findig to evaluate the function's return value after all resources have been declared to determine if it returns anything useful (for example, a :class:DataRecord which can be used as a model).
- **name** – A name that uniquely identifies the resource. If not given, it will be randomly generated.
- **model** – A data-model that describes how to read and write the resource's data. By default, a generic *findig.data_model.DataModel* is attached.
- **formatter** – A function that should be used to format the resource's data. By default, a generic *findig.content.Formatter* is attached.
- **parser** – A function that should be used to parse request content for the resource. By default, a generic *findig.content.Parser* is attached.
- **error_handler** – A function that should be used to convert exception into Responses. By default, a *findig.content.ErrorHandler* is used.

model
The value that was passed for *model* to the constructor, or if None was given, a *DataModel*.

formatter
If a *formatter* function was given to the constructor, then that is used. Otherwise, a generic *findig.content.Formatter* is used.

parser
The value that was passed for *parser* to the constructor. If no argument for *parser* was given to the constructor, then a generic *findig.content.Parser* is used.

error_handler
The value that was passed for error_handler to the constructor, or if None was given, then a generic *findig.content.ErrorHandler*.

collection(*wrapped=None*, ***args*)
Create a *Collection* instance

Parameters **wrapped** – A wrapped function for the collection. In most cases, this should be a function that returns an iterable of resource data.

The keyword arguments are passed on to the constructor for :class:Collection, except that if no *name* is given, it defaults to {module}.{name} of the wrapped function.

This function may also be used as a decorator factory:

```
@resource.collection(include_urls=True)
def mycollection(self):
    pass
```

The decorated function will be replaced in its namespace by a *Collection* that wraps it. Any keyword arguments passed to the decorator factory will be handed over to the *Collection* constructor. If no keyword arguments are required, then `@collection` may be used instead of `@collection()`.

compose_model(*wrapper_args=None*)

Noindex

Make a composite model for the resource by combining a lazy data handler (if present) and the model specified on the resource.

Parameters **wrapper_args** – A set of arguments to call the wrapped function with, so that a lazy data handler can be retrieved. If none is given, then fake data values are passed to the wrapped function. In this case, the data-model returned *must not* be used.

Returns A data-model for the resource

This is an internal method.

get_supported_methods(*model=None*)

Return a set of HTTP methods supported by the resource.

Parameters **model** – The data-model to use to determine what methods supported. If none is given, a composite data model is built from `self.model` and any data source returned by the resource's wrapped function.

handle_request(*request*, *wrapper_args*)

Dispatch a request to a resource.

See *AbstractResource.handle_request()* for accepted parameters.

class `findig.resource.`**AbstractResource**

Bases: `object`

Represents a very low-level web resource to be handled by Findig.

Findigs apps are essentially a collection of routed resources. Each resource is expected to be responsible for handling some requests to a set of one or more URLs. When requests to such a URL is received, Findig looks-up what resource is responsible, and hands the request object over to the resource for processing.

Custom implementations of the abstract class are possible. However, this class operates at a very low level in the Findig stack, so it is recommended that they are only used for extreme cases where those low-level operations are needed.

In addition to the methods defined here, resources should have a name attribute, which is a string that uniquely identifies it within the app. Optional *parser* and *formatter* attributes corresponding to `findig.content.AbstractParser` and `finding.content.AbstractFormatter` instances respectively, will also be used if added.

build_url(*values*)

Build a URL for this resource.

The URL is built using the current WSGI environ, so this function must be called from inside a request context. Furthermore, the resource must have already been routed to in the current app (see: *findig.dispatcher.Dispatcher.route()*), and this method must be passed values for any variables in the URL rule used to route to the resource.

Parameters **values** (`dict`) – Values for the variables in a URL rule used to route to the resource.

Example:

```
>>> from findig import App
>>> app = App()
>>> @app.route("/index/<int:num>")
... @app.resource
... def item(num):
```

```
...     return {}
...
>>> with app.test_context(path="/index/1"):
...     item.build_url(dict(num=4))
...
'/index/4'
```

This method is *not* abstract.

get_supported_methods()
Return a Python set of HTTP methods to be supported by the resource.

handle_request(*request*, *url_values*)
Handle a request to one of the resource URLs.

Parameters

- **request** (*Request*, which in turn is a subclass of werkzeug.wrappers.Request) – An object encapsulating information about the request. It is the same as *findig.context.request*.
- **url_values** – A dictionary of arguments that have been parsed from the URL routes, which may help to better identify the request. For example, if a resource is set up to handle URLs matching the rule `/items/<int:id>` and a request is sent to `/items/43`, then *url_values* will be `{'id': 43}`.

Returns This function should return data that will be transformed into an HTTP response. This is usually a dictionary, but depending on how formatting is configured, it may be any object the output formatter configured for the resource will accept.

class findig.resource.**Collection**(*of*, *include_urls=False*, *bindargs=None*, ***keywords*)
Bases: *findig.resource.Resource*

A *Resource* that acts as a collection of other resources.

Parameters

- **of** (*Resource*) – The type of resource to be collected.
- **include_urls** – If `True`, the collection will attempt to insert a `url` field on each of the child items that it returns. Note that this only works if the child already has enough information in its fields to build a url (i.e., if the URL for the child contains an `:id` fragment, then the child must have an id field, which is then used to build its URL.
- **bindargs** – A dictionary mapping field names to URL variables. For example: a child resource may have the URL variable `:id`, but have a corresponding field named `user_id`; the appropriate value for *bindargs* in this case would be `{'user_id': 'id'}`.

2.1.7 findig.wrappers — The Findig Request object

class findig.wrappers.**Request**(*environ*, *populate_request=True*, *shallow=False*)
Bases: werkzeug.wrappers.Request

A default request class for wrapping WSGI environs.

input
Request content that has been parsed into a python object. This is a read-only property.

max_content_length = 10485760
The maximum allowed content-length for the requests is set to 10MB by default.

2.2 Higher-level tools

The *findig.tools* package contains modules each implementing higher-level tools for use on Findig applications and resources. They're completely optional, but using them should make writing a Findig application significantly easier. These tools are implemented in such a way that they can be easily dropped into any Findig application to provide additional functionality without making any underlying changes. The tools available are documented in the following pages:

2.2.1 findig.tools.counter — Hit counters for apps and resources

The *findig.tools.counter* module defines the *Counter* tool, which can be used as a hit counter for your application. Counters can count hits to a particular resource, or globally within the application.

class findig.tools.counter.**Counter**(*app=None*, *duration=-1*, *storage=None*)

A *Counter* counter keeps track of hits (requests) made on an application and its resources.

Parameters

- **app** (*findig.App*, or a subclass like *findig.json.App*.) – The findig application whose requests the counter will track.
- **duration** (datetime.timedelta or int representing seconds.) – If given, the counter will only track hits that occurred less than this duration before the current time. Otherwise, all hits are tracked.
- **storage** – A subclass of *AbstractLog* that should be used to store hits. By default, the counter will use a thread-safe, in-memory storage class.

attach_to(*app*)

Attach the counter to a findig application.

Note: This is called automatically for any app that is passed to the counter's constructor.

By attaching the counter to a findig application, the counter is enabled to wrap count hits to the application and fire callbacks.

Parameters **app** (*findig.App*, or a subclass like *findig.json.App*.) – The findig application whose requests the counter will track.

partition(*name*, *fgroup*)

Create a partition that is tracked by the counter.

A partition can be thought of as a set of mutually exclusive groups that hits fall into, such that each hit can only belong to one group in any single partition. For example, if we partition a counter by the IP address of the requesting clients, each possible client address can be thought of as one group, since it's only possible for any given hit to come from just one of those addresses.

For every partition, a *grouping function* must be supplied to help the counter determine which group a hit belongs to. The grouping function takes a request as its parameter, and returns a hashable result that identifies the group. For example, if we partition by IP address, our grouping function can either return the IP address's string representation or 32-bit (for IPv4) integer value.

By setting up partitions, we can query a counter for the number of hits belonging to a particular group in any of our partitions. For example, if we wanted to count the number GET requests, we could partition the counter on the request method (here our groups would be GET, PUT, POST, etc) and query the counter for the number of hits in the GET group in our request method partition:

```
counter = Counter(app)

# Create a partition named 'method', which partitions our
# hits by the request method (in uppercase).
counter.partition('method', lambda request: request.method.upper())

# Now we can query the counter for hits belonging to the 'GET'
# group in our 'method' partition
hits = counter.hits()
number_of_gets = hits.count(method='GET')
```

Parameters

- **name** – The name for our partition.
- **fgroup** – The grouping function for the partition. It must] be a callable that takes a request and returns a hashable value that identifies the group that the request falls into.

This method can be used as a decorator factory:

```
@counter.partition('ip')
def getip(request):
    return request.remote_addr
```

A counter may define more than one partition.

every(*n*, *callback*, *after=None*, *until=None*, *resource=None*)
Call a callback every *n* hits.

Parameters

- **resource** – If given, the callback will be called on every *n* hits to the resource.
- **after** – If given, the callback won't be called until *after* this number of hits; it will be called on the (after+1)th hit and every nth hit thereafter.
- **until** – If given, the callback won't be called after this number of hits; it will be called up to and including this number of hits.

If partitions have been set up (see *partition()*), additional keyword arguments can be given as `{partition_name}={group}`. In this case, the hits are filtered down to those that match the partition before issuing callbacks. For example, we can run some code on every 100th GET request after the first 1000 like this:

```
counter.partition('method', lambda r: r.method.upper())

@counter.every(100, after=1000, method='GET')
def on_one_hundred_gets(method):
    pass
```

Furthermore, if we wanted to issue a callback on every 100th request of any specific method, we can do this:

```
@counter.every(100, method=counter.any)
def on_one_hundred(method):
    pass
```

The above code is different from simply `every(100, callback)` in that `every(100, callback)` will call the callback on every 100th request received, while the example will call the callback of every 100th request of a particular method (every 100th GET, every 100th PUT, every 100th POST etc).

Whenever partition specs are used to register callbacks, then the callback must take a named argument matching the partition name, which will contain the partition group for the request that triggered the callback.

at(*n*, *callback*, *resource=None*)
Call a callback on the *nth* hit.

Parameters **resource** – If given, the callback will be called on every *n* hits to the resource.

Like `every()`, this function can be called with partition specifications.

This function is equivalent to `every(1, after=n-1, until=n)`

after_every(*n*, *callback*, *after=None*, *until=None*, *resource=None*)
Call a callback after every *n* hits.

This method works exactly like `every()` except that callbacks registered with `every()` are called before the request is handled (and therefore can throw errors that interupt the request) while callbacks registered with this function are run after a request has been handled.

after(*n*, *callback*, *resource=None*)
Call a callback after the *nth* hit.

This method works exactly like `at()` except that callbacks registered with `at()` are called before the request is handled (and therefore can throw errors that interupt the request) while callbacks registered with this function are run after a request has been handled.

hits(*resource=None*)
Get the hits that have been recorded by the counter.

The result can be used to query the number of total hits to the application or resource, as well as the number of hits belonging to specific partition groups:

```
# Get the total number of hits
counter.hits().count()

# Get the number of hits belonging to a partition group
counter.hits().count(method='GET')
```

The result is also an iterable of (`datetime.datetime`, *partition_mapping*) objects.

Parameters **resource** – If given, only hits for this resource will be retrieved.

class `findig.tools.counter.`**AbstractLog**(*duration*, *resource*)
Abstract base for a storage class for hit records.

This module provides a thread-safe, in-memory concrete implementation that is used by default.

__init__(*duration*, *resource*)
Initialize the abstract log

All implementations must support this signature for their constructor.

Parameters

- **duration** (`datetime.timedelta` or int representing seconds.) – The length of time for which the log should store records. Or if -1 is given, the log should store all records indefinitely.
- **resource** – The resource for which the log will store records.

__iter__()
Iter the stored hits.

Each item iterated must be a 2-tuple in the form (`datetime.datetime`, partitions).

count (***partition_spec*)
Return the number of hits stored.

If no keyword arguments are given, then the total number of hits stored should be returned. Otherwise, keyword arguments must be in the form `{partition_name}={group}`. See *Counter.partition()*.

track (*partitions*)
Store a hit record

Parameters **partitions** – A mapping from partition names to the group that the hit matches for the partition. See *Counter.partition()*.

Counter example

Counters can be used to implement more complex tools. For example, a simple rate-limiter can be implemented using the counter API:

```
from findig.json import App
from findig.tools.counter import Counter
from werkzeug.exceptions import TooManyRequests

app = App()

# Using the counter's duration argument, we can set up a
# rate-limiter to only consider requests in the last hour.
counter = counter(app, duration=3600)

LIMIT = 1000

@counter.partition('ip')
def get_ip(request):
    return request.remote_addr

@counter.every(1, after=1000, ip=counter.any)
def after_thousandth(ip):
    raise TooManyRequests("Limit exceeded for: {}".format(ip))
```

2.2.2 `findig.tools.protector` — Authorization tools

For many web API resources, it is desirable to restrict access to only specific users or clients that have been authorized to use them. The tools in this module provide one mechanism for putting such restrictions in place.

The *Protector* is the core tool provided here. Its instances collect information about resources that should be guarded against authorized access, and on each request, it checks that requests to those resources present a valid authorization.

The precise authorization mechanism used by the protector is controlled by the *GateKeeper* abstract class for which an application developer may supply their own concrete instance. Alternatively, some protectors supply their own gatekeepers (example: *BasicProtector* uses **RFC 2617#section-2** as its authorization mechanism).

Scopes

Protectors provide implicit support for 'scopes' (an idea borrowed from OAuth 2, with some enhancements). While completely optional, their use provides a way allow access to only portions of an API while denying access to others.

An authorization (commonly represented by a token) may have some scopes (identified by application chosen strings) associated with it which define what portions of the API that a request using the authorization can access; only resources belong to one of the scopes associated with the authorization, or resources that belong to no scopes are accessible. Protectors provide a mechanism for marking a guarded resource as belonging to a scope.

For example, an application may provide a resource guarded under the scope `foobar`. In order to access the resource, then a request must present authorization that encapsulates the `foobar` scope, otherwise the request is denied.

Tip: While recommended, using scopes is optional. In fact, some authorization mechanisms do not provide a way to encapsulate scopes. To provide scope support for a custom authentication mechanism that encapsulates scopes, see *GateKeeper*.

Findig extends authorization scopes with special semantics that can affect the way they are used by a protector. The grammar for authorization scopes is given below:

```
auth_scope   ::=  scope_name["+"permissions]
scope_name   ::=  scope_fragment{"/"scope_fragment}
permissions  ::=  permission{permission}
permission   ::=  "c" | "r" | "u" | "d"
```

`scope_fragment` is token that does not include the '+' or '/' characters. Whenever a permission is omitted, the 'r' is permission is implied.

Permissions are used to control which actions an authorization permits on the resources falling into its scope, according to this table:

Action	Permission
HEAD	`r`
GET	`r`
POST	`c`
PUT	`c` and `u`
DELETE	`d`

So for example, an authorization with the scope `foobar+rd` can read and delete resources under the `foobar` scope.

The '/' character is used to denote sub-scopes (and super-scopes). `foo/bar` is considered a sub-scope of `foo` (and `foo` a super-scope of `foo/bar`), and so on. This is useful, because by default if a request possesses authorization for a super-scope, then this implicitly authorizes its sub-scopes as well.

Scopes attached to a resource follow a simpler grammar:

```
resource_scope  ::=  scope_name
```

In other words, the permissions are omitted (because the protector multiplexes which permission is required from the request method).

The *findig.tools.protector.scopeutil* module provides some functions for working with scopes.

Protectors

class `findig.tools.protector.`**`Protector`**(*app=None*, *subscope_separator="/"*, *gatekeeper=None*)

A protector is responsible for guarding access to a restricted resource:

```
from findig import App

app = App()
protector = Protector(app)
protector.guard(resource)
```

Parameters

- **app** – A findig application instance.
- **subscope_separator** – A separator used to denote sub-scopes.
- **gatekeeper** – A concrete implementation of *GateKeeper*. If not provided, the protector will deny all requests to its guarded resources.

attach_to(*app*)

Attach the protector to a findig application.

Note: This is called automatically for any app that is passed to the protector's constructor.

By attaching the protector to a findig application, the protector is enabled to intercept requests made to the application, performing authorization checks as needed.

Parameters **app** (*findig.App*, or a subclass like *findig.json.App*.) – A findig application whose requests the protector will intercept.

guard(*resource*[, *scope*[, *scope*[, ...]]])

Guard a resource against unauthorized access. If given, the *scopes* will be used to protect the resource (similar to oauth) such that only requests with the appropriate *scope* will be allowed through.

If this function is called more than once, then a grant by *any* of the specifications will allow the request to access the resource. For example:

```
# This protector will allow requests to res with BOTH
# "user" and "friends" scope, but it will also allow
# requests with only "foo" scope.
protector.guard(res, "user", "friends")
protector.guard(res, "foo")
```

A protector can also be used to decorate resources for guarding:

```
@protector.guard
@app.route("/foo")
def foo():
    # This resource is guarded with no scopes; any authenticated
    # request will be allowed through.
    pass

@protector.guard("user/email_addresses")
@app.route("/bar")
def bar():
    # This resource is guarded with "user/email_addresses" scope,
    # so that only requests authorized with that scope will be
    # allowed to access the resource.
    pass

@protector.guard("user/phone_numbers", "user/contact")
@app.route("/baz")
def baz():
```

```
            # This resource is guarded with both "user/phone_numbers" and
            # "user/contact" scope, so requests must be authorized with both
            # to access this resource.
            pass

        # NOTE: Depending on the value passed for 'subscope_separator' to the
        # protector's constructor, authenticated requests authorized with "user" scope
        # will also be allowed to access all of these resources (default behavior).
```

authenticated_client
Get the client id of the authenticated client for the current request, or None.

authenticated_user
Get the username/id of the authenticated user for the current request.

authorized_scope
Get the a list of authorized scopes for the current request.

class findig.tools.protector.**BasicProtector**(*app=None*, *subscope_separator='/'*, *auth_func=None*, *realm='guarded'*)

A *Protector* that implements HTTP Basic Auth.

While straightforward, this protector has a few security considerations:

- Credentials are transmitted in plain-text. If you must use this protector, then at the very least the HTTPS protocol should be used.
- Credentials are transmitted with *each request*. It requires that clients either store user credentials, or prompt the user for their credentials at frequent intervals (possibly every request).
- This protector offers no scoping support; a grant from this protector allows unlimited access to any resource that it guards.

auth_func(*fauth*)
Supply an application-defined function that performs authentication.

The function has the signature `fauth(username:str, password:str) -> bool` and should return whether or not the credentials given authenticate successfully.

auth_func is usable as a decorator:

```
        @protector.auth_func
        def check_credentials(usn, pwd):
            user = db.get_obj(usn)
            return user.password == pwd
```

GateKeepers

Each *Protector* should be supplied with a *GateKeeper* that extracts any authorization information embedded in a request. *Protector* uses a default gatekeeper which denies all requests made to its guarded resources.

An application may provide its own gatekeeper that implements the desired authorization mechanism. That's done by implementing the *GateKeeper* abstract base class.

class findig.tools.protector.**GateKeeper**
To implement a gatekeeper, implement at least *check_auth()* and *get_username()*.

check_auth()
Try to perform an authorization check using the request context variables.

Perform the authorization check using whatever mechanism that the gatekeeper's authorization is handled. If authorization fails, then an `Unauthorized` error should be raised.

Return a 'grant' that will be used to query the gatekeeper about the authorization.

get_clientid(*grant*)
Return the client that sent the request to the grant. (Optional)

get_scopes(*grant*)
Return a list of scopes that the grant is authorized with. (Optional)

get_username(*grant*)
Return the username/id of the user that authorized the grant.

2.2.3 findig.tools.protector.scopeutil — tools for working with auth scopes

These functions are used protectors to implement *scoping*.

findig.tools.protector.scopeutil.**check_encapsulates**(*root*, *child*, *sep='/'*)
Check that one scope item encapsulates of another.

A `scope` item encapsulates when it is a super-scope of the other, and when its permissions are a superset of the other's permissions.

This is used to implement sub-scopes, where permissions granted on a broad scope can be used to imply permissions for a sub-scope. By default, sub-scopes are denoted by a preceeding '/'.

For example, a scope permission if `user+r` is granted to an agent, then that agent is also implied to have been granted `user/emails+r`, `user/friends+r` and so on.

Parameters

- **root** – A super-scope
- **child** – A potential sub-scope
- **sep** – The separator that is used to denote sub-scopes.

findig.tools.protector.scopeutil.**compress_scope_items**(*scopes*, *default_mode='r'*)
Return a set of equivalent scope items that may be smaller in size.

Input scope items must be a normalized set of scope items.

findig.tools.protector.scopeutil.**normalize_scope_items**(*scopes*, *default_mode='r'*, *raise_err=True*)
Return a set of scope items that have been normalized.

A normalized set of scope items is one where every item is in the format:

```
norm_scope  ::=  scope_name+permission
```

Input scope items are assumed to be 'r' by default. Example, the scope item `user` will normalize to `user+r`.

Input scope items that contain more than one permission are expanded to multiple scope items. For example the scope item `user+ud` is expanded to (`user+u`, `user+d`).

Note that permissions are atomic, and none implies another. For example, `user+u` will expand to `user+u` and NOT (`user+r`, `user+u`).

Parameters

- **scopes** – A list of *scope items*.
- **default_mode** – The permission that should be assumed if one is omitted.
- **raise_err** – If `True`, malformed scopes will raise a `ValueError`. Otherwise they are omitted.

`findig.tools.protector.scopeutil.`**ANY = {'$^&#THISISGARBAGE#*@&@#$*@$&DFDF#&#@&@&##*&@DH**

A special scope item that implicitly encapsulates all other scope items

2.2.4 `findig.tools.validator` — Request input validators

The *`findig.tools.validator`* module exposes the *`Validator`* which can be used to validate an application or request's input data.

Validators work by specifying a converter for each field in the input data to be validated:

```
validator = Validator(app)

@validator.enforce(id=int)
@app.route("/test")
def resource():
    pass

@resource.model("write")
def write_resource(data):
    assert isinstance(data['id'], int)
```

If the converter fails to convert the field's value, then a `400 BAD REQUEST` error is sent back.

Converters don't have to be functions; they can be a singleton list containing another converter, indicating that the field is expected to be a list of items for which that converter works:

```
@validator.enforce(ids=[int])
@app.route("/test2")
def resource2():
    pass

@resource2.model("write")
def write_resource(data):
    for id in data['ids']:
        assert isinstance(id, int)
```

Converters can also be string specifications corresponding to a pre-registered converter and its arguments. All of werkzeug's builtin converters and their arguments and their arguments are pre-registered and thus usable:

```
@validator.enforce(foo='any(bar,baz)', cid='string(length=3)')
@app.route("/test3")
def resource3():
    pass

@resource3.model("write")
def write_resource(data):
    assert data['foo'] in ('bar', 'baz')
    assert len(data['cid']) == 3
```

exception `findig.tools.validator.`**ValidationFailed**

Raised whenever a *`Validator`* fails to validate one or more fields.

This exception is a subclass of `werkzeug.exceptions.BadRequest`, so if allowed to bubble up, findig will send a `400 BAD REQUEST` response automatically.

Applications can, however, customize the way this exception is handled:

```
from werkzeug.wrappers import Response

# This assumes that the app was not supplied a custom error_handler
# function as an argument.
# If a custom error_handler function is being used, then
# do a test for this exception type inside the function body
# and replicate the logic
@app.error_handler.register(ValidationFailed)
def on_validation_failed(e):
    # Construct a response based on the error received
    msg = "Failed to convert input data for the following fields: "
    msg += str(e.fields)
    return Response(msg, status=e.status)
```

fields
A list of field names for which validation has failed. This will always be a complete list of failed fields.

validator
The *Validator* that raised the exception.

exception findig.tools.validator.**UnexpectedFields**(*fields*, *validator*)
Bases: *findig.tools.validator.ValidationFailed*

Raised whenever a resource receives an unexpected input field.

exception findig.tools.validator.**MissingFields**(*fields*, *validator*)
Bases: *findig.tools.validator.ValidationFailed*

Raised when a resource does not receive a required field in its input.

exception findig.tools.validator.**InvalidFields**(*fields*, *validator*)
Bases: *findig.tools.validator.ValidationFailed*

Raised when a resource receives a field that the validator can't convert.

class findig.tools.validator.**Validator**(*app=None*, *include_collections=True*)
Bases: object

A higher-level tool to be used to validate request input data.

Parameters

- **app** (*findig.App*) – The Findig application that the validator is attached to.
- **include_collections** – If `True`, any validation rules set on any resource will also be used for any *Collection* that collects it. Even when this argument is set, inherited rules can still be overridden by declaring rules specifically for the collection.

Validators are only capable of validating request input data (i.e., data received as part of the request body). To validate URL fragments, consider using *converters* in your URL rules. See werkzeug's routing reference.

Validators work by specifying converters for request input fields. If a converter is specified, the validator will use it to convert the field and replace it with the converted value. See *enforce()* for more about converters.

attach_to(*app*)
Hook the validator into a Findig application.

Doing so allows the validator to inspect and replace incoming input data. This is called automatically for an app passed to the validator's constructor, but can be called for additional app instances. This function should only be called once per application.

Parameters **app** (*findig.App*) – The Findig application that the validator is attached to.

static date (*format*[, *format*[, ...]])

Create a function that validates a date field.

Parameters **format** – A date/time format according to `datetime.datetime.strptime()`. If more than one formats are passed in, the generated function will try each format in order until one of them works on the field (or until there are no formats left to try).

Example:

```
>>> func = Validator.date("%Y-%m-%d %H:%M:%S%z")
>>> func("2015-07-17 09:00:00+0400")
datetime.datetime(2015, 7, 17, 9, 0, tzinfo=datetime.timezone(datetime.timedelta(0, 14400)))
>>> func("not-a-date")
Traceback (most recent call last):
  ...
ValueError: time data 'not-a-date' does not match format '%Y-%m-%d %H:%M:%S%z'

>>> func = Validator.date("%Y-%m-%d %H:%M:%S%z", "%Y-%m-%d")
>>> func("2015-07-17")
datetime.datetime(2015, 7, 17, 0, 0)
```

enforce (*resource*, ***validation_spec*)

Register a validation specification for a resource.

The validation specification is a set of `field=converter` arguments linking an input field name to a converter that should be used to validate the field. A converter can be any of the following:

- `collections.abc.Callable` (including functions) – This can be a simple type such as `int` or `uuid.UUID`, but any function or callable can work. It should take a field value and convert it to a value of the desired type. If it throws an error, then findig will raise a `BadRequest` exception.

  Example:

```
# Converts an int from a valid string base 10 representation:
validator.enforce(resource, game_id=int)

# Converts to a float from a valid string
validator.enforce(resource, duration=float)
```

- `str` – If a string is given, then it is interpreted as a converter specification. A converter specification includes the converter name and optionally arguments for pre-registered converters. The following converters are pre-registered by default (you may notice that they correspond to the URL rule converters available for werkzeug):

  string (*minlength=1*, *length=None*, *maxlength=None*)

  This converter will accept a string.

  Parameters

  - **length** – If given, it will indicate a fixed length field.
  - **minlength** – The minimum allowed length for the field.
  - **maxlength** – The maximum allowed length for the field.

  any (**items*)

  This converter will accept only values from the variable list of options passed as the converter

arguments. It's useful for limiting a field's value to a small set of possible options.

int (*fixed_digits=0*, *min=None*, *max=None*)
This converter will accept a string representation of a non-negative integer.
Parameters
- **fixed_digits** – The number of fixed digits in the field. For example, set this to **3** to convert `'001'` but not `'1'`. The default is a variable number of digits.
- **min** – The minimum allowed value for the field.
- **max** – The maximum allowed value for the field.

float (*min=None*, *max=None*)
This converter will accept a string representation of a non-negative floating point number.
Parameters
- **min** – The minimum allowed value for the field.
- **max** – The maximum allowed value for the field.

uuid()
This converter will accept a string representation of a uuid and convert it to a `uuid.UUID`.

Converters that do not need arguments can omit the parentheses in the converter specification.

Examples:

```
# Converts a 4 character string
validator.enforce(resource, student_id='string(length=10)')

# Converts any of these string values: 'foo', 1000, True
validator.enforce(resource, field='any(foo, 1000, True)')

# Converts any non-negative integer
validator.enforce(resource, game_id='int')

# and any float <1000
validator.enforce(resource, duration='float(max=1000)')
```

Important: Converter specifications in this form **cannot** match strings that contain forward slashes. For example, *'string(length=2)'* will fail to match *'/e'* and *'any(application/json,html)'* will fail to match *'application/json'*.

•or, `list` – This must be a singleton list containing a converter. When this is given, the validator will treat the field like a list and use the converter to convert each item.

Example:

```
# Converts a list of integers
validator.enforce(resource, games=[int])

# Converts a list of uuids
validator.enforce(resource, components=['uuid'])

# Converts a list of fixed length strings
validator.enforce(resource, students=['string(length=10)'])
```

This method can be used as a decorator factory for resources:

```
@validator.enforce(uid=int, friends=[int])
@app.route("/")
def res():
    return {}
```

Converter specifications given here are only checked when a field is present; see *restrict()* for specifying required fields.

> **Warning:** Because of the way validators are hooked up, registering new specifications after the first request has run might cause unexpected behavior (and even internal server errors).

enforce_all(***validation_spec*)
Register a global validation specification.

This function works like *enforce()*, except that the validation specification is registered for all resources instead of a single one.

Global validation specifications have lower precedence than resource specific ones.

static **regex**(*pattern*, *flags=0*, *template=None*)
Create a function that validates strings against a regular expression.

```
>>> func = Validator.regex("boy")
>>> func("boy")
'boy'
>>> func("That boy")
Traceback (most recent call last):
  ...
ValueError: That boy
>>> func("boy, that's handy.")
Traceback (most recent call last):
  ...
ValueError: boy, that's handy.
```

If you supply a template, it is used to construct a return value by doing backslash substitution:

```
>>> func = Validator.regex("(male|female)", template=r"Gender: \1")
>>> func("male")
'Gender: male'
>>> func("alien")
Traceback (most recent call last):
  ...
ValueError: alien
```

restrict([*field*[, *field*[, ...]]], *strip_extra=True*)
Restrict the input data to the given fields

Parameters

- **field** – A field name that should be allowed. An asterisk at the start of the field name indicates a required field (asterisks at the start of field names can be escaped with another asterisk character). This parameter can be used multiple times to indicate different fields.
- **strip_extra** – Controls the behavior upon encountering a field not contained in the list, during validation. If `True`, the field will be removed. Otherwise, a *UnexpectedFields* is raised.

Once this method is called, any field names that do not appear in the list are disallowed.

validate(*data*)
Validate the data with the validation specifications that have been collected.

This function must be called within an active request context in order to work.

Parameters **data** (*mapping, or object with gettable/settable fields*) – Input data

Raises *ValidationFailed* if one or more fields could not be validated.

This is an internal method.

2.2.5 Abstract classes for higher-level tools

Some higher-level tools aren't explicitly implemented by Findig; rather abstract classes for how they are expected to behave are defined so that support libraries and applications can provide their own implementations.

Data sets

Data sets are one example of higher level tools without an explicit implementation. Abstractly, they're collections of resource data that also encapsulate an implicit data model (i.e., instructions on data access). This makes them a powerful replacement for explicitly defining a data-model for each resource, since a resource function can instead return a data set and have Findig construct a resource from it:

```
@app.route("/items/<int:id>")
@app.resource(lazy=True)
def item(id):
    return items().fetch(id=id)

@app.route("/items")
@item.collection(lazy=True)
def items():
    return SomeDataSet()
```

Findig currently includes one concrete implementation: *findig.extras.redis.RedisSet*.

class findig.tools.dataset.**AbstractDataSet**

An abstract data set is a representation of a collection of items.

Concrete implementations must provide *at least* an implementation for `__iter__`, which should return an iterator of `AbstractRecord` instances.

fetch(***search_spec*)

Fetch an `AbstractRecord` matching the search specification.

If this is called outside a request, a lazy record is returned immediately (i.e., the backend isn't hit until the record is explicitly queried).

fetch_now(***search_spec*)

Fetch an `AbstractRecord` matching the search specification.

Unlike `fetch()`, this function will always hit the backend.

filtered(***search_spec*)

Return a filtered view of this data set.

Each keyword represents the name of a field that is checked, and the corresponding argument indicates what it is checked against. If the argument is `Callable`, then it should be a predicate that returns `True` if the field is valid (be aware that the predicate will passed be `None` if the field isn't present on the record), otherwise it is compared against the field for equality.

limit(*count*, *offset=0*)

Return a limited version of this data set.

Parameters

- **offset** – The number of items to skip from the beginning

- **count** – The maximum number of items to return

sorted(**sort_spec*, *descending=False*)
Return a sorted view of this data set.

The method takes a variable number of arguments that specify its sort specification.

If a single, callable argument is provided, it is taken as a sort key for a record.

Otherwise, the arguments are taken as field names to be sorted, in the same order given in the argument list. Records that omit one of these fields appear later in the sorted set than those that don't.

class findig.tools.dataset.**MutableDataSet**
An abstract data set that can add new child elements.

add(*data*)
Add a new child item to the data set.

class findig.tools.dataset.**AbstractRecord**
An representation of an item belonging to a collection.

read()
Read the record's data and return a mapping of fields to values.

class findig.tools.dataset.**MutableRecord**
An abstract record that can update or delete itself.

close_edit_block(*token*)
End a transaction started by `start_edit_block()`.

delete()
Delete the record's data.

edit_block()
A context manager for grouping a chain of edits together. Some subclasses may not support performing reads inside an edit block.

patch(*add_data*, *remove_fields*)
Update the record's data with the new data.

start_edit_block()
Start a transaction to the backend.

Backend edits made through this object should be grouped together until `close_edit_block()` is called.

Returns A token that is passed into `close_edit_block()`.

2.3 Added features enabled by third-party libraries

The modules in the package provide some extra functionality, but rely on additional third-party libraries being present in order to work.

2.3.1 findig.extras.redis — Some tools that are backed by Redis

Note: This module requires *redis-py* and will raise an `ImportError` if redis-py is not installed.

class findig.extras.redis.**RedisSet**(*key=None*, *client=None*, *index_size=4*)
Bases: *findig.tools.dataset.MutableDataSet*

A RedisSet is an AbstractDataSet that stores its items in a Redis database (using a Sorted Set to represent the collection, and a sorted set to represent items).

Parameters

- **key** – The base key that should be used for the sorted set. If not given, one is deterministically generated based on the current resource.
- **client** – A redis.StrictRedis instance that should be used to communicate with the redis server. If not given, a default instance is used.
- **index_size** – The number of bytes to use to index items in the set (per item).

2.4 General Utilities

class findig.utils.**DataPipe**(**funcs*)
An object that folds data over a set of functions.

Parameters **funcs** – A variable list of functions. Each function must take one parameter and return a single value.

Calling this object with data will pass the data through each one of the functions that is has collected, using the result of one function as the argument for the next function. For example, if the data pipe dpipe contains the functions [f1, f2, ..., fn], then dpipe(data) is equivalent to fn(...(f2(f1(data)))).

stage(*func*)
Append a function to the data pipes internal list.

This returns the function that it is called with, so it can be used as a decorator.

class findig.utils.**extremum**(*direction=1*)
A class whose instances are always ordered at one extreme.

Parameters **direction** – If positive, always order as greater than every other object. If negative, orders as less than every other object.

findig.utils.**tryeach**(*funcs*, **args*, ***kwargs*)
Call every item in a list a functions with the same arguments, until one of them does not throw an error. If all of the functions raise an error, then the error from the last function will be re-raised.

Parameters **funcs** – An iterable of callables.

f

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

P

R

S

T

U

V