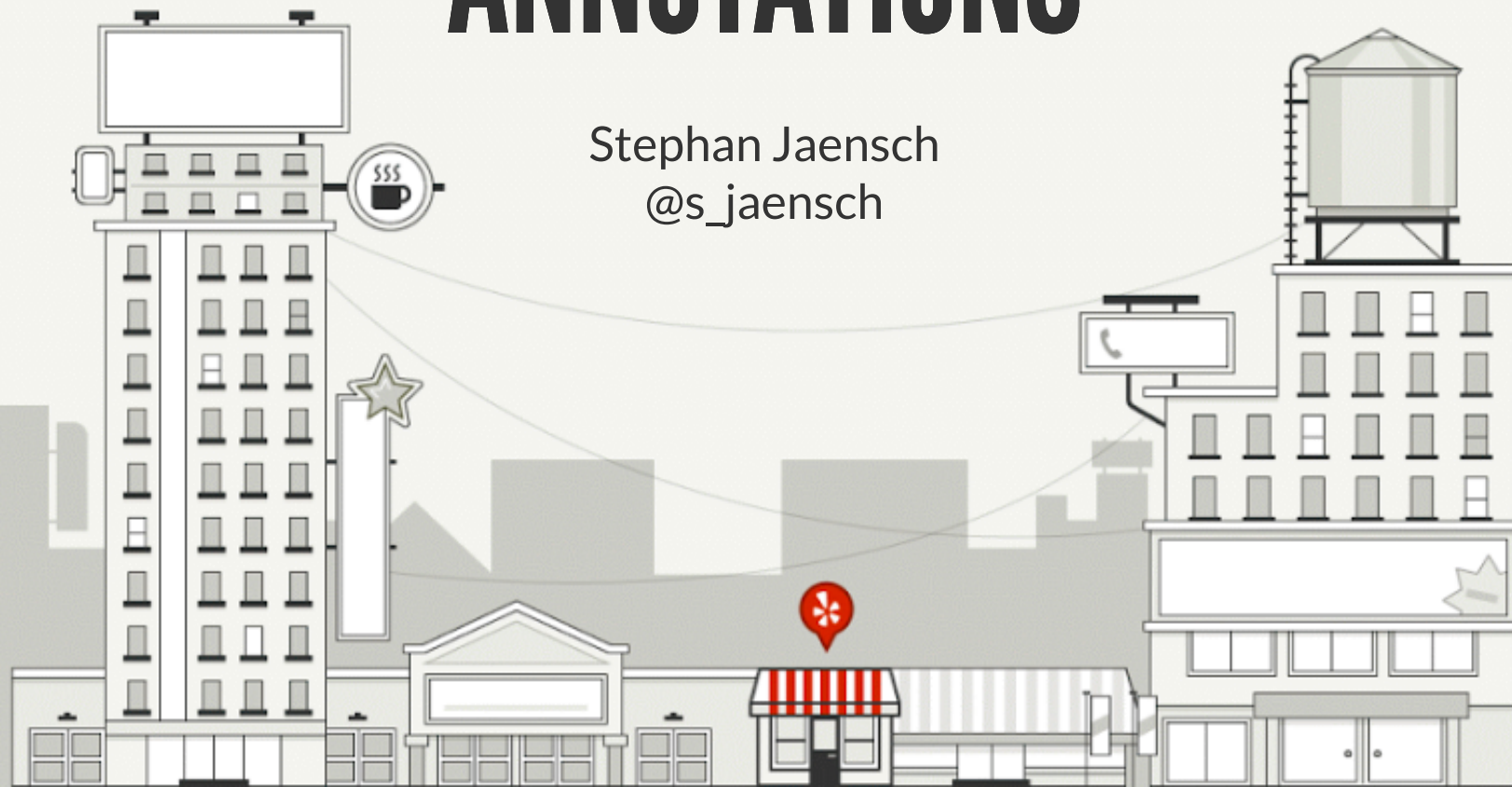


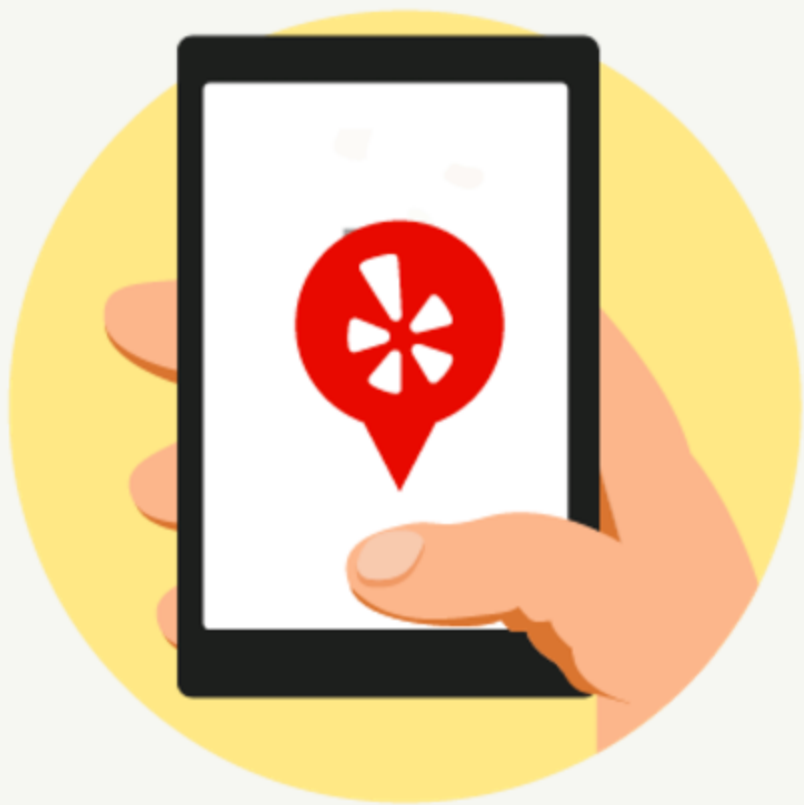
MIGRATING EXISTING CODEBASES TO USING TYPE ANNOTATIONS

Stephan Jaensch
@s_jaensch



YELP'S MISSION

To connect people with great local businesses



WHAT I'LL TALK ABOUT

- What are type annotations, and why you should use them
- How do you incrementally migrate an existing codebase to them
- What are some issues you might encounter
- How can type annotations help across services



OTHER TALKS ABOUT TYPE ANNOTATIONS

- Carl Meyer: Type-checked Python in the real world (Instagram)
- Greg Price: Clearer Code at Scale: Static Types at Zulip and Dropbox



PYTHON TYPE ANNOTATIONS

```
def hello(who: str) -> str:  
    return 'Hello, {}'.format(who)
```

```
hello(5)
```

error: Argument 1 to "hello" has incompatible type "int";
expected "str"

```
def process_data(self, items):  
    self.values = [item.value.id for item in items]
```



MIGRATE A CODEBASE TO USING TYPE ANNOTATIONS

- Goal: All code is type annotated
- Incrementally annotate code
- Make sure checks are run for annotated code



THE MYPY TYPE CHECKER

Friday, 13 July 2018

Mypy 0.620 Released

We've just uploaded mypy 0.620 to the Python Package Index (PyPI). Mypy is an optional static type checker for Python. This release includes new features, bug fixes and library stub (typeshed) updates. You can install it as follows:

```
python3 -m pip install -U mypy
```



PYRE

Pyre

A performant type-checker for Python 3

[INSTALL PYRE](#)

[DOCUMENTATION](#)




ENFORCE ANNOTATIONS

```
[mypy]
check_untyped_defs = True
disallow_untyped_calls = False
disallow_untyped_defs = True
follow_imports = silent
ignore_missing_imports = True
python_version = 3.6
strict_optional = True
warn_redundant_casts = True
```



CHECKING SOURCE CODE ON COMMIT

[Documentation](#) [Supported hooks](#) [Demo](#)

pre-commit

A framework for managing and maintaining multi-language pre-commit hooks.

build passing coverage 100% build passing

Star 1,534 Tweet



CONFIGURING PRE-COMMIT

```
- repo: local
  hooks:
  - id: mypy
    name: mypy
    entry: mypy
    language: python
    language_version: 'python3.6'
    additional_dependencies: ['mypy']
    args: ['--config-file', 'mypy-pre-commit.ini']
    files: ^package_name/.+\.py$
```

```
pre-commit install -f --install-hooks
```



CHECKING SOURCE CODE ON COMMIT

```
Trim Trailing Whitespace.....Passed
Fix End of Files.....Passed
autopep8 wrapper.....Passed
Check Yaml.....(no files to check)Skipped
Debug Statements (Python).....Passed
Tests should end in _test.py.....(no files to check)Skipped
Flake8.....Passed
Check for byte-order marker.....Passed
Fix requirements.txt.....(no files to check)Skipped
Check for added large files.....Passed
Verify biz_app capabilities.....(no files to check)Skipped
pyupgrade.....Passed
Reorder python imports.....Passed
Validate Swagger Specification.....(no files to check)Skipped
mypy.....Failed
hookid: mypy
```

```
biz_app/logic/media.py:166: error: Argument 1 to "GetPhotosFuture" has incompatible type "str"; expected "int"
biz_app/logic/media.py:172: error: Function is missing a return type annotation
biz_app/logic/media.py:216: error: Function is missing a type annotation for one or more arguments
```



RUNNING MYPY AS PART OF YOUR TEST SUITE

```
[mypy]  
ignore_missing_imports = True  
python_version = 3.6  
strict_optional = True  
warn_redundant_casts = True
```



AUTO-GENERATING TYPE ANNOTATIONS

MonkeyType

MonkeyType collects runtime types of function arguments and return values, and can automatically generate stub files or even add draft type annotations directly to your Python code based on the types collected at runtime.

Wednesday, 15 November 2017

Dropbox releases PyAnnotate -- auto-generate type annotations for mypy



TYPE YOUR DATA

```
from typing import Iterable, NamedTuple, Optional

class Business(NamedTuple):
    id: int
    name: str
    photos: Iterable[Photo]
    address1: Optional[str]
    address2: Optional[str]
    address3: Optional[str]
    city: str
    latitude: float
    longitude: float
    ...
```



TYPED DICTIONARIES

```
from typing import Optional
from mypy_extensions import TypedDict

class BusinessDict(TypedDict):
    id: int
    name: str
    address1: Optional[str]
    address2: Optional[str]
```

```
def get_biz_address(business: BusinessDict) -> str:
    ...
    value = business.get('address2', '')
```

```
error: TypedDict "BusinessDict" has no key 'address2'
```



CONVERTING DICTS TO NAMEDTUPLES

```
def namedtuple_from_dict(  
    nt_class,  
    dict_values,  
):  
    """Create a namedtuple from a dict, using the namedtuple  
    attribute names to look up values in the dict."""  
    return nt_class._make(  
        dict_values.get(k) for k in nt_class._fields  
    )
```



HOW TO TYPE THE HELPER FUNCTION?

```
def namedtuple_from_dict(  
    nt_class: Type[NamedTuple],  
    dict_values: Dict[str, Any],  
) -> NamedTuple:  
    """Create a namedtuple from a dict, using the namedtuple  
    attribute names to look up values in the dict."""  
    return nt_class._make(  
        dict_values.get(k) for k in nt_class._fields  
    )
```

```
error: Incompatible return value type  
      (got "NamedTuple", expected "Business")  
error: Argument 1 to "namedtuple_from_dict" has incompatible  
      type "Type[Business]"; expected "Type[NamedTuple]"
```



USING GENERICS

```
Struct = TypeVar('Struct', bound=NamedTuple)

def namedtuple_from_dict(
    nt_class: Type[Struct],
    dict_values: Dict,
) -> Struct:
    """Create a namedtuple from a dict, using the namedtuple
    attribute names to look up values in the dict."""
    return nt_class._make(
        dict_values.get(k) for k in nt_class._fields
    )
```

error: Value of type variable "Struct" of "namedtuple_from_dict" cannot be "Business"



THE SOLUTION: PROTOCOLS

```
from typing_extensions import Protocol

T = TypeVar('T')
class NTProto(Protocol):
    _source: str
    _fields: Tuple[str, ...]

    @classmethod
    def _make(cls: Type[T], iterable: Iterable[Any]) -> T: ...
    # add other methods, if needed

NT = TypeVar('NT', bound=NTProto)
def namedtuple_from_dict(
    nt_class: Type[NT],
    dict_values: Dict[str, Any],
) -> NT:
    return nt_class._make(
        dict_values.get(k) for k in nt_class._fields
    )
```



NAMEDTUPLE AND COUNT / INDEX

```
class Pagination(NamedTuple):  
    count: int  
    index: int
```

```
error: Incompatible types in assignment (expression has type "int",  
base class "tuple" defined the type as  
"Callable[[Tuple[int, ...], Any], int]")
```

```
error: Incompatible types in assignment (expression has type "int",  
base class "tuple" defined the type as  
"Callable[[Tuple[int, ...], Any, int, int], int]")
```



HOW TO ANNOTATE DESCRIPTORS

```
T = TypeVar('T')
V = TypeVar('V')

class SetOnceProperty(Generic[T, V]):

    def __get__(self, instance: T, owner: Type[T]) -> V:
        return self._property_map[instance]

    def __set__(self, instance: T, value: V) -> None:
        if instance in self._property_map:
            raise AttributeError(
                'this attribute can only be set once.'
            )
        self._property_map[instance] = value

class BizAppContext():
    biz_user_id = SetOnceProperty['BizAppContext', int]()
    ...
```

Document descriptors #2566



JukkaL opened this issue on Dec 13, 2016 · 0 comments



RECURSIVE TYPES

```
class Category(NamedTuple):  
    id: int  
    name: str  
    children: List['Category']
```

error: Recursive types not fully supported yet,
nested types replaced with "Any"

Support recursive types. #731



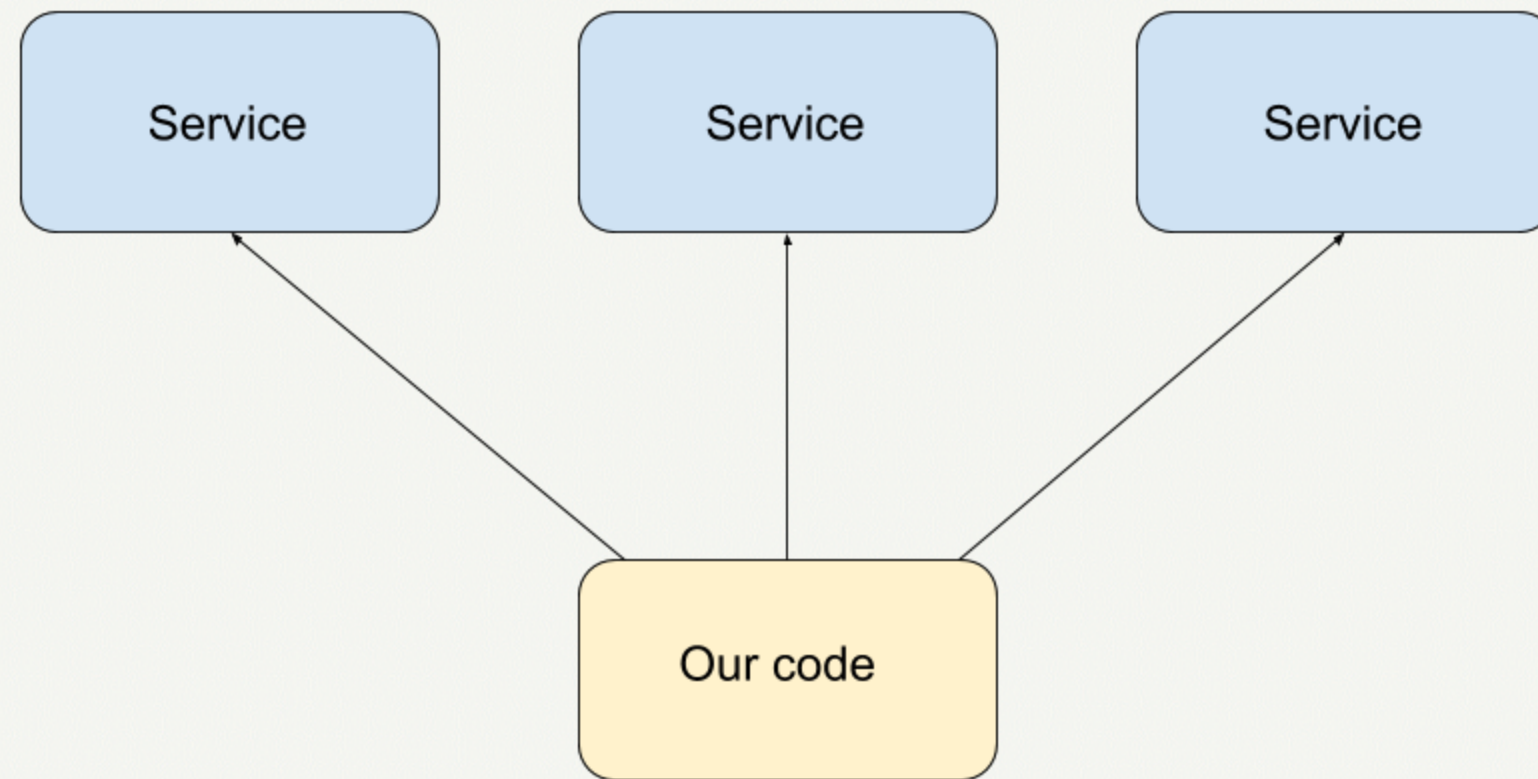
o11c opened this issue on Jul 30, 2015 · 13 comments



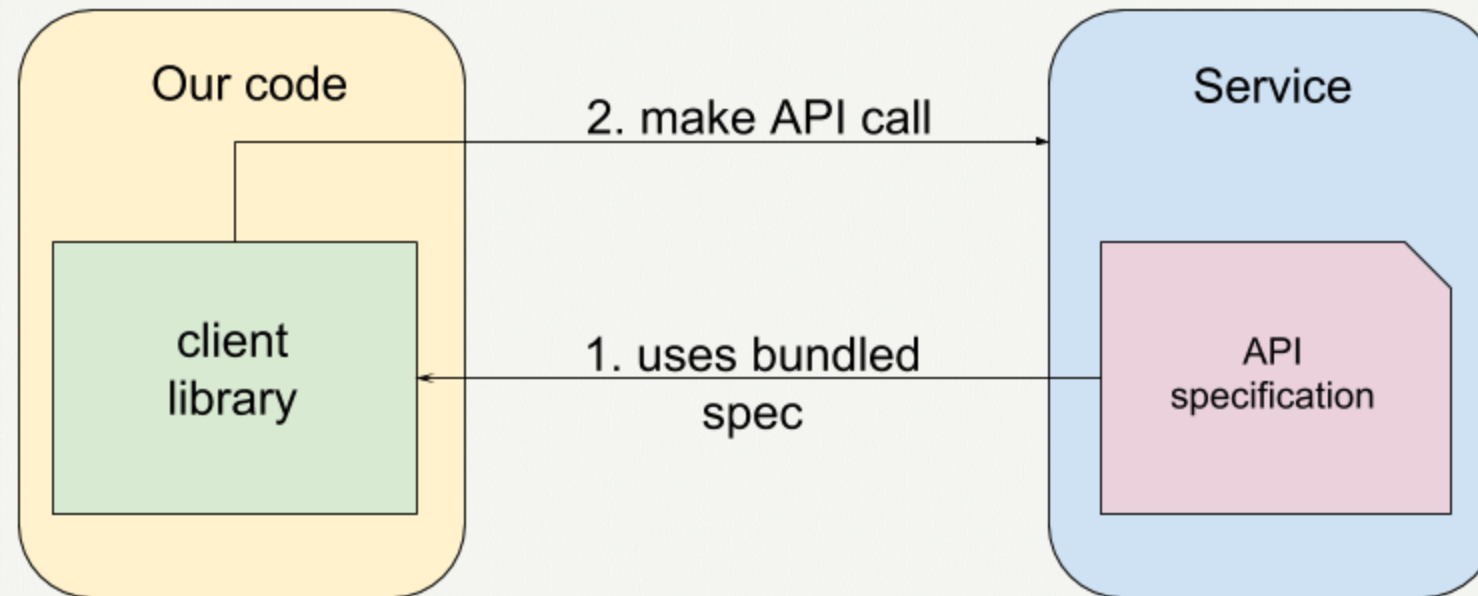
TYPE ANNOTATIONS WITH DISTRIBUTED CODE



SERVICE ORIENTED ARCHITECTURE



ANATOMY OF A SERVICE CALL



THE OPENAPI SPEC

```
/business/{business_id}/v1:  
  get:  
    operationId: business_info  
    parameters:  
      - $ref: '#/parameters/AcceptLanguage'  
      - description: Business identifier  
        in: path  
        name: business_id  
        required: true  
        type: int  
    responses:  
      '200':  
        schema:  
          $ref: '#/definitions/Business'  
...  

```



AN OPENAPI MODEL

```
Business:
  properties:
    address1:
      type: string
    address2:
      type: string
    alias:
      type: string
    has_business_upgrades:
      type: boolean
    review_rating:
      type: string
```



MAKING A SERVICE CALL

```
from business_clientlib.client import create_client  
  
client = create_client(...)  
  
business = client.business.business_info(  
    business_id=business_id,  
).result(timeout=TIMEOUT)  
  
return business.review_rating
```



TESTING OUR NETWORK CODE

```
def get_business_review_rating(business_id: int) -> float:
    business = client.business.business_info(
        business_id=business_id,
    ).result(timeout=TIMEOUT)

    return business.review_rating
```

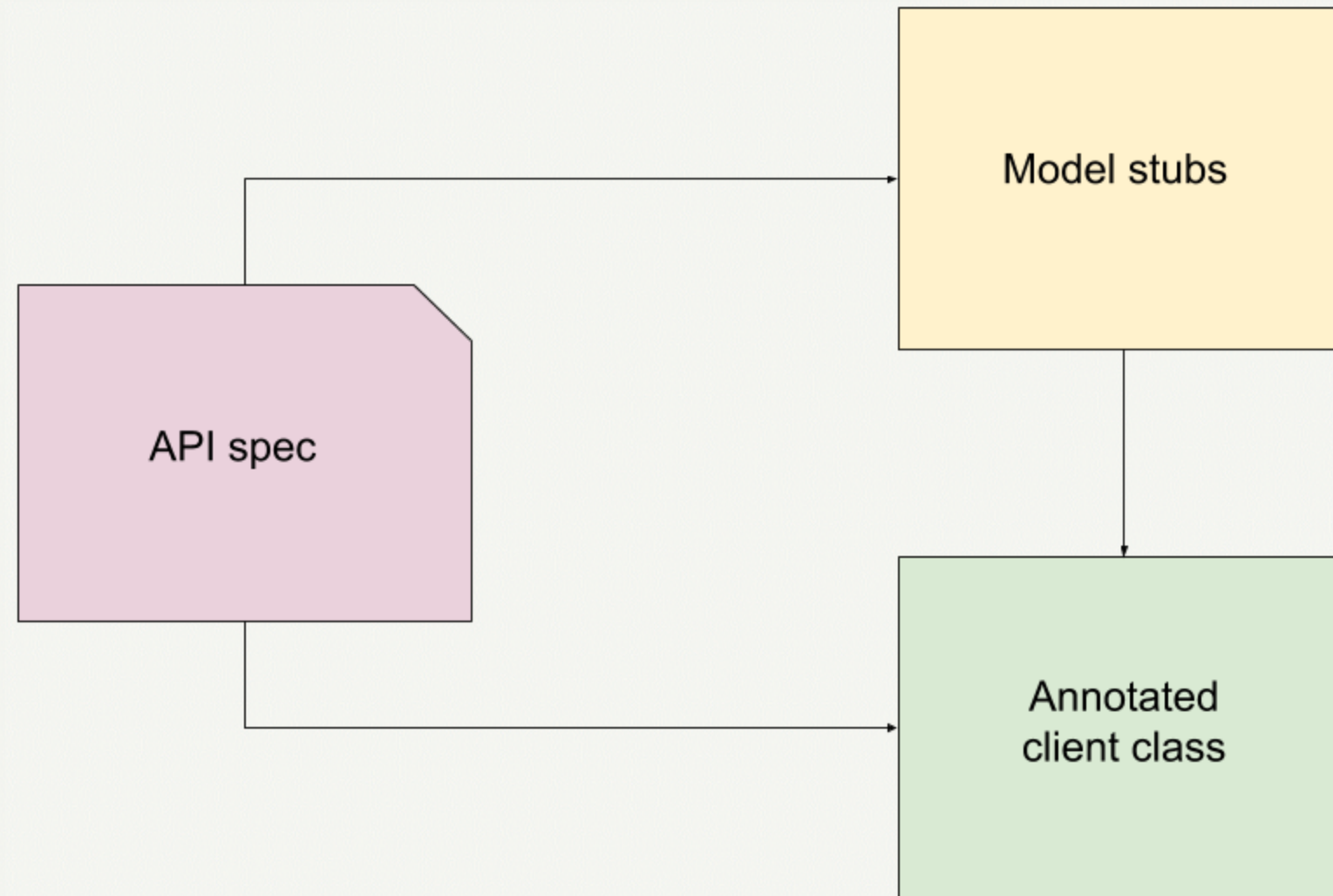
```
def test_get_business_review_rating():
    mock_business = mock.Mock(review_rating=4.5)
    with mock.patch('my_package.client') as client:
        client.business.business_info.return_value.\
            result.return_value = mock_business

        review_rating = get_business_review_rating(5)

    assert review_rating == mock_business.review_rating
```



GENERATING TYPED OBJECTS AND FUNCTIONS



GENERATING MODEL ANNOTATIONS

```
class Business():  
    id: int  
    address1: str  
    address2: Optional[str]  
    review_rating: str
```



TESTING WITH TYPE SAFE DATA MODEL OBJECTS

```
def get_business_review_rating(business_id: int) -> float:
    business = client.business.business_info(
        business_id=business_id,
    ).result(timeout=TIMEOUT)

    return business.review_rating
```

```
def test_get_business_review_rating():
    mock_business = models.Business(review_rating=4.5)
    with mock.patch('get_business_future') as mock_future:
        mock_future.return_value.\
            result.return_value = mock_business

        review_rating = get_business_review_rating(5)

    assert review_rating == mock_business.review_rating
```

```
error: Argument 1 to "Business" has incompatible
type "float"; expected "str"
```



ANNOTATING THE CLIENT CLASS

```
business = client.business.business_info(  
    business_id=business_id,  
).result(timeout=TIMEOUT)
```

```
T = TypeVar('T')
```

```
class BusinessServiceClient:  
    business: business_resource
```

```
class business_resource:  
    def business_info(  
        self,  
        business_id: int,  
    ) -> HttpFuture[Business]: ...
```

```
class HttpFuture(Generic[T]):  
    def result(self, timeout: Optional[float]=None) -> T:  
    ...
```



TAKE AWAYS

- Annotate your code to **improve documentation** and **catch bugs earlier**
- With fine-grained typed data structures you gain a lot of insights into the data flow of your application
- Potentially reduce the number of tests you have to write
- Make the tests you do write more correct and comprehensive, and therefore **more valuable**
- You can use **generated annotations** to type check communication across network boundaries





fb.com/YelpEngineers



[@YelpEngineering](https://twitter.com/YelpEngineering)



engineeringblog.yelp.com



github.com/yelp

A large, diverse group of young adults, likely students or young professionals, are gathered together, cheering and raising their hands in excitement. They are dressed in casual attire, and the overall atmosphere is one of high energy and enthusiasm. The group is composed of people of various ethnicities and ages, all appearing to be in a celebratory mood.

We're Hiring!

www.yelp.com/careers/

THANK YOU!

github.com/sjaensch/type_annotations_talk

@s_jaensch

