# Writing Custom Tracing Functions

Tom Offermann
October 1, 2016
PyDX

## Where to Find Me

- **Work**: New Relic
- **Github**: toffer
- **Twitter**: toffermann
- **Email**: tom@offermann.us
- **Web**: www.offermann.us

## What is Tracing?

A "running narrative" of the line-by-line execution of your code.

## Why Tracing?

- Understand your program
- Powerful, but underutilized tool
- Better understand Python

## Agenda

1. Simple Tracing
2. How Does Tracing Work?
3. Custom Tracing
4. Who Else Uses Custom Tracing?
5. Full-Fledged Example

## Agenda

1. **Simple Tracing**
2. How Does Tracing Work?
3. Custom Tracing
4. Who Else Uses Custom Tracing?
5. Full-Fledged Example

# Example: parent_child.py

```python
def child():
    for i in range(3):
        print(' ' * 8, 'in child loop')

def parent():
    for i in range(2):
        print(' ' * 4, 'in parent loop')
        child()

if __name__ == '__main__':
    print('main start')
    parent()
    print('main end')
```

## Output: parent_child.py

```
$ python parent_child.py

main start
     in parent loop
         in child loop
         in child loop
         in child loop
     in parent loop
         in child loop
         in child loop
         in child loop
main end
```

## How to Trace

```
$ python -m trace --trace parent_child.py
```

## Trace Output

```
--- modulename: parent_child, funcname: <module>
parent_child.py(3): def child():
parent_child.py(7): def parent():
parent_child.py(12): if __name__ == '__main__':
parent_child.py(13):     print('main start')
main start
parent_child.py(14):     parent()
 --- modulename: parent_child, funcname: parent
parent_child.py(8):     for i in range(2):
parent_child.py(9):         print(' ' * 4, 'in parent loop')
     in parent loop
...
```

## Trace Output

```
--- modulename: parent_child, funcname: <module>
parent_child.py(3): def child():
parent_child.py(7): def parent():
parent_child.py(12): if __name__ == '__main__':
parent_child.py(13):     print('main start')
main start
parent_child.py(14):     parent()
 --- modulename: parent_child, funcname: parent
parent_child.py(8):     for i in range(2):
parent_child.py(9):         print(' ' * 4, 'in parent loop')
     in parent loop
...
```

# Trace Output

```
--- modulename: parent_child, funcname: <module>
parent_child.py(3): def child():
parent_child.py(7): def parent():
parent_child.py(12): if __name__ == '__main__':
parent_child.py(13):     print('main start')
main start
parent_child.py(14):     parent()
 --- modulename: parent_child, funcname: parent
parent_child.py(8):     for i in range(2):
parent_child.py(9):         print(' ' * 4, 'in parent loop')
     in parent loop
...
```

## Example: teapot.py

```python
import requests

def teapot():
    url = 'http://httpbin.org/status/418'
    resp = requests.get(url)
    print(resp.status_code, resp.reason)
    print(resp.text)

teapot()
```

# Output: teapot.py

```
$ python teapot.py
418 I'M A TEAPOT

    -=[ teapot ]=-


       _...._
      ,      `
     .   _ _   .
    | ."` ^ `". _,
    \_;`"---"`|//
      |       ;/
      \_     _/
        `"""`
```

## Trace teapot.py

```
$ python -m trace --trace teapot.py
```

## Trace Output (Start)

```
 --- modulename: origin, funcname: <module>
origin.py(3): import requests
 --- modulename: _bootstrap, funcname: _find_and_load
<frozen importlib._bootstrap>(968):  --- modulename:
_bootstrap, funcname: __init__
<frozen importlib._bootstrap>(160): <frozen
importlib._bootstrap>(161):  --- modulename: _bootstrap,
funcname: __enter__
<frozen importlib._bootstrap>(164): <frozen
importlib._bootstrap>(165):  --- modulename: _bootstrap,
funcname: _get_module_lock
...
```

## Trace Output (End)

```
models.py(801):         return content
    -=[ teapot ]=-

       _...._
     .'  _ _ `.
    | ."` ^ `". _,
    \_;`"---"`|//
      |       ;/
      \_     _/
        `"""`
 --- modulename: response, funcname: closed
response.py(408):         if self._fp is None:
response.py(410):         elif hasattr(self._fp, 'closed'):
response.py(411):             return self._fp.closed
 --- modulename: trace, funcname: _unsettrace
trace.py(77):         sys.settrace(None)
```

## Standard Library

```
$ python -m trace --trace teapot.py |
>     grep client.py |  head -5

client.py(69): """
client.py(71): import email.parser
<frozen ...: client.py(72): import email.message
client.py(73): import http
client.py(74): import io
```

# Requests

```
$ python -m trace --trace teapot.py |
>      grep api.py |  head -5

api.py(11): """
api.py(13): from . import sessions
<frozen ...: api.py(16): def request(method, ...)
api.py(59): def get(url, params=None, **kwargs):
api.py(73): def options(url, **kwargs):
```

## Lots of Trace Output

```
$ python -m trace --trace teapot.py | wc -l
   250230
```

## Lots of Trace Output

```
$ python -m trace --trace teapot.py |
>     grep -v ' ---'|       # Ignore function entry
>     cut -f 1 -d '(' |     # Grab module name
>     sort | uniq -c |      # Count by module
>     sort -rn | head -n 5  # Show top 5
```

```
120836 sre_parse.py
 36957 sre_compile.py
 11800 enum.py
  4325 ipaddress.py
  3245 entities.py
```

## Example: teapot_trace.py

```python
import requests
from trace import Trace

def teapot():
    url = 'http://httpbin.org/status/418'
    resp = requests.get(url)
    print(resp.status_code, resp.reason)
    print(resp.text)

Trace().runfunc(teapot)
```

## More Focused Results

```
$ python teapot_trace.py | wc -l
```

```
   13829
```

## Summary: Simple Tracing

- Built in, so always available
- Can run it from command line
- Extremely verbose!
- Use **`Trace()`** for focused results

## Agenda

## How Does Tracing Work?

1. Create a tracing function
2. Register the tracing function
3. Python will run tracer before executing each line of code.

## Create a Tracing Function

## Trace Line

```
class Trace:
    ...
    def localtrace_trace(self, frame, why, arg):
        if why == "line":
            filename = frame.f_code.co_filename
            bname = os.path.basename(filename)
            lineno = frame.f_lineno
            line = linecache.getline(filename, lineno)
            print("%s(%d): %s" % (bname, lineno, line))
        return self.localtrace_trace
```

## Trace Line: Arguments

```
class Trace:
    ...
    def localtrace_trace(self, frame, why, arg):
        if why == "line":
            filename = frame.f_code.co_filename
            bname = os.path.basename(filename)
            lineno = frame.f_lineno
            line = linecache.getline(filename, lineno)
            print("%s(%d): %s" % (bname, lineno, line))
        return self.localtrace_trace
```

## Trace Line: Return value

```
class Trace:
    ...
    def localtrace_trace(self, frame, why, arg):
        if why == "line":
            filename = frame.f_code.co_filename
            bname = os.path.basename(filename)
            lineno = frame.f_lineno
            line = linecache.getline(filename, lineno)
            print("%s(%d): %s" % (bname, lineno, line))
        return self.localtrace_trace
```

## Argument #1: Frame

```
class Trace:
    ...
    def localtrace_trace(self, frame, why, arg):
        if why == "line":
            filename = frame.f_code.co_filename
            bname = os.path.basename(filename)
            lineno = frame.f_lineno
            line = linecache.getline(filename, lineno)
            print("%s(%d): %s" % (bname, lineno, line))
        return self.localtrace_trace
```

## What is a Frame?

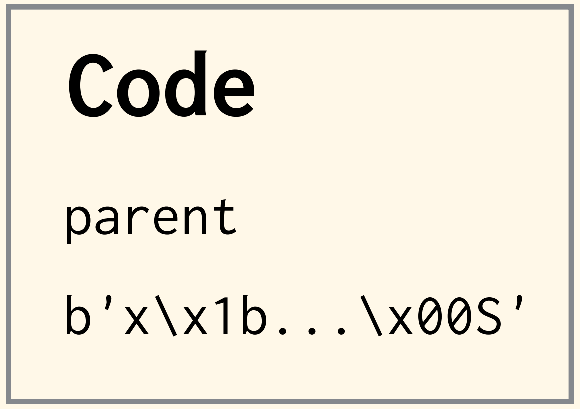
## (Detour ahead!)

## How Python executes code

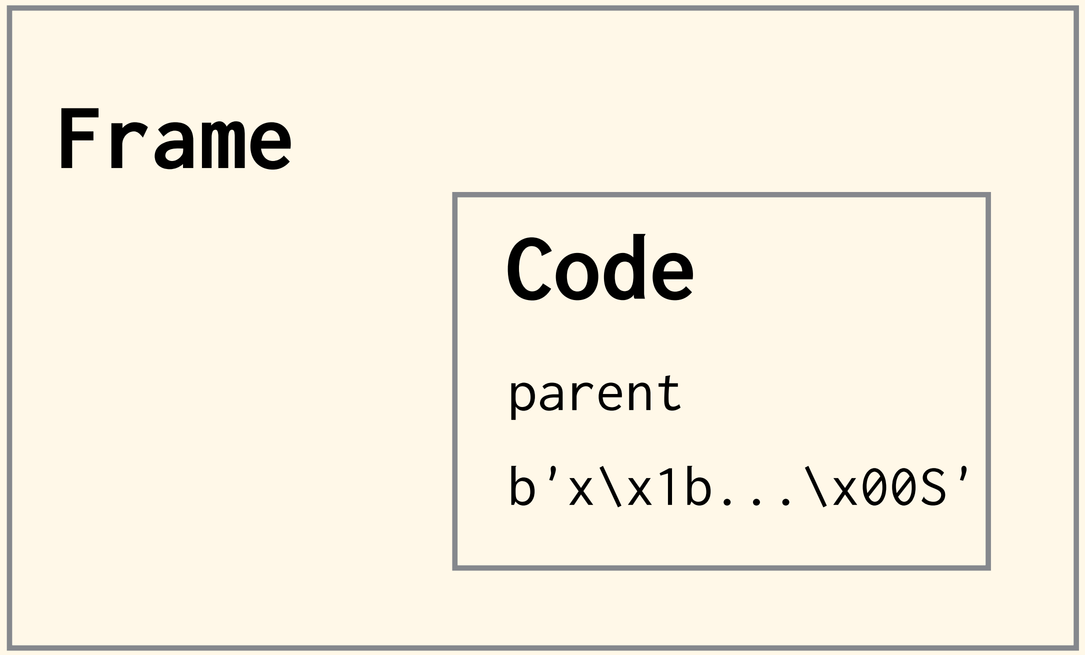
```
def parent():
    for i in range(3):
        child()
    return 'parent done'
```

```
b'x\x1b\x00t\x00\x00d\x01\x00\x83\x01\x00D]\r
\x00}\x00\x00t\x01\x00\x83\x00\x00\x01q\r\x00Wd
\x02\x00S'
```

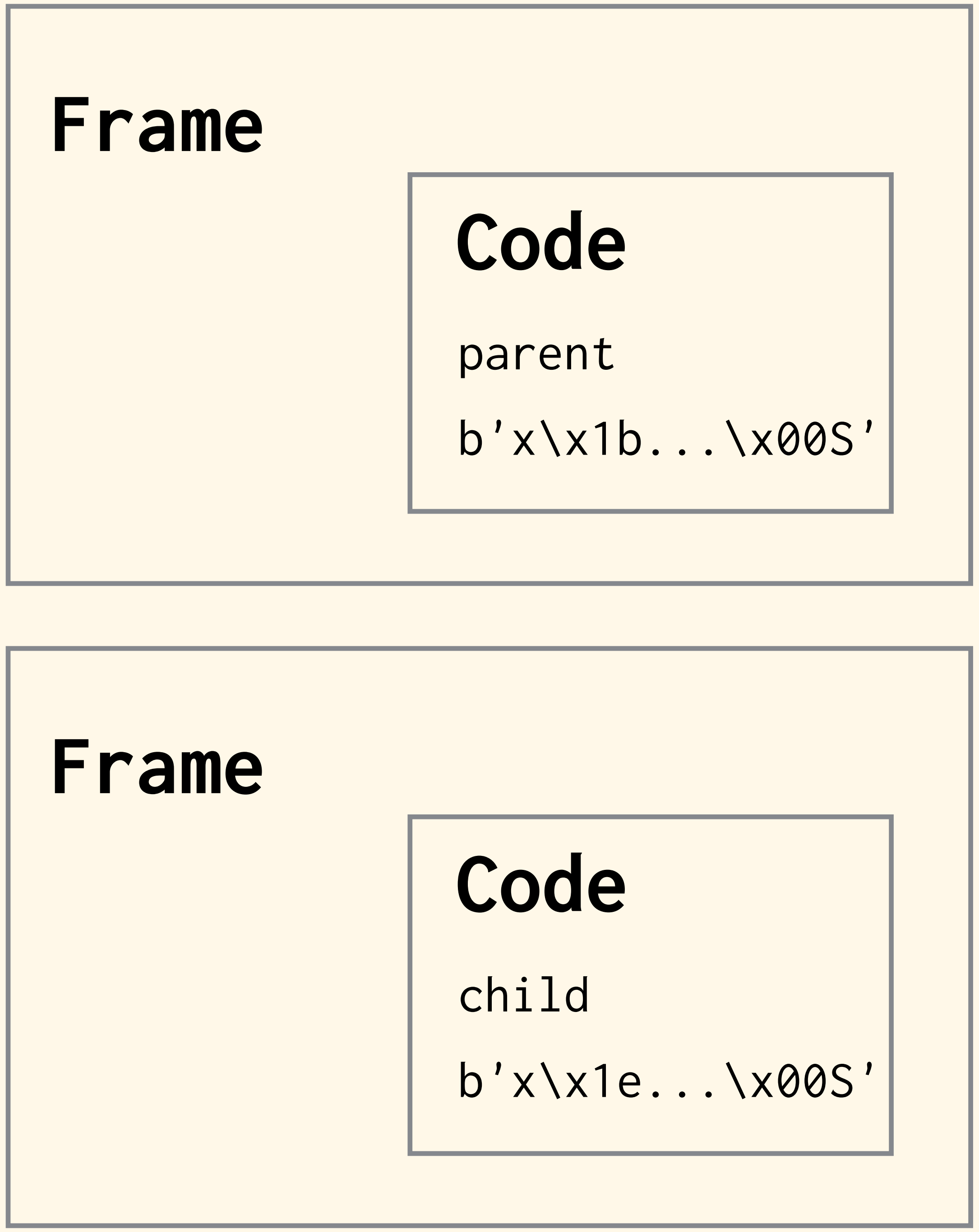
## Code Object



## Frame Object



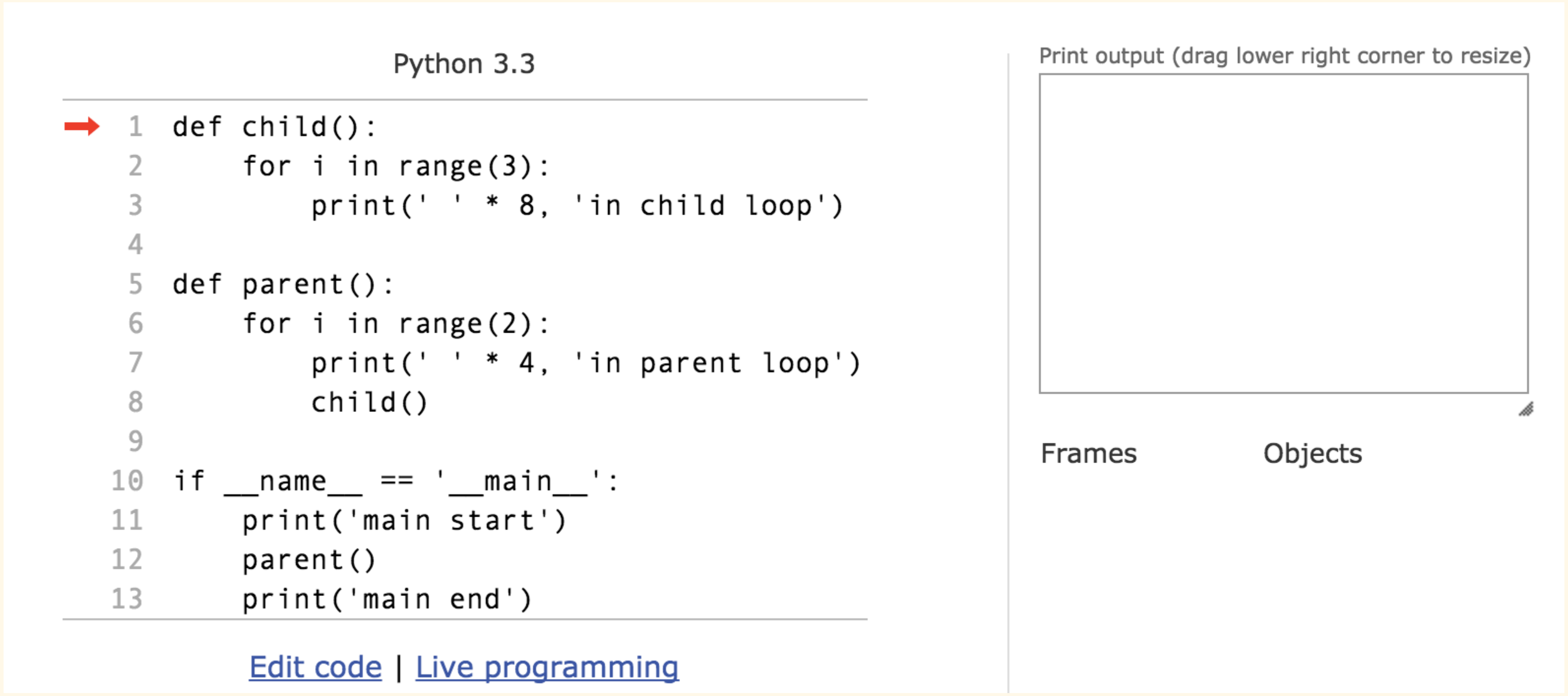
## Stack



## Summary: Frames

- Source code compiled to bytes
- Python interpreter runs byte-compiled code
- Byte-compiled code is in "code object"
- Code object is in "execution frame"
- New frame created for every function call.
- Frames exist on "Stack"

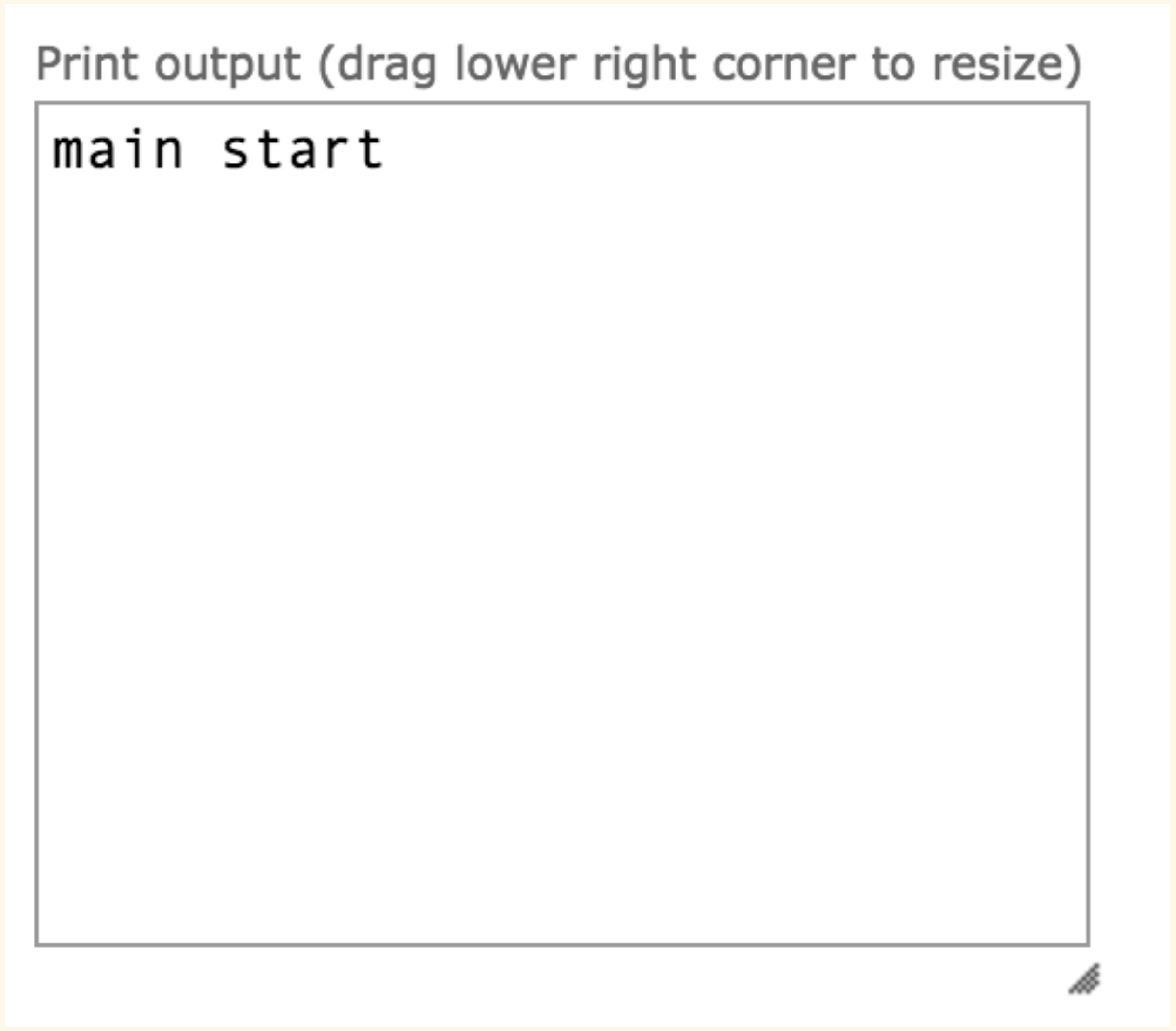
# Stack Visualization



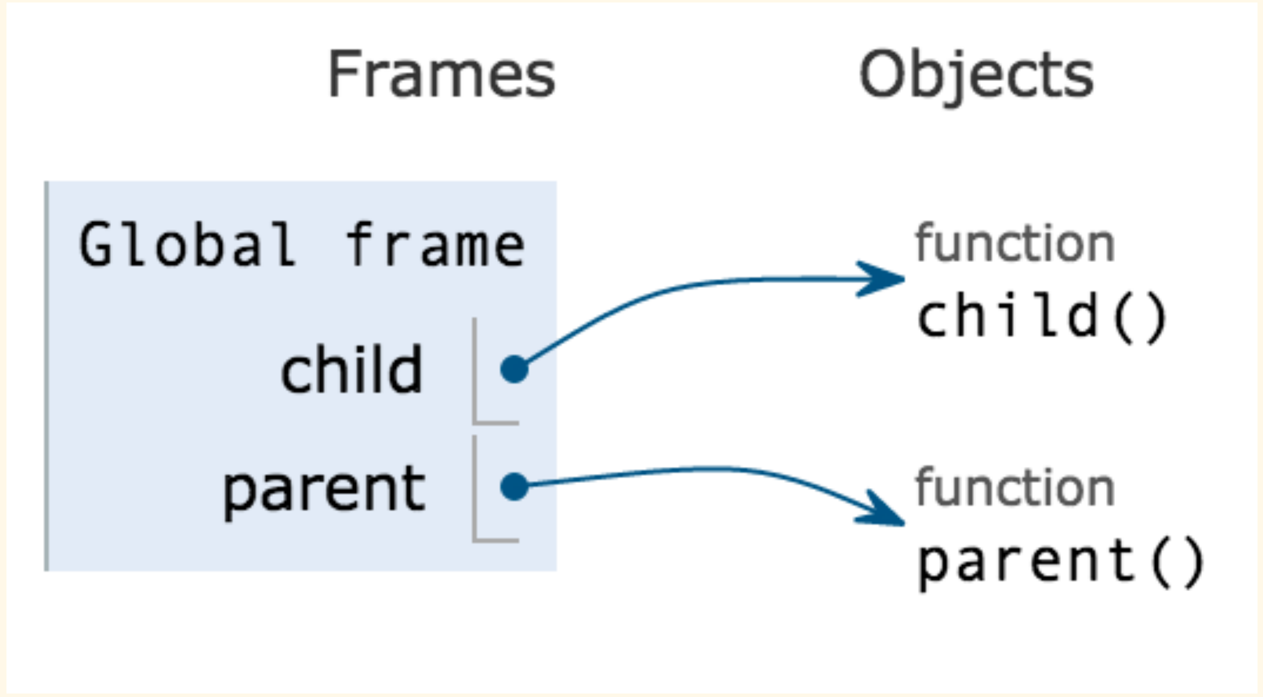
```
def child():
    for i in range(3):
        print(' ' * 8, 'in child loop')

def parent():
    for i in range(2):
        print(' ' * 4, 'in parent loop')
        child()

if __name__ == '__main__':
    print('main start')
    parent()
    print('main end')
```

Edit code | Live programming
Print output (drag lower right corner to resize)
Frames
Objects

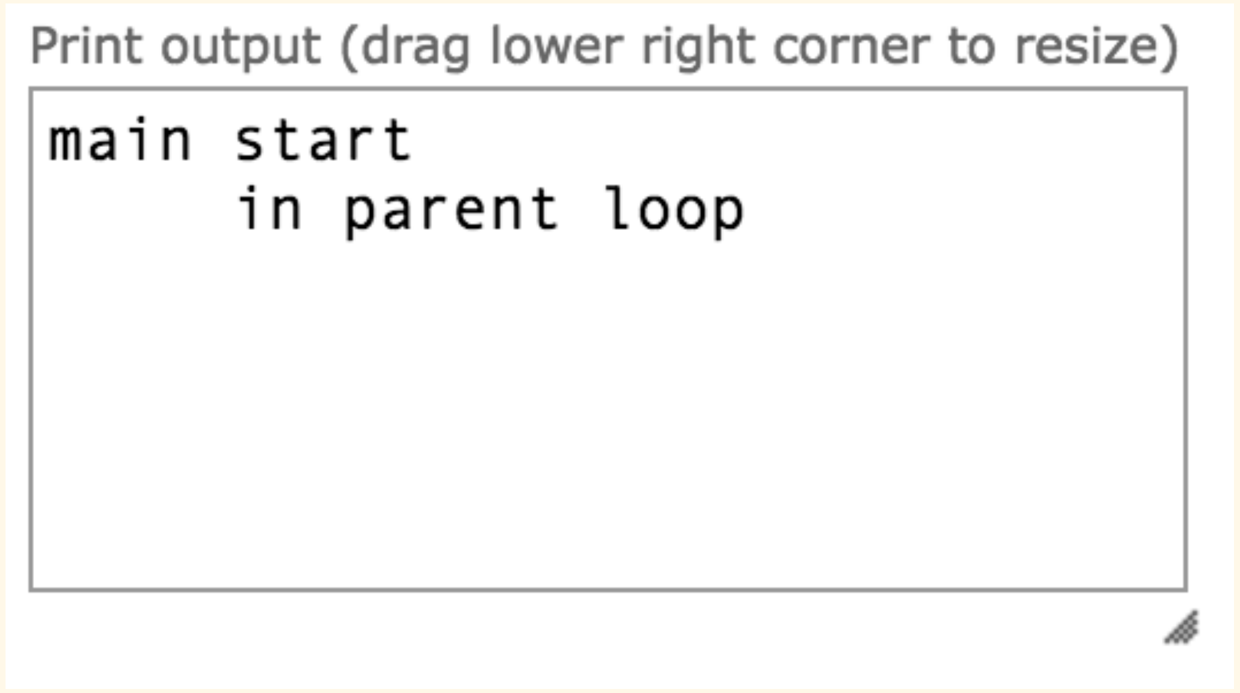
## pythontutor.com #1


Print output (drag lower right corner to resize)

```
main start
```

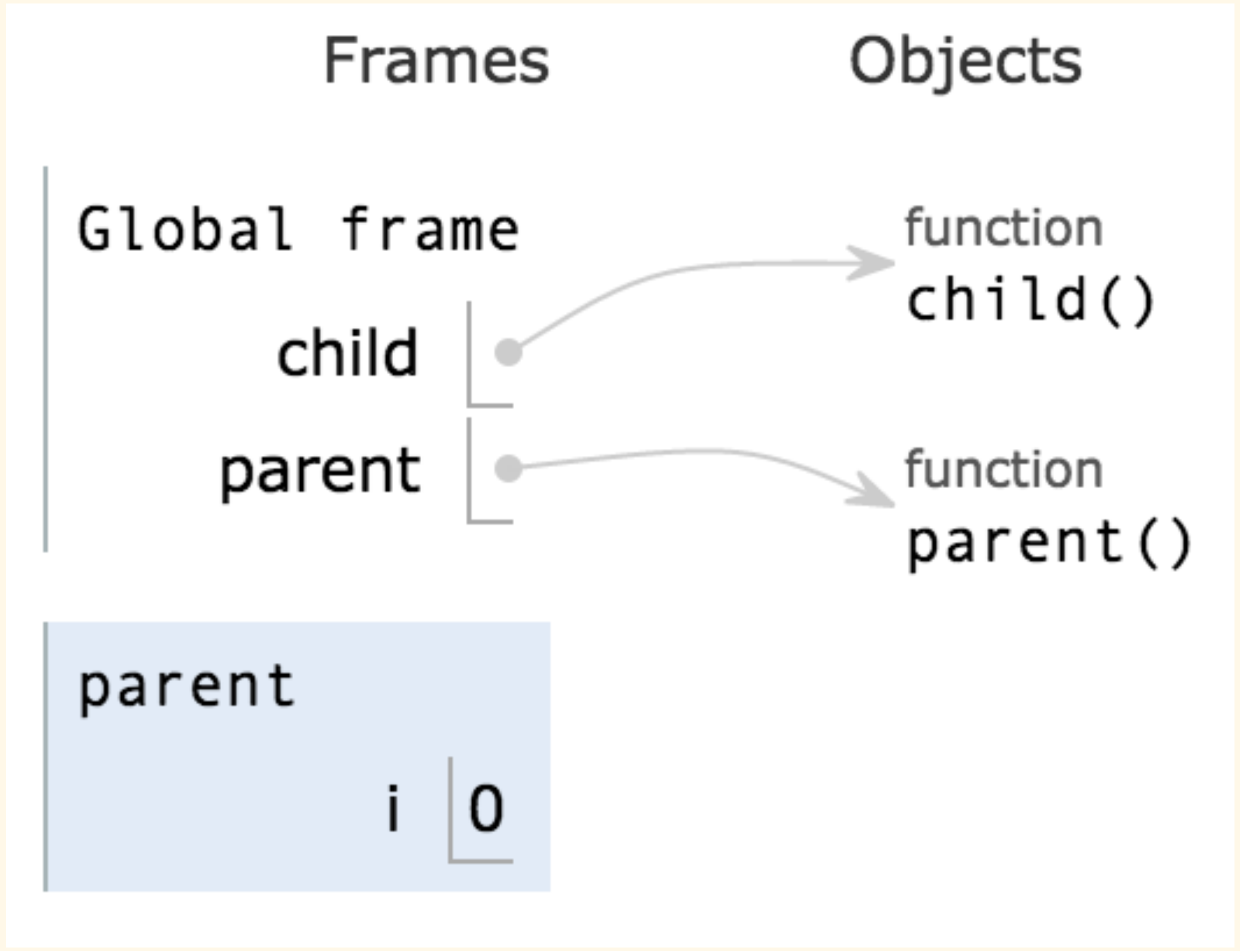


# pythontutor.com #2

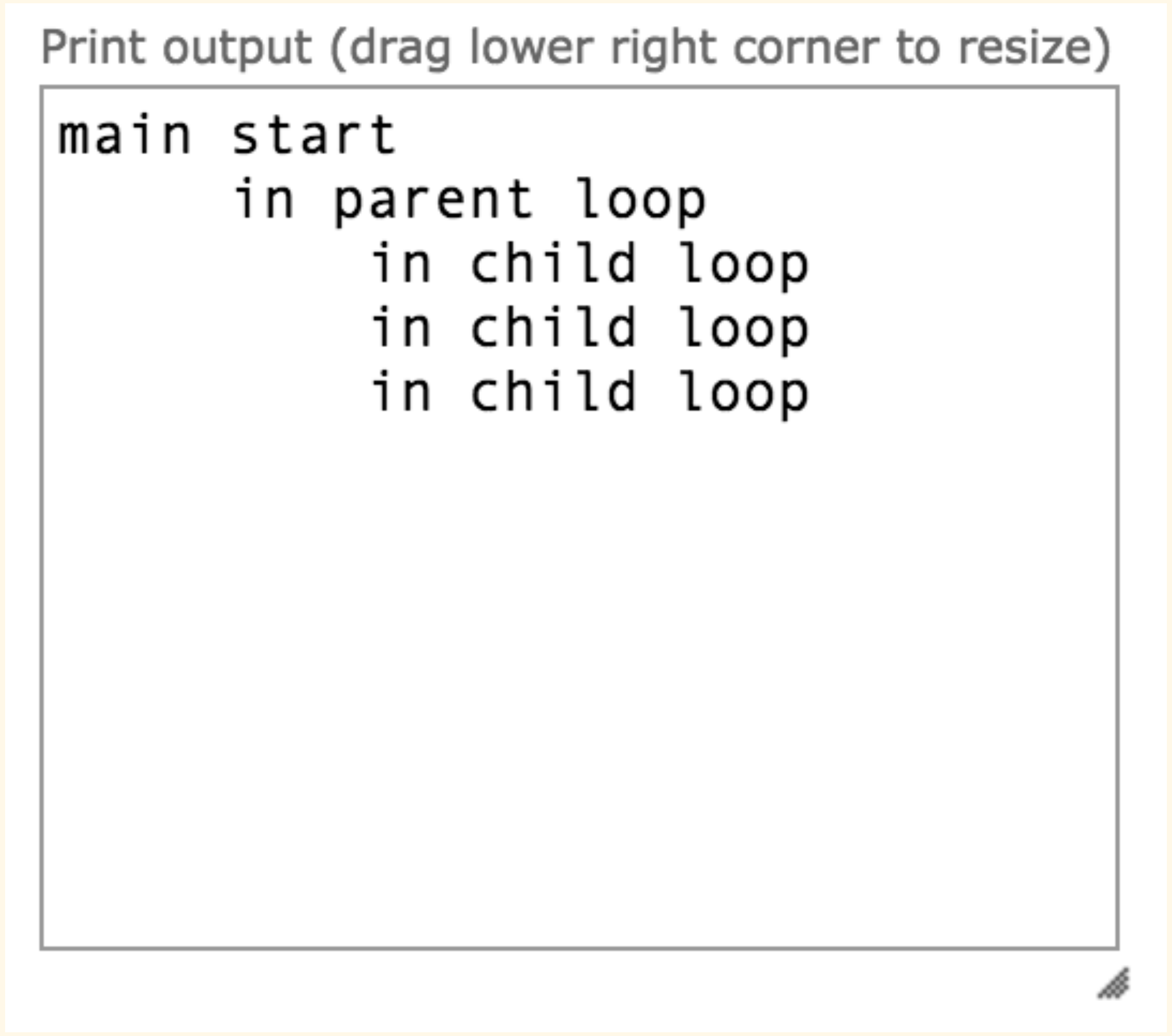


Print output (drag lower right corner to resize)

```
main start
     in parent loop
```

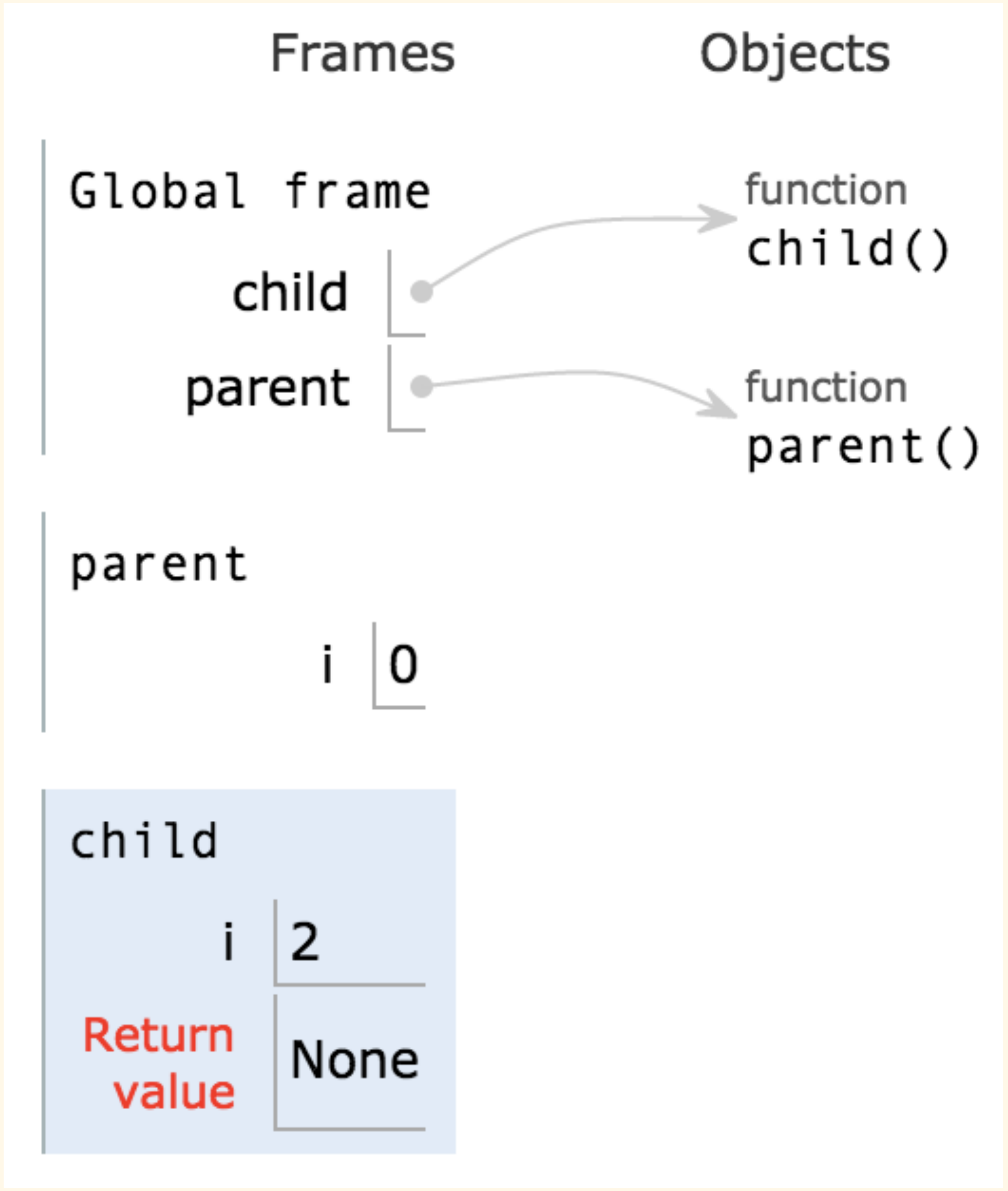


# pythontutor.com #3

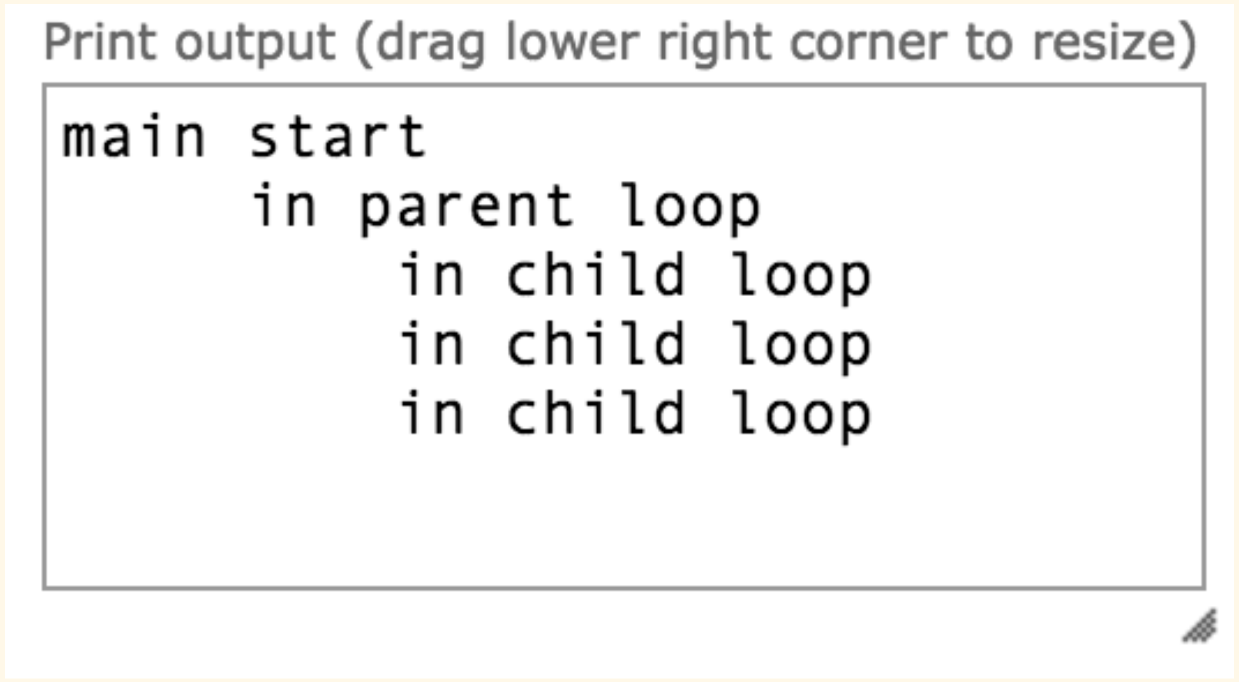


Print output (drag lower right corner to resize)

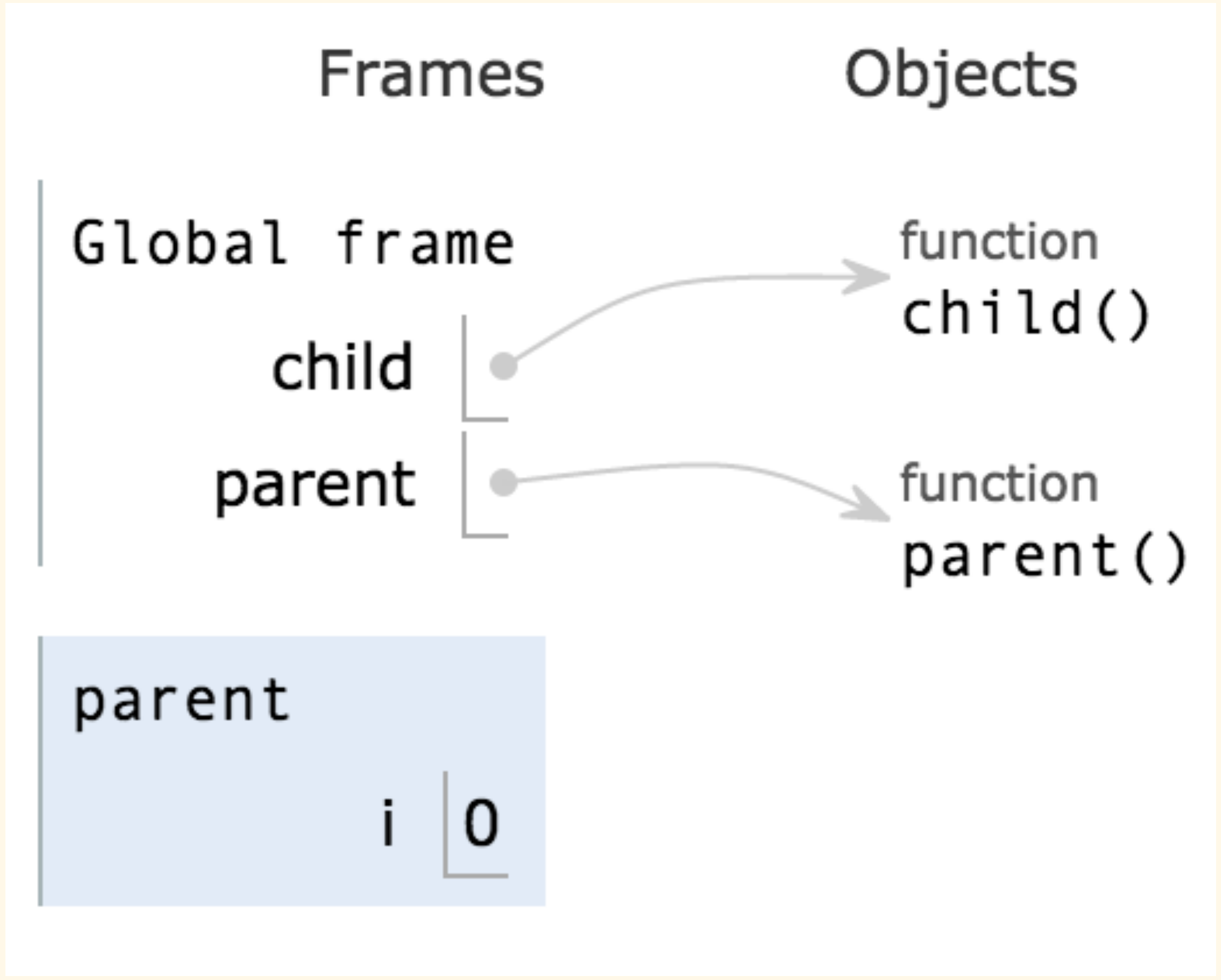
```
main start
     in parent loop
         in child loop
         in child loop
         in child loop
```



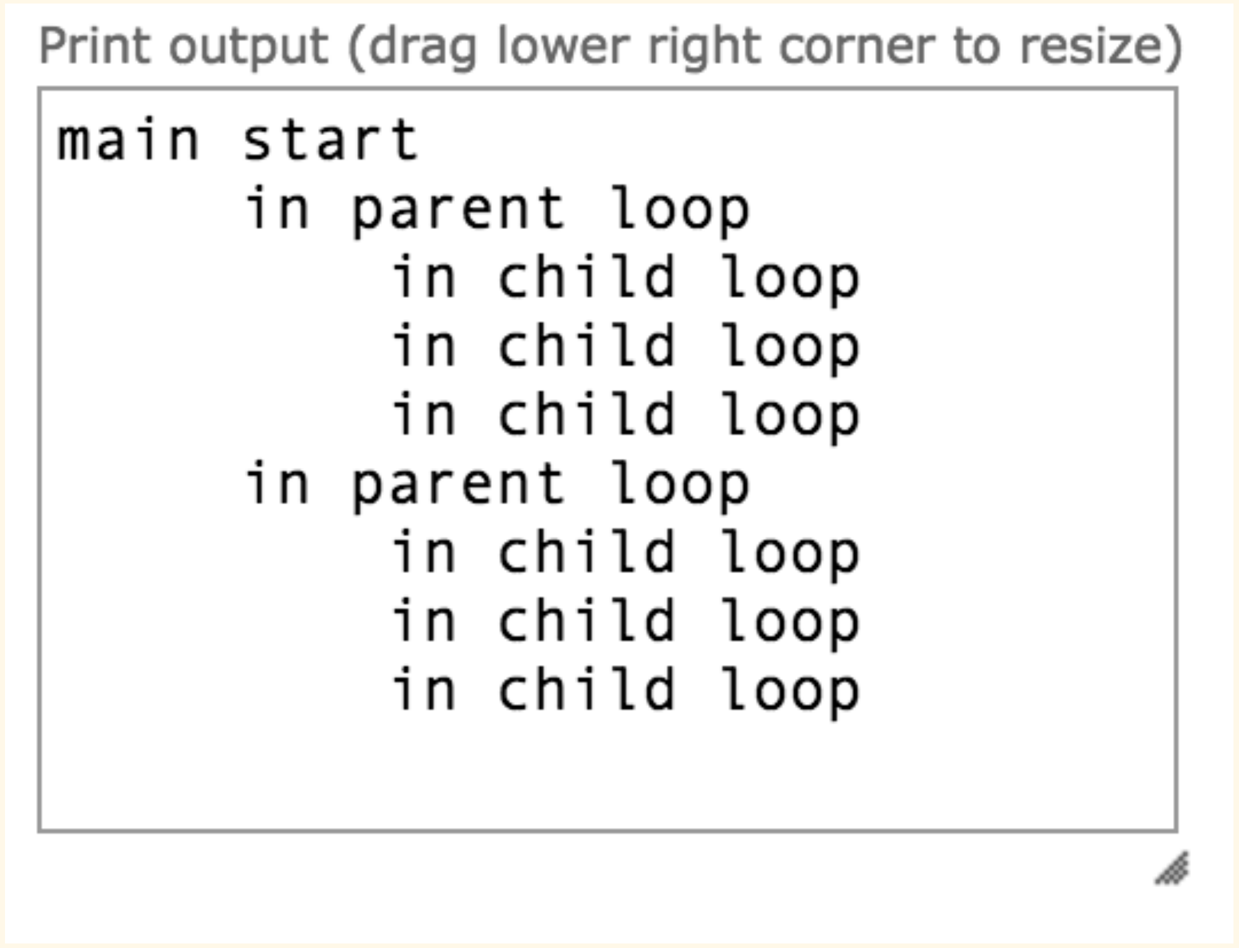
# pythontutor.com #4


Print output (drag lower right corner to resize)

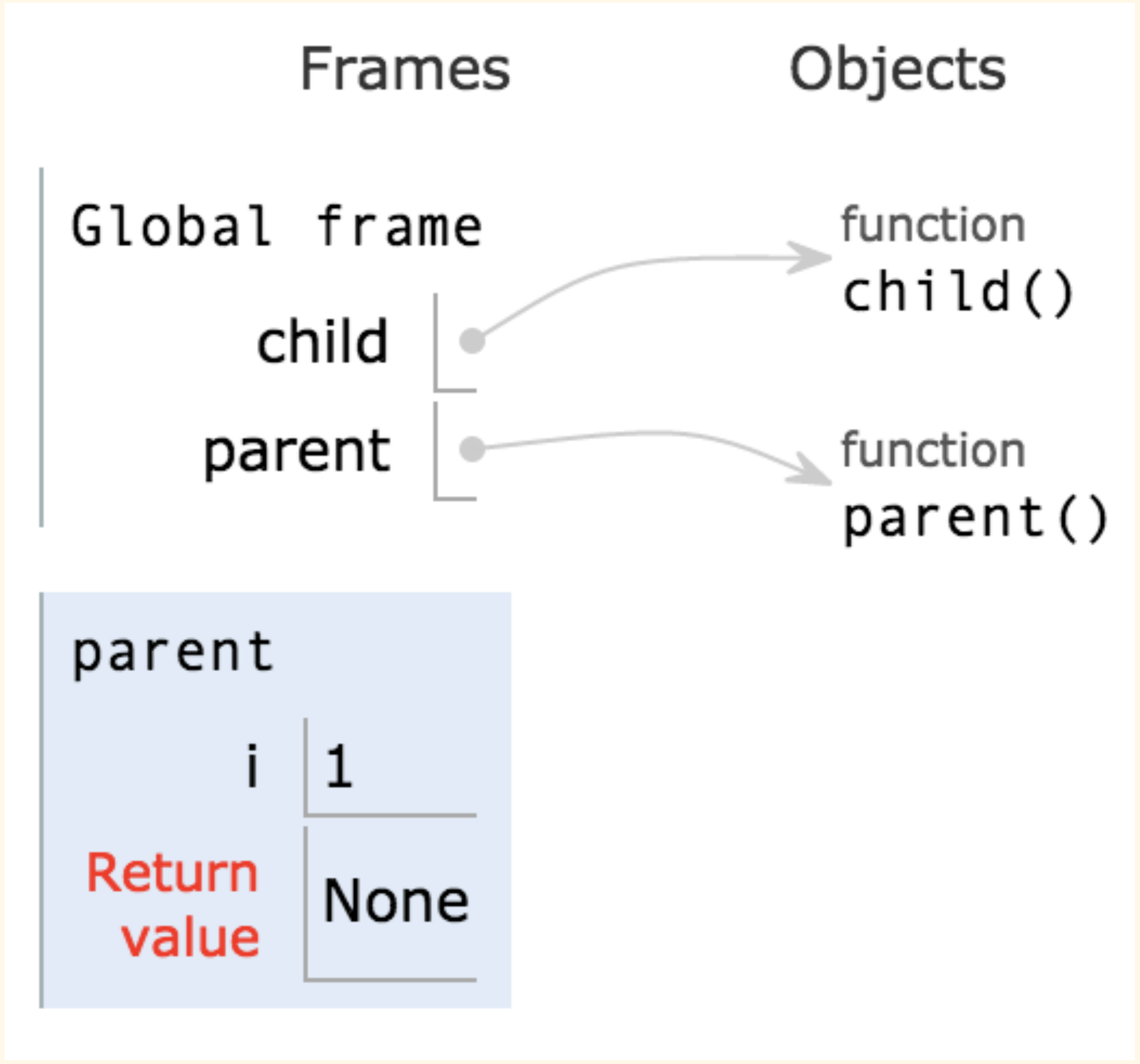
```
main start
     in parent loop
          in child loop
          in child loop
          in child loop
```



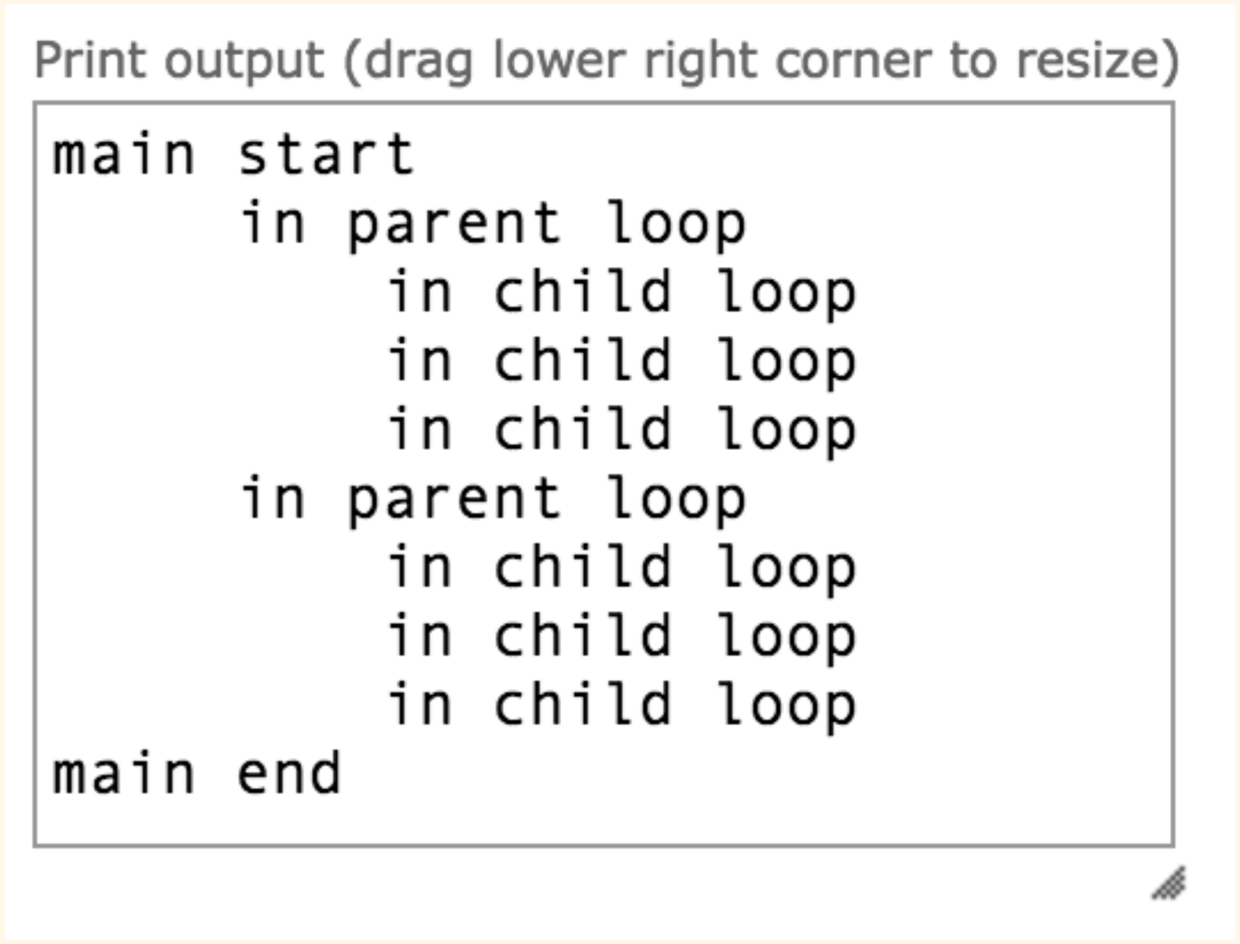
# pythontutor.com #5



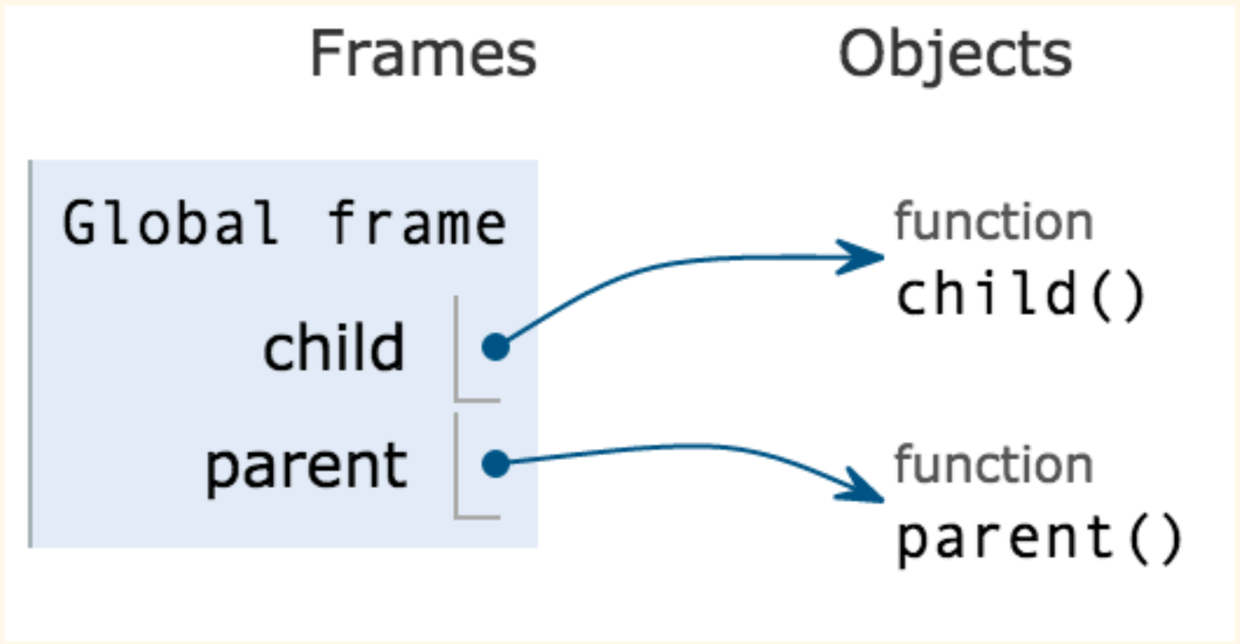
```
main start
     in parent loop
         in child loop
         in child loop
         in child loop
     in parent loop
         in child loop
         in child loop
         in child loop
```



# pythontutor.com #6


Print output (drag lower right corner to resize)

```
main start
     in parent loop
         in child loop
         in child loop
         in child loop
     in parent loop
         in child loop
         in child loop
         in child loop
main end
```



## Frames are objects

## Frame Attributes

```
>>> help(sys._getframe())

class frame(object)
 ...
 |   f_back       # this frame's caller
 |   f_builtins   # builtins seen by this frame
 |   f_code       # code object being executed
 |   f_globals    # globals seen by this frame
 |   f_lasti      # index of last instruction
 |   f_lineno     # current line number in source
 |   f_locals     # locals seen by this frame
 |   f_trace      # tracing function for this frame
```

## Code objects are objects, too!

# Code Object Attributes

```
>>> help(sys._getframe().f_code)

class code(object)
...
 |   co_argcount      # Number of args
 |   co_code          # String of bytecode
 |   co_filename      # Name of file
 |   co_firstlineno   # First source line number
 |   co_name          # Name of code object
 |   co_names         # Tuple of local var names
 |   co_nlocals       # Number of local vars
 |   co_varnames      # Names of args and local vars
```

## Detour Finished!

## Arguments #2 & 3: Why and Arg

```
class Trace:
    ...
    def localtrace_trace(self, frame, why, arg):
        if why == "line":
            filename = frame.f_code.co_filename
            bname = os.path.basename(filename)
            lineno = frame.f_lineno
            line = linecache.getline(filename, lineno)
            print("%s(%d): %s" % (bname, lineno, line))
        return self.localtrace_trace
```

## Why and Arg

| Why (or Event) | Argument Value |
|---|---|
| call | None |
| line | None |
| return | Value being returned |
| exception | The (exception, val, tb) tuple |

Source:
Doug Hellmann's Python 3 Module of the Week Series

- https://pymotw.com/3/sys/tracing.html

## Using Frame, Event and Arg

## Trace Line

```python
def localtrace_trace(self, frame, why, arg):
    if why == "line":
        filename = frame.f_code.co_filename
        bname = os.path.basename(filename)
        lineno = frame.f_lineno
        line = linecache.getline(filename, lineno)

        print("%s(%d): %s" % (bname, lineno, line),
                end='')

    return self.localtrace
```

## Check Event

```
def localtrace_trace(self, frame, why, arg):
    if why == "line":
        filename = frame.f_code.co_filename
        bname = os.path.basename(filename)
        lineno = frame.f_lineno
        line = linecache.getline(filename, lineno)

        print("%s(%d): %s" % (bname, lineno, line),
                end='')

    return self.localtrace
```

## Access frame attribute

```
def localtrace_trace(self, frame, why, arg):
    if why == "line":
        filename = frame.f_code.co_filename
        bname = os.path.basename(filename)
        lineno = frame.f_lineno
        line = linecache.getline(filename, lineno)

        print("%s(%d): %s" % (bname, lineno, line),
                end='')

    return self.localtrace
```

## Access code attribute

```
def localtrace_trace(self, frame, why, arg):
   if why == "line":
       filename = frame.f_code.co_filename
       bname = os.path.basename(filename)
       lineno = frame.f_lineno
       line = linecache.getline(filename, lineno)

       print("%s(%d): %s" % (bname, lineno, line),
               end='')

   return self.localtrace
```

## How do I register tracer?

## Use sys.settrace()

```python
import sys
from trace import my_tracing_func

def child():
    print('child')

def parent():
    child()

if __name__ == '__main__':
    sys.settrace(my_tracing_func)
    parent()
```

## Why Return a Tracer Function?

## Why Return a Tracer Function?

- You can have multiple tracer functions!
    - "System" Tracer
    - "Local" Tracer

## When does each tracer run?

1. System tracer runs for "Call" event.
2. Local tracer runs for all other events.

# Frame Attributes (again!)

```
>>> help(sys._getframe())

class frame(object)
  ...
  |   f_back       # this frame's caller
  |   f_builtins   # builtins seen by this frame
  |   f_code       # code object being executed
  |   f_globals    # globals seen by this frame
  |   f_lasti      # index of last instruction
  |   f_lineno     # current line number in source
  |   f_locals     # locals seen by this frame
  |   f_trace      # tracing function for this frame
```

## System and Local Example

## System and Local Tracer

```python
def system_tracer(frame, event, arg):
    if event == 'call':
        func_name = frame.f_code.co_name
        print('... system_tracer: %r' % func_name)
    return local_tracer

def local_tracer(frame, event, arg):
    if event == 'line':
        lineno = frame.f_lineno
        print('local_tracer: lineno %d' % lineno)
    return local_tracer
```

## System and Local Tracer

```python
def system_tracer(frame, event, arg):
    if event == 'call':
        func_name = frame.f_code.co_name
        print('... system_tracer: %r' % func_name)
    return local_tracer

def local_tracer(frame, event, arg):
    if event == 'line':
        lineno = frame.f_lineno
        print('local_tracer: lineno %d' % lineno)
    return local_tracer
```

## Example: count.py

```python
def count_one():
    print('1')
    return 'Done'

def count_two():
    print('one')
    print('two')
    return 'Done'

if __name__ == '__main__':
    sys.settrace(system_tracer)
    count_one()
    count_two()
    sys.settrace(None)
```

## When does each tracer run?

| Code | Event | Tracer |
|---|---|---|
| `def count_one():` | Call | System |

```
def count_one():
    print('1')
    return 'Done'

def count_two():
    print('one')
    print('two')
    return 'Done'
```

## When does each tracer run?

| Code | Event | Tracer |
|---|---|---|
| `print('1')` | Line | Local |

```
def count_one():
    print('1')
    return 'Done'

def count_two():
    print('one')
    print('two')
    return 'Done'
```

## When does each tracer run?

| Code | Event | Tracer |
|---|---|---|
| `def count_one():` | | |
| `    print('1')` | | |
| `    return 'Done'` | Return | Local |
| | | |
| `def count_two():` | | |
| `    print('one')` | | |
| `    print('two')` | | |
| `    return 'Done'` | | |

## When does each tracer run?

| Code | Event | Tracer |
|---|---|---|
| `def count_one():` | | |
| `    print('1')` | | |
| `    return 'Done'` | | |
| | | |
| `def count_two():` | Call | System |
| `    print('one')` | | |
| `    print('two')` | | |
| `    return 'Done'` | | |

# When does each tracer run?

| Code | Event | Tracer |
|---|---|---|
| `print('one')` | Line | Local |

```
def count_one():
    print('1')
    return 'Done'

def count_two():
    print('one')
    print('two')
    return 'Done'
```

# When does each tracer run?

| Code | Event | Tracer |
|---|---|---|
| `def count_one():` | | |
| `    print('1')` | | |
| `    return 'Done'` | | |
| | | |
| `def count_two():` | | |
| `    print('one')` | | |
| `    print('two')` | `Line` | `Local` |
| `    return 'Done'` | | |

## When does each tracer run?

**Code** **Event** **Tracer**

```
def count_one():
    print('1')
    return 'Done'

def count_two():
    print('one')
    print('two')
    return 'Done'          Return          Local
```

## Local Tracer can be None

If tracer returns None from "Call" event, no further tracing in frame.

## Recap: How tracing works

1. Create System tracer
   - Signature: (frame, event, arg)
   - Return local tracer, or None
2. Register System tracer
   - sys.settrace(system_tracer)

## Agenda

1. Simple Tracing
2. How Does Tracing Work?
3. **Custom Tracing**
4. Who Else Uses Custom Tracing?
5. Full-Fledged Example

## Example #1
## Print Full Filename Tracer

## Print Full Filename Tracer

```python
def filename_tracer(frame, event, arg):
    fname = frame.f_code.co_filename
    co_name = frame.f_code.co_name
    num = frame.f_lineno
    line = linecache.getline(fname, num)

    if event == 'call':
        logger.info("--- %s: %s", fname, co_name)
    elif event == 'line':
        logger.info("%s(%d): %s", fname, num, line)

    return filename_tracer
```

# Print Full Filename Tracer

```python
def filename_tracer(frame, event, arg):
    fname = frame.f_code.co_filename
    co_name = frame.f_code.co_name
    num = frame.f_lineno
    line = linecache.getline(fname, num)

    if event == 'call':
        logger.info("--- %s: %s", fname, co_name)
    elif event == 'line':
        logger.info("%s(%d): %s", fname, num, line)

    return filename_tracer
```

# Print Full Filename Tracer

```
def filename_tracer(frame, event, arg):
    fname = frame.f_code.co_filename
    co_name = frame.f_code.co_name
    num = frame.f_lineno
    line = linecache.getline(fname, num)

    if event == 'call':
        logger.info("--- %s: %s", fname, co_name)
    elif event == 'line':
        logger.info("%s(%d): %s", fname, num, line)

    return filename_tracer
```

## teapot_filename_trace.py

```python
import sys
import requests
from filename_trace import filename_tracer

def teapot():
    url = 'http://httpbin.org/status/418'
    resp = requests.get(url)
    print(resp.status_code, resp.reason)
    print(resp.text)

if __name__ == '__main__':
    sys.settrace(filename_tracer)
    teapot()
    sys.settrace(None)
```

## Standard Library

```
$ grep client.py logs/trace.log | head -5

--- /.../lib/python3.5/http/client.py: __init__
/.../lib/python3.5/http/client.py(728):         self.timeout...
/.../lib/python3.5/http/client.py(729):         self.source_...
/.../lib/python3.5/http/client.py(730):         self.sock...
/.../lib/python3.5/http/client.py(731):         self._buffer...
```

## Requests

```
$ grep api.py logs/trace.log | head -5

--- /.../lib/python3.5/site-packages/requests/api.py: get
/.../lib/python3.5/site-packages/requests/api.py(69):     kwargs...
/.../lib/python3.5/site-packages/requests/api.py(70):     return...
--- /.../lib/python3.5/site-packages/requests/api.py: request
/.../lib/python3.5/site-packages/requests/api.py(55):     with...
```

## Example #2
## Print Function Arguments

## Print function args

```
def trace_call_args(frame, event, arg):
    if event == 'call':
        code = frame.f_code
        arg_count = code.co_argcount
        arg_names = code.co_varnames[:arg_count]

        print("%s: %s" % (code.co_filename, code.co_name))

        for name in arg_names:
            val = frame.f_locals[name]
            print('{:<4}{}: {}'.format('  ', name, val))

    return None
```

## Print function args

```
def trace_call_args(frame, event, arg):
    if event == 'call':
        code = frame.f_code
        arg_count = code.co_argcount
        arg_names = code.co_varnames[:arg_count]

        print("%s: %s" % (code.co_filename, code.co_name))

        for name in arg_names:
            val = frame.f_locals[name]
            print('{:<4}{}: {}'.format(' ', name, val))

    return None
```

## Print function args

```
def trace_call_args(frame, event, arg):
    if event == 'call':
        code = frame.f_code
        arg_count = code.co_argcount
        arg_names = code.co_varnames[:arg_count]

        print("%s: %s" % (code.co_filename, code.co_name))

        for name in arg_names:
            val = frame.f_locals[name]
            print('{:<4}{}: {}'.format(' ', name, val))

    return None
```

## Example: add.py

```python
import sys
from trace_call_args import trace_call_args

def add(x, y):
    return x + y

def add_with_defaults(x=49, y=50):
    return x + y

if __name__ == '__main__':
    sys.settrace(trace_call_args)
    add(1, 2)
    add_with_defaults()
    sys.settrace(None)
```

## Output: add.py

```
$ python add.py

add.py: add
    x: 1
    y: 2
add.py: add_with_defaults
    x: 49
    y: 50
```

## Example: var_args_1.py

```
import sys
from trace_call_args import trace_call_args

def var_args(*args, **kwargs):
    x = 1
    y = 2
    return x, y

if __name__ == '__main__':
    sys.settrace(trace_call_args)
    args = (1, 2, 'foo')
    kwargs = {'a': 8, 'b': 9}
    var_args(*args, **kwargs)
    sys.settrace(None)
```

## Output: var_args_1.py

```
$ python var_args_1.py

var_args_1.py: var_args
```

## Print function args

```
def trace_call_args(frame, event, arg):
    if event == 'call':
        code = frame.f_code
        arg_count = code.co_argcount
        arg_names = code.co_varnames[:arg_count]

        print("%s: %s" % (code.co_filename, code.co_name))

        for name in arg_names:
            val = frame.f_locals[name]
            print('{:<4}{}: {}'.format(' ', name, val))

    return None
```

## Print function args: Take 2

```
def trace_call_args_2(frame, event, arg):
    if event == 'call':
        code = frame.f_code

        print("%s: %s" % (code.co_filename, code.co_name))

        for name in frame.f_locals:
            val = frame.f_locals[name]
            print('{:<4}{:<7}: {}'.format(' ', name, val))

    return None
```

## Example: var_args_2.py

```python
import sys
from trace_call_args import trace_call_args_2

def var_args(*args, **kwargs):
    x = 1
    y = 2
    return x, y

if __name__ == '__main__':
    sys.settrace(trace_call_args_2)
    args = (1, 2, 'foo')
    kwargs = {'a': 8, 'b': 9}
    var_args(*args, **kwargs)
    sys.settrace(None)
```

## Output: var_args_2.py

```
$ python var_args_2.py

var_args_2.py: var_args
    args   : (1, 2, 'foo')
    kwargs : {'b': 9, 'a': 8}
```

## Agenda

## coverage.py

- Pytracer
- CTracer

## coverage.py tracer

```
class PyTracer(object):
    ...
    def _trace(self, frame, event, arg_unused):
        """
        The trace function passed to sys.settrace.
        """

        ...
        if event == 'call':
            ...
        elif event == 'line':
            ...
        elif event == 'return':
            ...
        return self._trace
```

## Python Debugger (Pdb)

- Built on top of Bdb
- "Debugger Framework"

## Bdb tracer

```
class Bdb:
    ...
    def trace_dispatch(self, frame, event, arg):
        ...
        if event == 'line':
            return self.dispatch_line(frame)
        if event == 'call':
            return self.dispatch_call(frame, arg)
        if event == 'return':
            return self.dispatch_return(frame, arg)
        if event == 'exception':
            return self.dispatch_exception(frame, arg)
        ...
```

## pythontutor.com

"PGLogger is a subclass of **bdb.Bdb**...

Here's where the magic happens...

As the user's program is running, **bdb** will pause execution at every function call, return, exception, and single-line step (most common)."

## Agenda

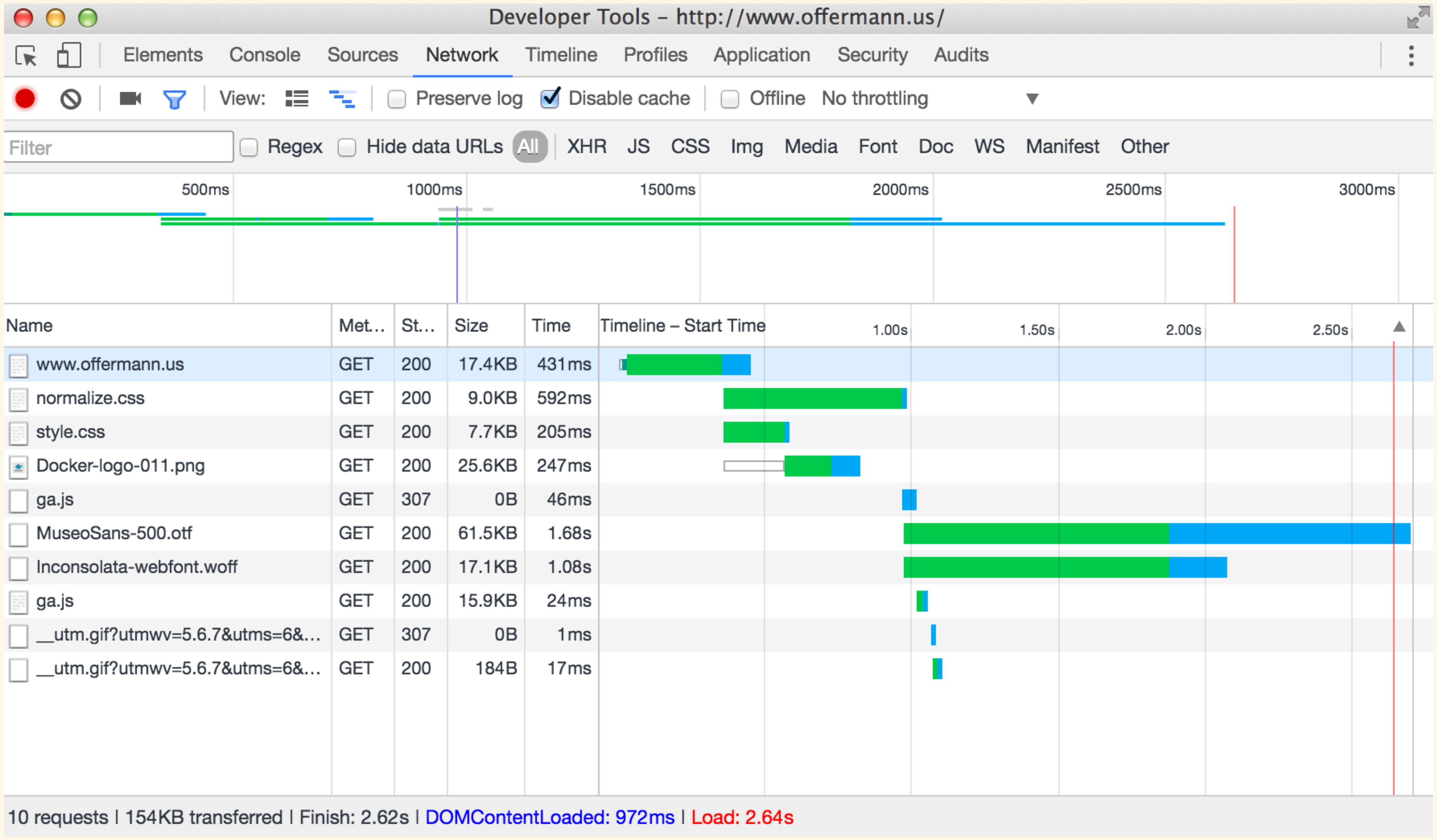
1. Simple Tracing
2. How Does Tracing Work?
3. Custom Tracing
4. Who Else Uses Custom Tracing?
5. **Full-Fledged Example**

## Let's build a Trace Visualizer...

## Let's build a Trace Visualizer...

**Or, borrow one!**

# Chrome Dev Tools

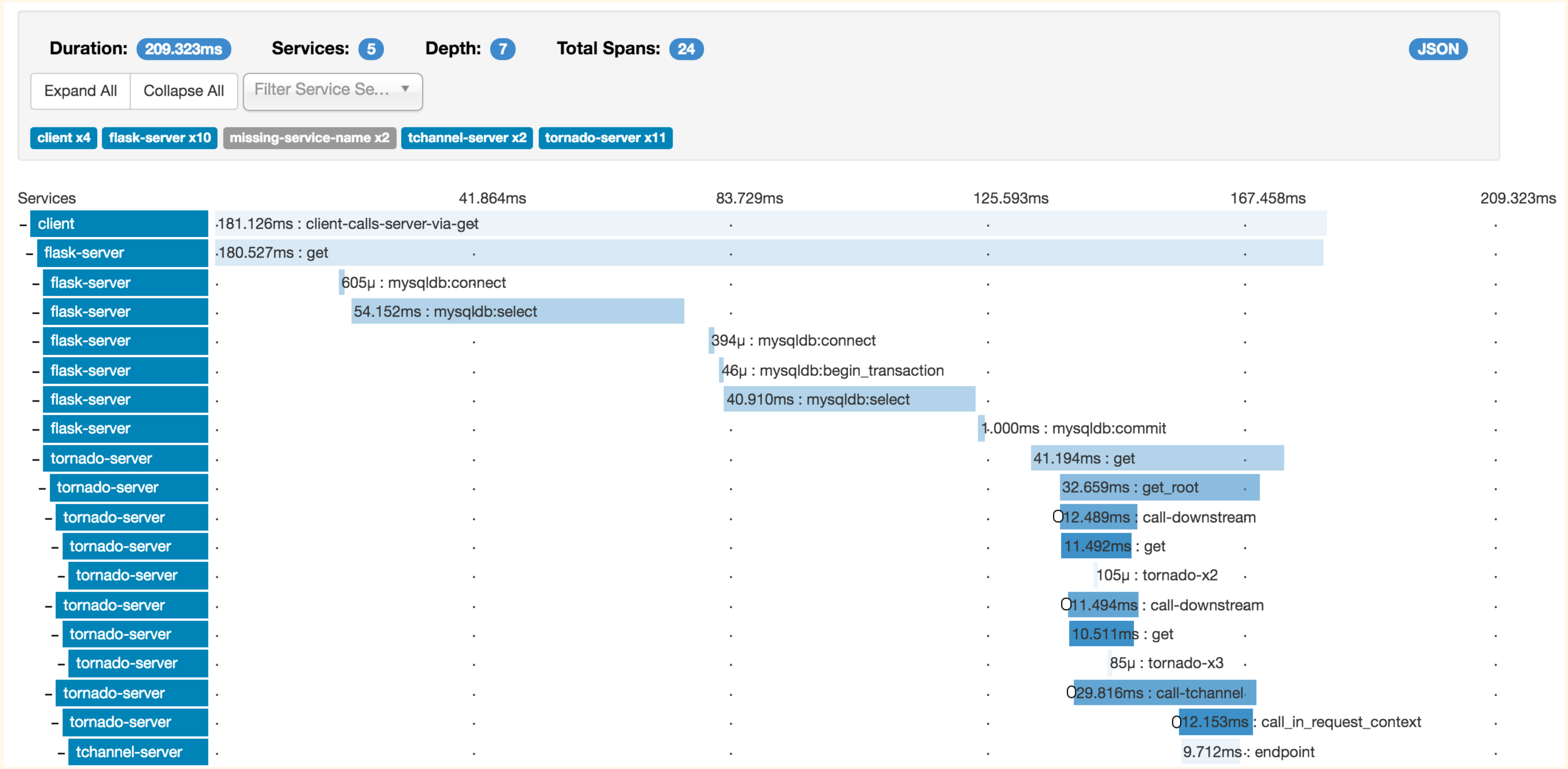


Developer Tools – http://www.offermann.us/
Elements Console Sources Network Timeline Profiles Application Security Audits
View: Preserve log Disable cache Offline No throttling
Filter Regex Hide data URLs All XHR JS CSS Img Media Font Doc WS Manifest Other
500ms 1000ms 1500ms 2000ms 2500ms 3000ms
Timeline – Start Time 1.00s 1.50s 2.00s 2.50s

| Name | Met… | St… | Size | Time |
|---|---|---|---|---|
| www.offermann.us | GET | 200 | 17.4KB | 431ms |
| normalize.css | GET | 200 | 9.0KB | 592ms |
| style.css | GET | 200 | 7.7KB | 205ms |
| Docker-logo-011.png | GET | 200 | 25.6KB | 247ms |
| ga.js | GET | 307 | 0B | 46ms |
| MuseoSans-500.otf | GET | 200 | 61.5KB | 1.68s |
| Inconsolata-webfont.woff | GET | 200 | 17.1KB | 1.08s |
| ga.js | GET | 200 | 15.9KB | 24ms |
| __utm.gif?utmwv=5.6.7&utms=6&… | GET | 307 | 0B | 1ms |
| __utm.gif?utmwv=5.6.7&utms=6&… | GET | 200 | 184B | 17ms |

10 requests | 154KB transferred | Finish: 2.62s | DOMContentLoaded: 972ms | Load: 2.64s

# Zipkin



**Duration:** 209.323ms **Services:** 5 **Depth:** 7 **Total Spans:** 24 JSON

Expand All | Collapse All | Filter Service Se...

client x4 | flask-server x10 | missing-service-name x2 | tchannel-server x2 | tornado-server x11

```
Services              41.864ms     83.729ms     125.593ms     167.458ms     209.323ms
– client              181.126ms : client-calls-server-via-get
 – flask-server       180.527ms : get
  – flask-server      605μ : mysqldb:connect
  – flask-server      54.152ms : mysqldb:select
  – flask-server      394μ : mysqldb:connect
  – flask-server      46μ : mysqldb:begin_transaction
  – flask-server      40.910ms : mysqldb:select
  – flask-server      1.000ms : mysqldb:commit
  – tornado-server    41.194ms : get
   – tornado-server   32.659ms : get_root
    – tornado-server  12.489ms : call-downstream
     – tornado-server 11.492ms : get
      – tornado-server 105μ : tornado-x2
    – tornado-server  11.494ms : call-downstream
     – tornado-server 10.511ms : get
      – tornado-server 85μ : tornado-x3
    – tornado-server  29.816ms : call-tchannel
     – tornado-server 12.153ms : call_in_request_context
      – tchannel-server 9.712ms : endpoint
```

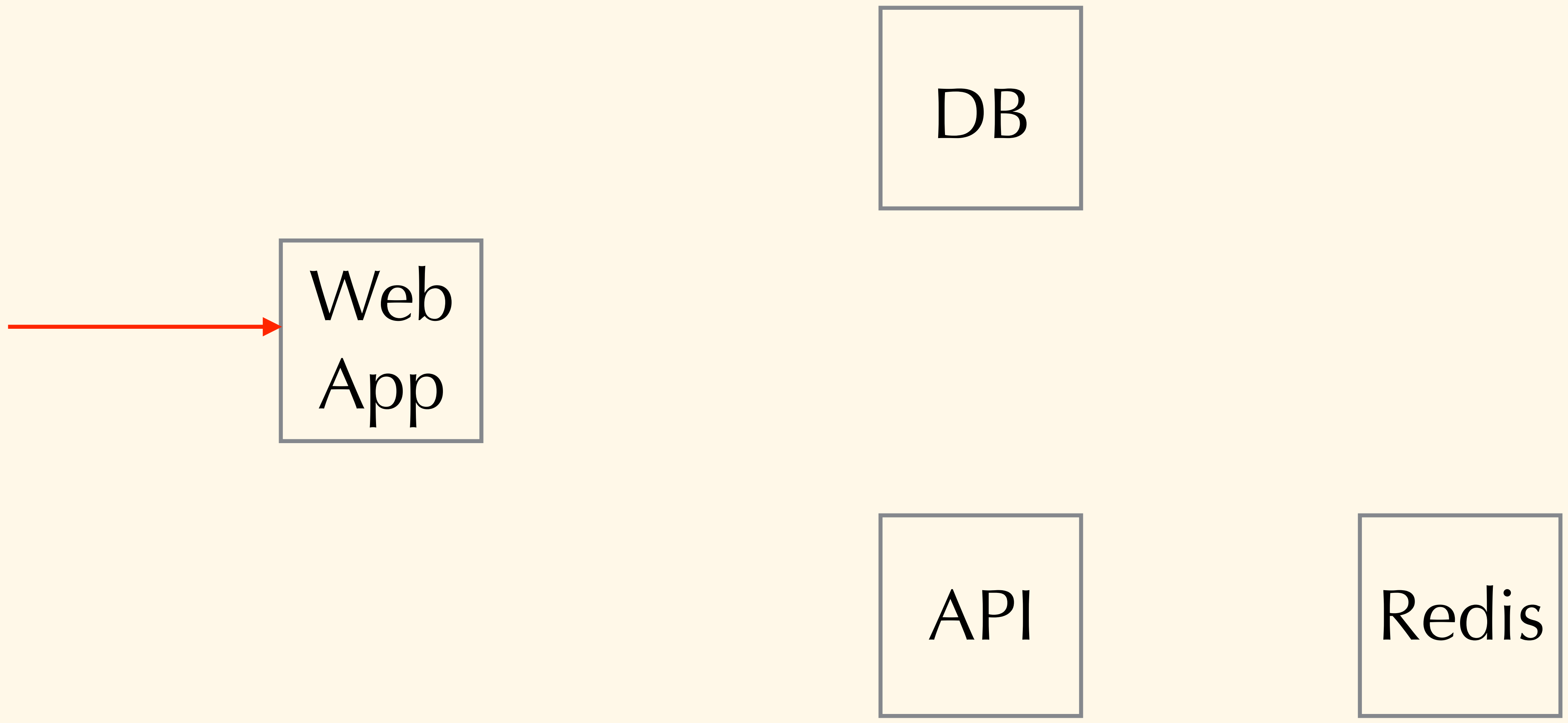
## What is Zipkin?

- Distributed Tracing for microservices
- Many RPC calls to serve single request.
    - HTTP calls to API endpoints
    - Database queries

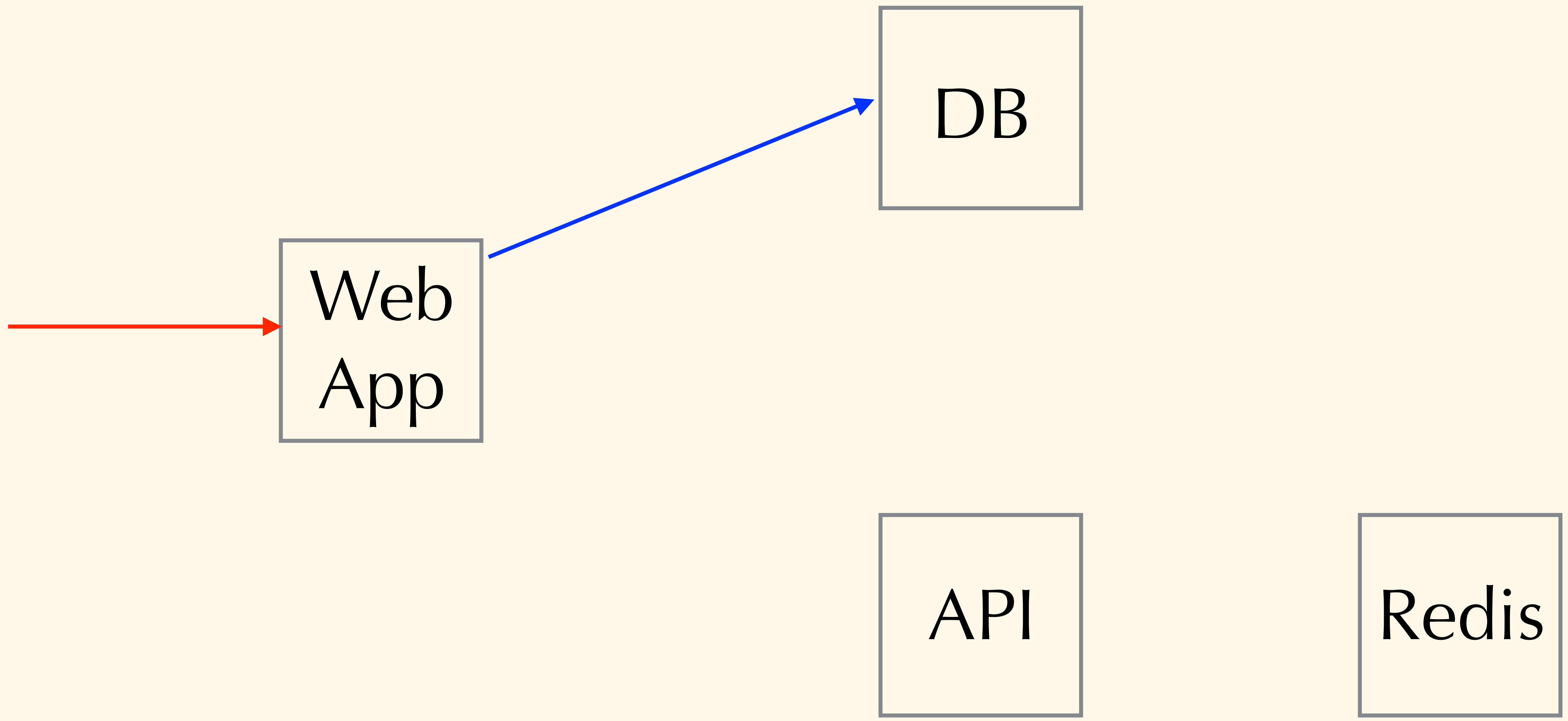
## What is Zipkin? (2)

- **Spans**: Individual RPC calls.
- **Trace**: Collection of Spans.

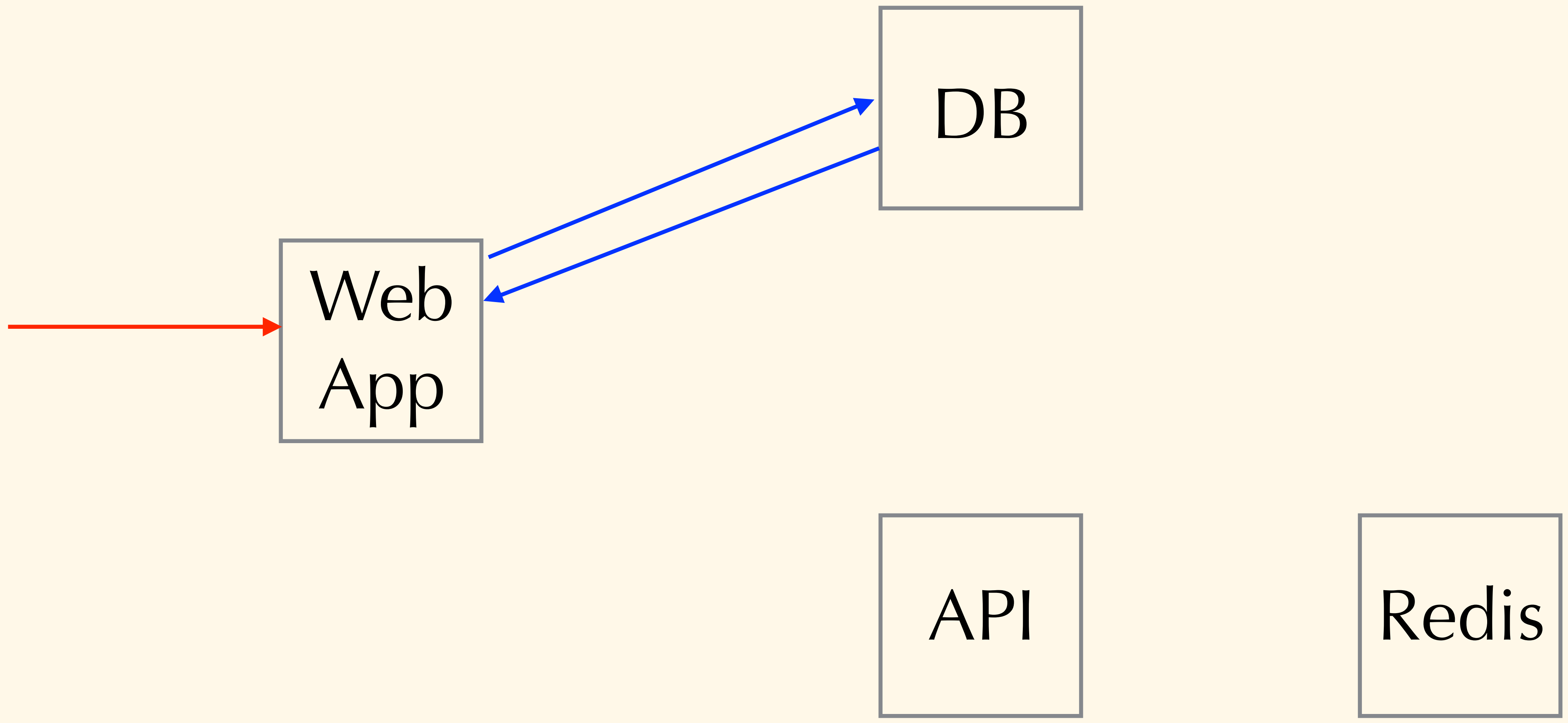
# Zipkin Example



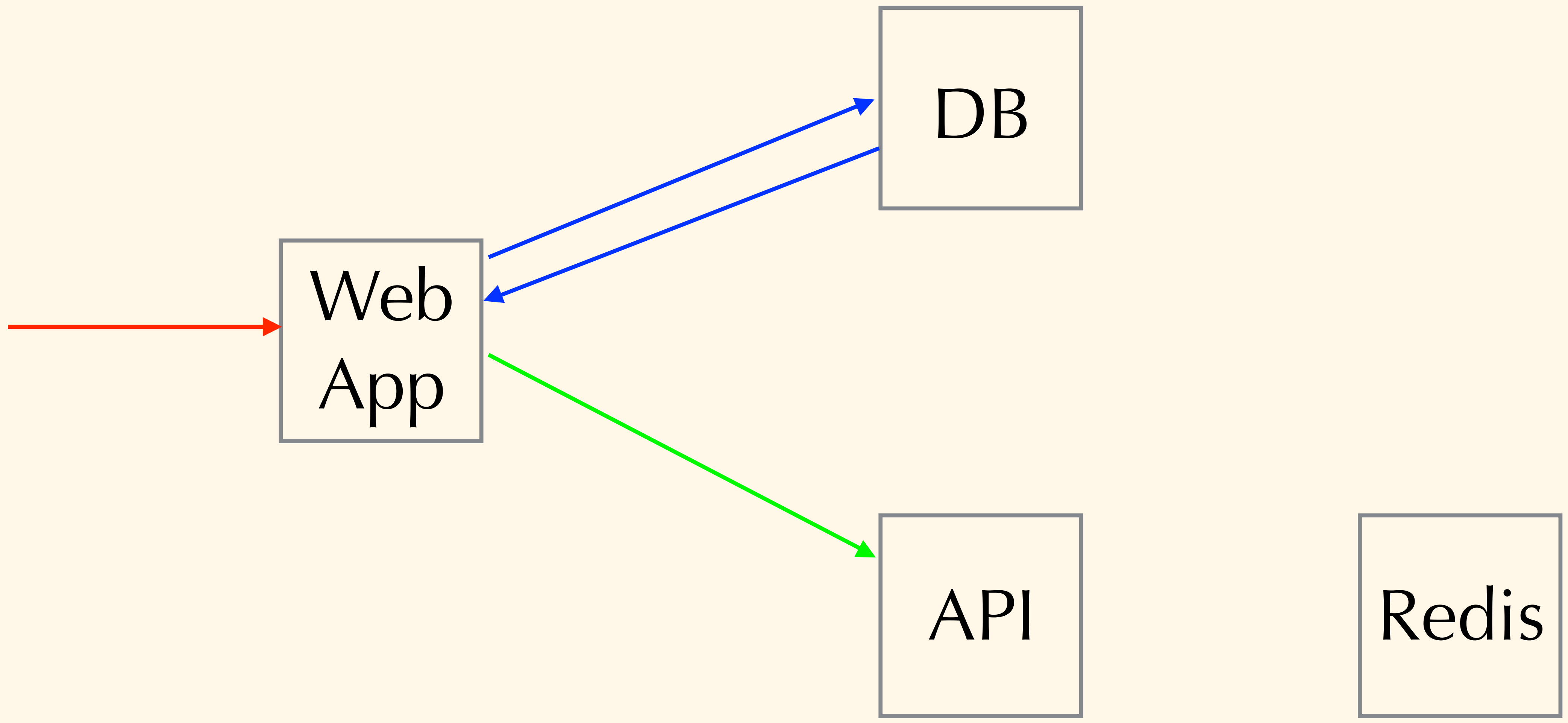
## Zipkin Example



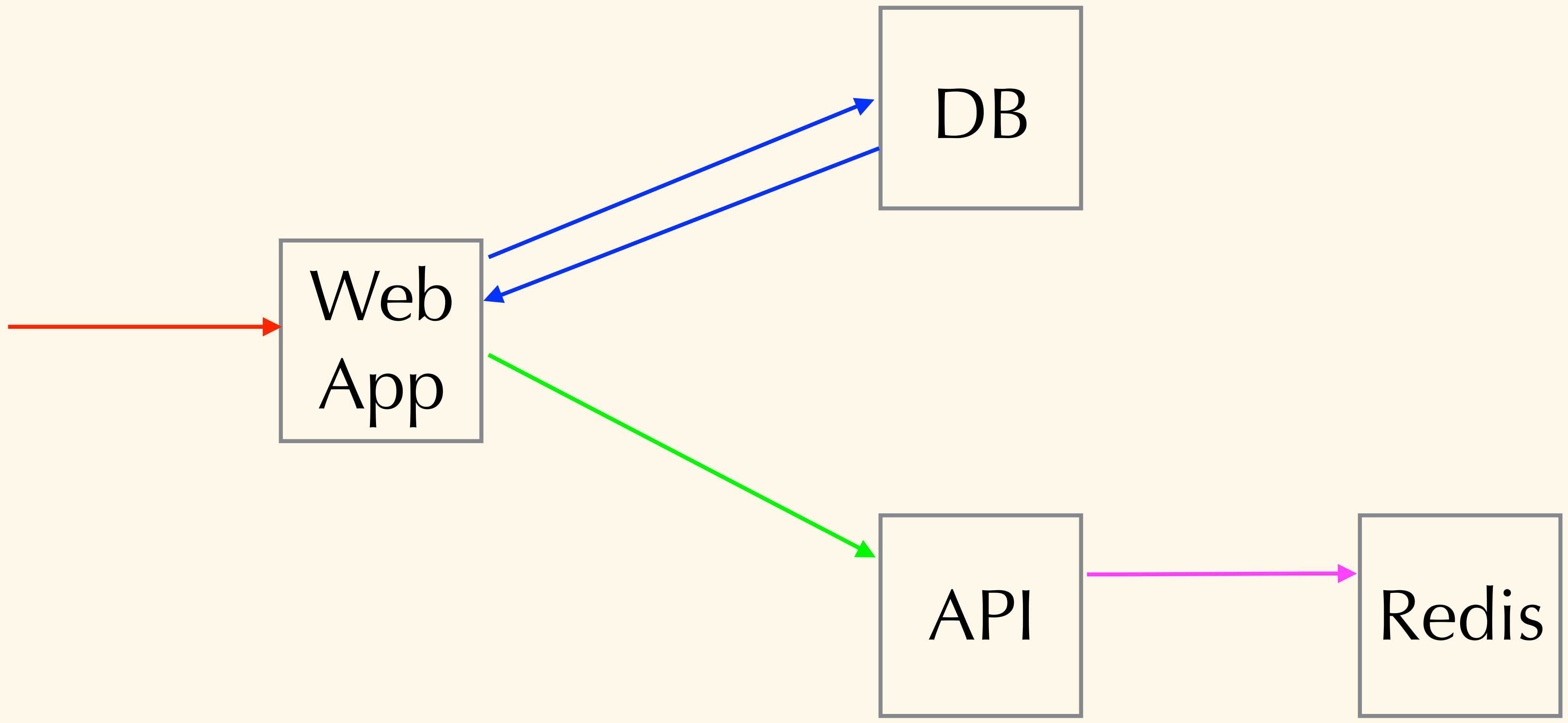
## Zipkin Example



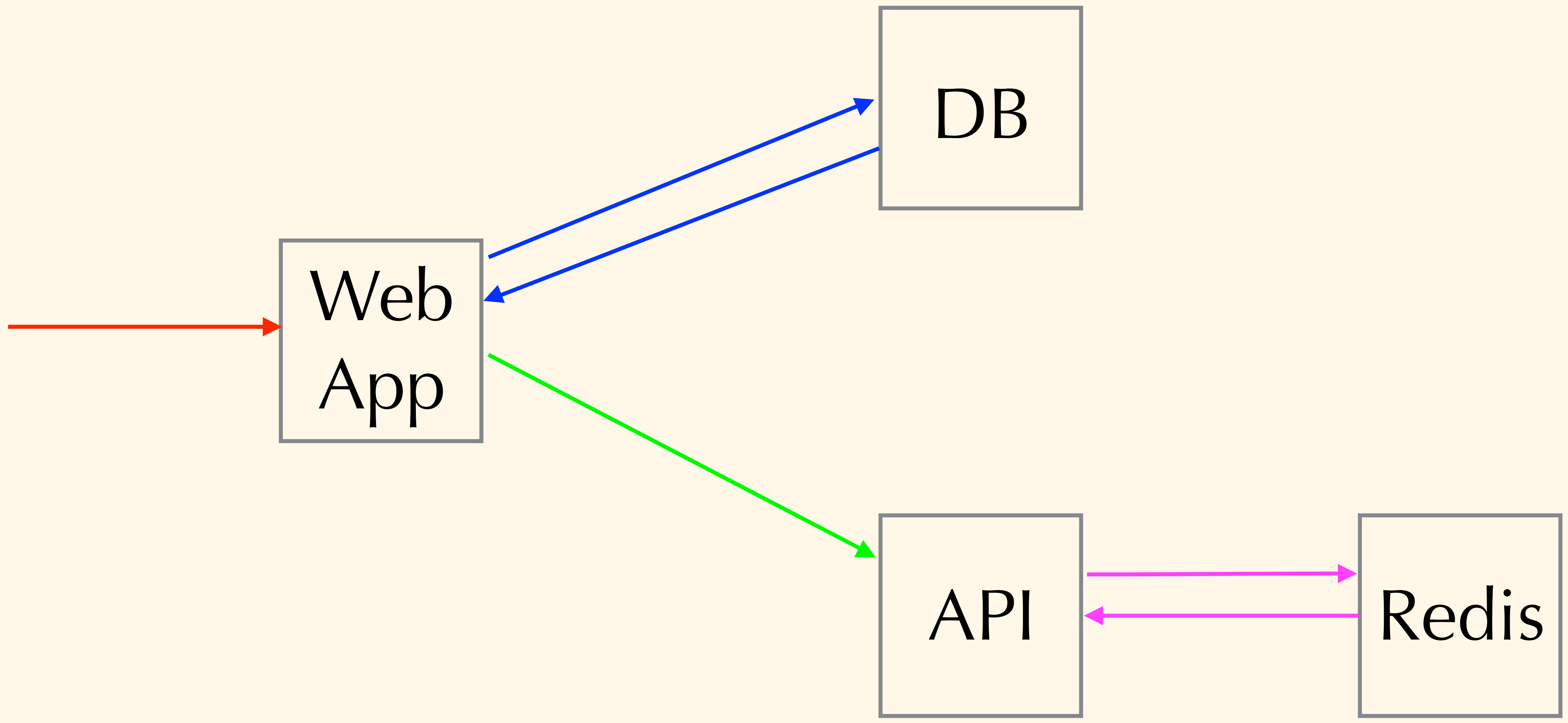
## Zipkin Example



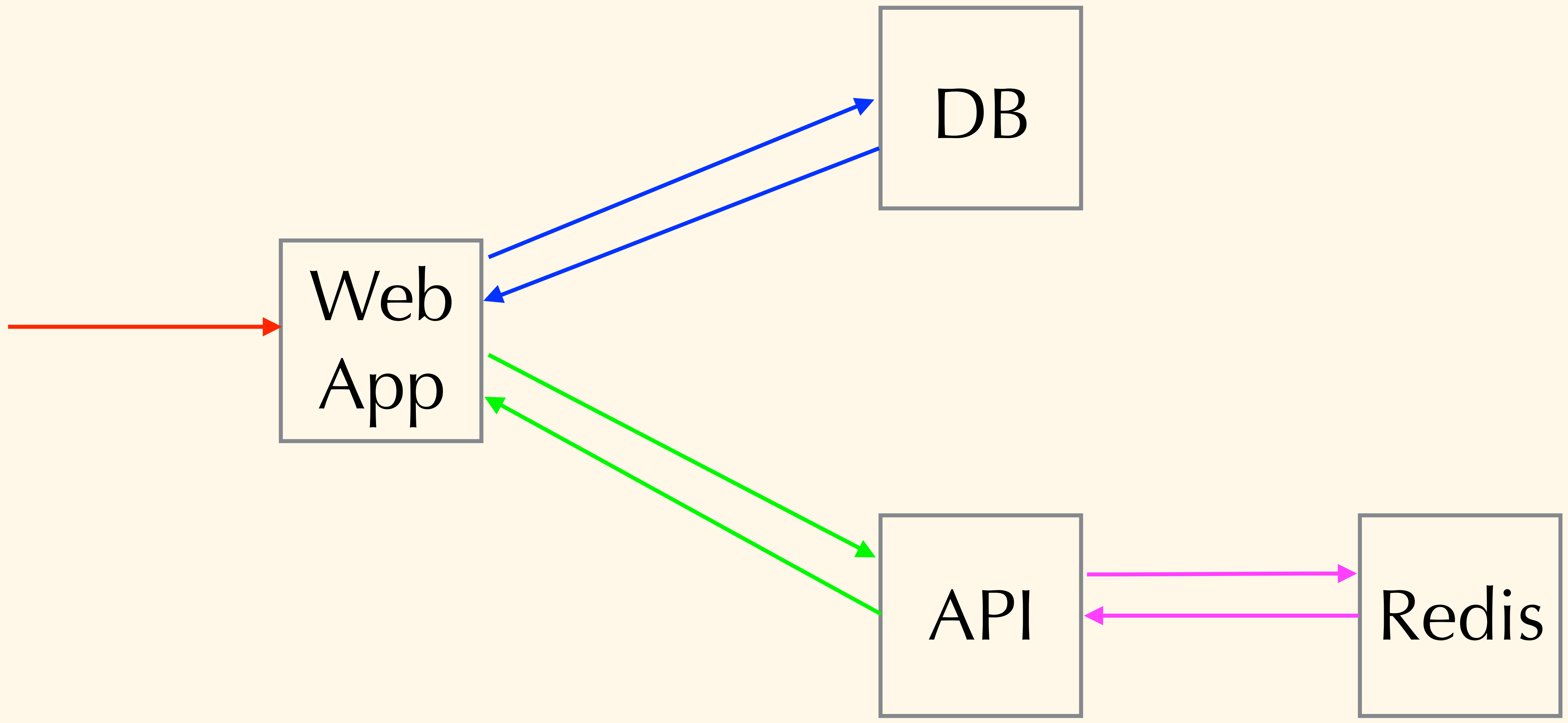
# Zipkin Example



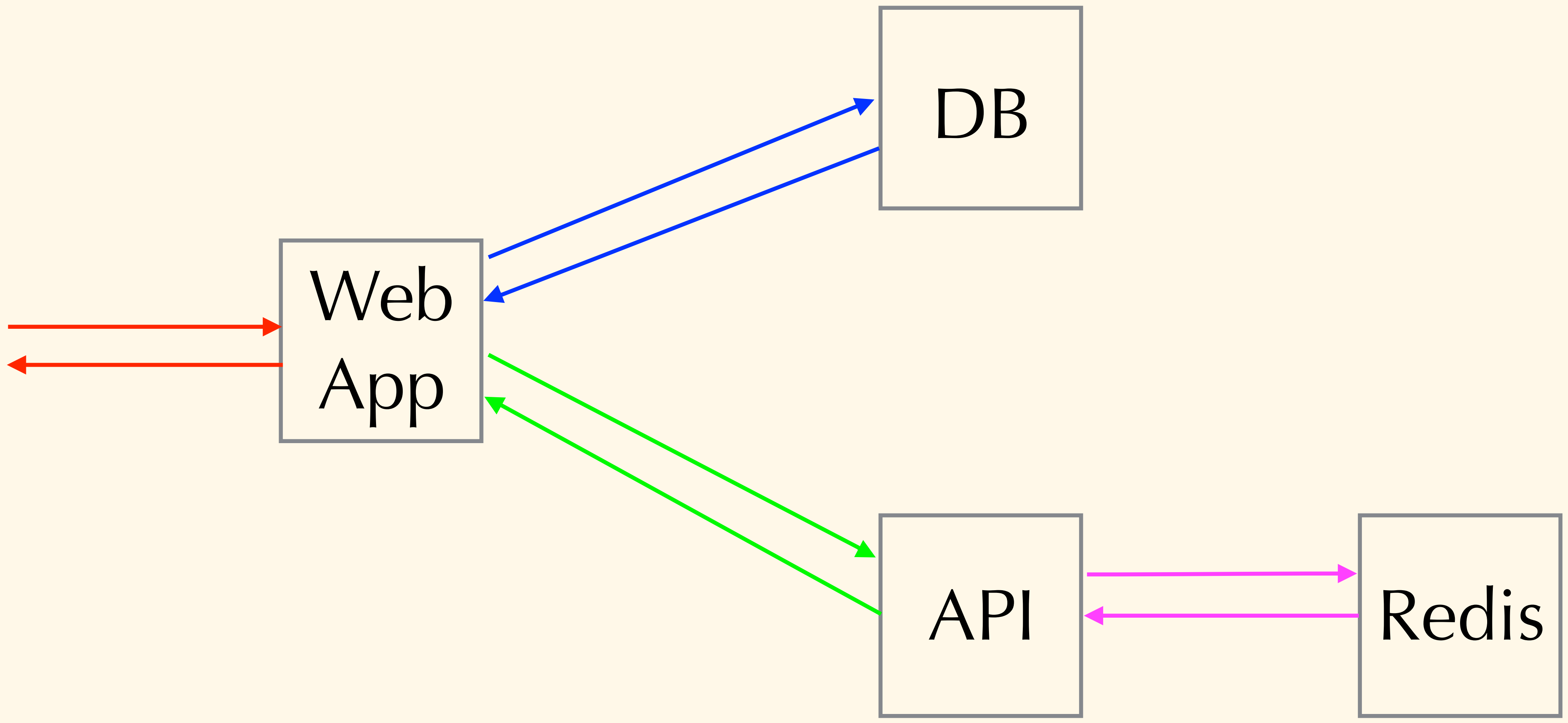
# Zipkin Example



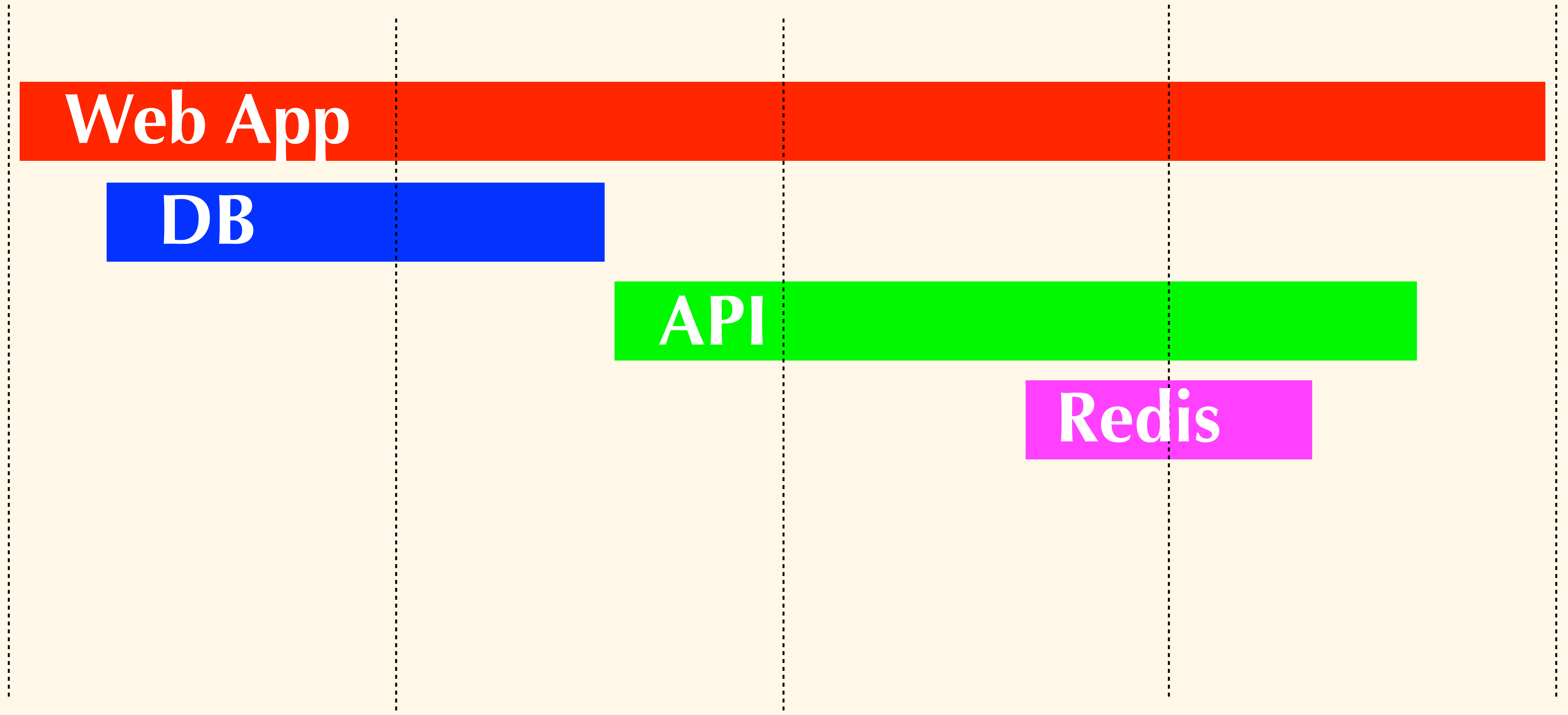
## Zipkin Example



# Zipkin Example



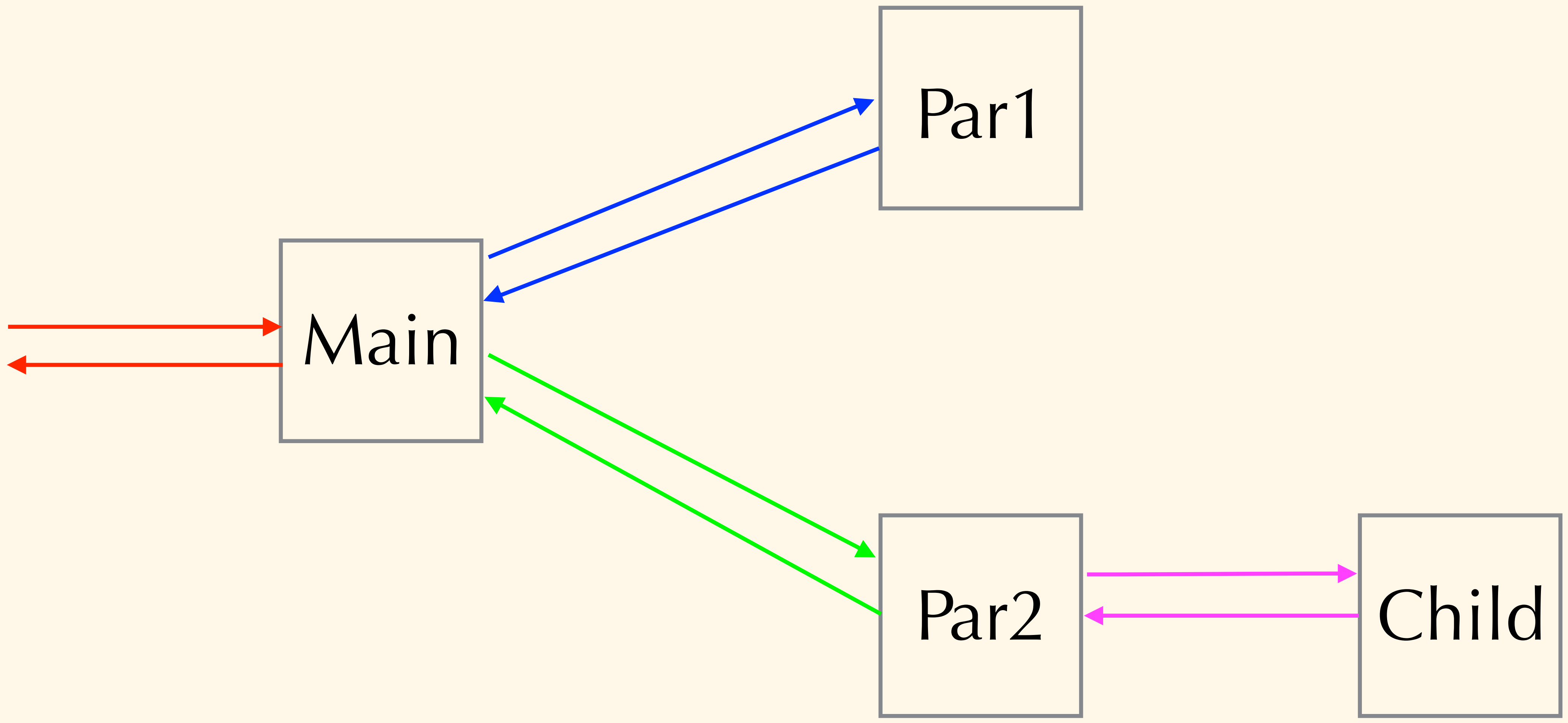
# Zipkin Diagram



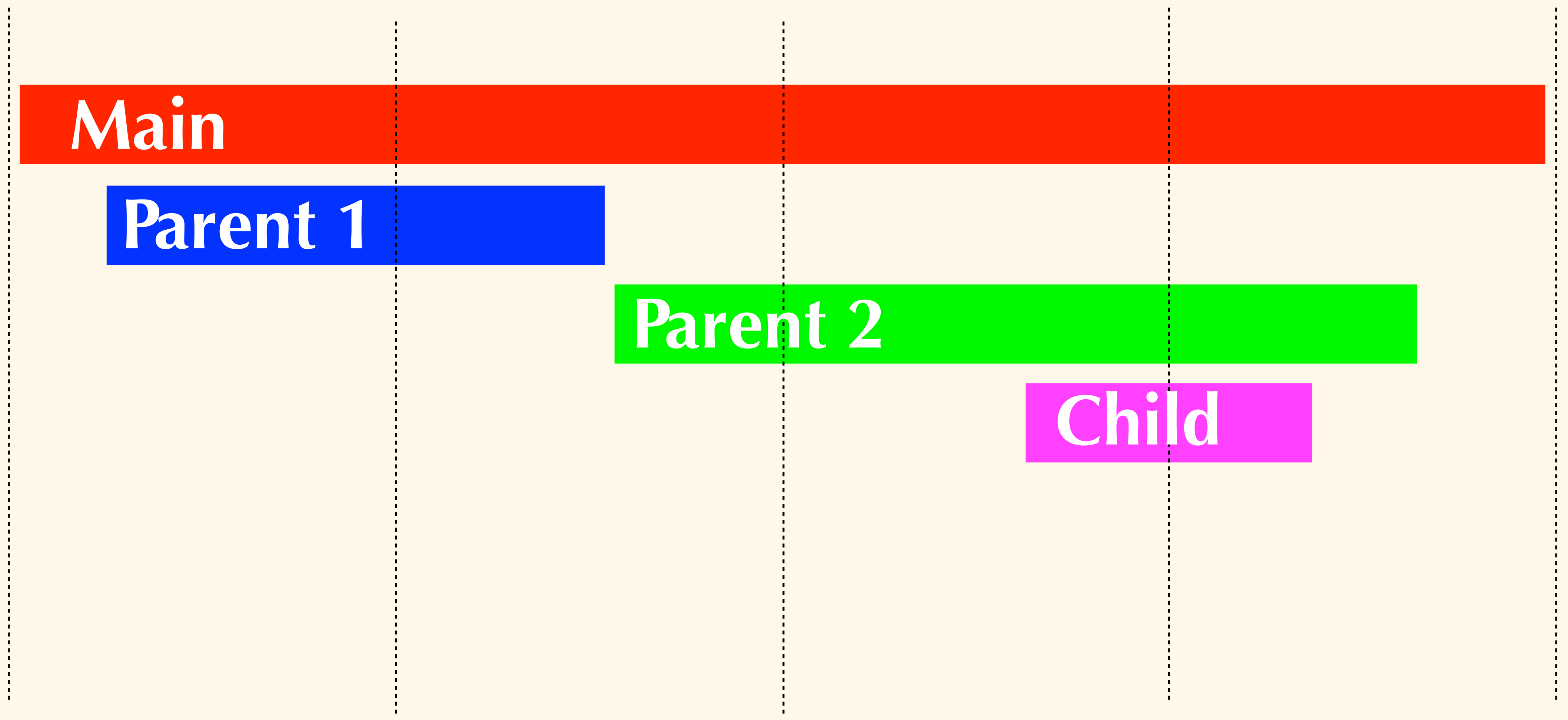
## Translation to Tracing

- Trace -> Run Program
- Span -> Function call (frame)

# Translation to Tracing



## Spans = Frames



## Zipkin Span Format

```
[{
   "parentId": "0ed0c2af926048a7",
   "name": "child",
   "timestamp": 1475272253175560.0,
   "debug": true,
   "id": "0e8a6ecd28d0b1bd",
   "annotations": [
     {
       "endpoint": {
         "ipv4": "127.0.0.1",
         "port": 8888,
         "serviceName": "simple_service"
       },
       "timestamp": 1475272253175560.0,
       "value": "sr"
     }],
   "traceId": "22bf9db1413e9fcf"
 }]
```

## Zipkin Span Format

```
[{
   "parentId": "0ed0c2af926048a7",
   "name": "child",
   "timestamp": 1475272253175560.0,
   "debug": true,
   "id": "0e8a6ecd28d0b1bd",
   "annotations": [
     {
       "endpoint": {
         "ipv4": "127.0.0.1",
         "port": 8888,
         "serviceName": "simple_service"
       },
       "timestamp": 1475272253175560.0,
       "value": "sr"
     }],
   "traceId": "22bf9db1413e9fcf"
 }]
```

## Zipkin Span Format

```
[{
   "parentId": "0ed0c2af926048a7",
   "name": "child",
   "timestamp": 1475272253175560.0,
   "debug": true,
   "id": "0e8a6ecd28d0b1bd",
   "annotations": [
     {
       "endpoint": {
         "ipv4": "127.0.0.1",
         "port": 8888,
         "serviceName": "simple_service"
       },
       "timestamp": 1475272253175560.0,
       "value": "sr"
     }],
   "traceId": "22bf9db1413e9fcf"
 }]
```

## Zipkin Span Format

```
[{
   "parentId": "0ed0c2af926048a7",
   "name": "child",
   "timestamp": 1475272253175560.0,
   "debug": true,
   "id": "0e8a6ecd28d0b1bd",
   "annotations": [
     {
       "endpoint": {
         "ipv4": "127.0.0.1",
         "port": 8888,
         "serviceName": "simple_service"
       },
       "timestamp": 1475272253175560.0,
       "value": "sr"
     }],
   "traceId": "22bf9db1413e9fcf"
 }]
```

## Zipkin Span Format

```
[{
   "parentId": "0ed0c2af926048a7",
   "name": "child",
   "timestamp": 1475272253175560.0,
   "debug": true,
   "id": "0e8a6ecd28d0b1bd",
   "annotations": [
     {
       "endpoint": {
         "ipv4": "127.0.0.1",
         "port": 8888,
         "serviceName": "simple_service"
       },
       "timestamp": 1475272253175560.0,
       "value": "sr"
     }],
   "traceId": "22bf9db1413e9fcf"
 }]
```

## Plan

1. Trace program
2. Log results
3. Convert log to Zipkin JSON
4. Visualize!

## Tracer

```
class ZipkinTracer(object):

    def __init__(self):
        self.span_ids = {}  # {frame: span_id}

    def __call__(self, frame, event, arg):
        if event == 'call':
            return self.trace_call(frame, event, arg)
        elif event == 'return':
            return self.trace_return(frame,event, arg)
        else:
            return self
```

## Tracer for Call Event

```python
def trace_call(self, frame, event, arg):
    current_id = os.urandom(8).hex()
    self.span_ids[frame] = current_id

    parent_id = self.parent_span_id(frame)
    func_name = frame.f_code.co_name
    zipkin_logger.debug('%s,%s,%s,%f,%s' % (
            current_id, parent_id, func_name,
            time.time(), 'sr'))

    return self
```

## Tracer for Return Event

```python
def trace_return(self, frame, event, arg):
    current_id = self.span_ids[frame]
    parent_id = self.parent_span_id(frame)
    func_name = frame.f_code.co_name
    zipkin_logger.debug('%s,%s,%s,%f,%s' % (
            current_id, parent_id, func_name,
            time.time(), 'ss'))

    del self.span_ids[frame]
    return self
```

## Tracer

```python
def parent_span_id(self, frame):
    parent = frame.f_back
    return self.span_ids.get(parent, '')
```

## Log File

```
3f5edd6b77ed0bf5,,main,1475053714.172785,sr
b6c53ba3caa2f504,3f5edd6b77ed0bf5,parent,1475053714.172898,sr
61b79578737e7423,b6c53ba3caa2f504,child,1475053714.172950,sr
61b79578737e7423,b6c53ba3caa2f504,child,1475053714.173023,ss
09abc624393d70d5,b6c53ba3caa2f504,child,1475053714.173066,sr
09abc624393d70d5,b6c53ba3caa2f504,child,1475053714.173134,ss
5d9be223d8d7bd4c,b6c53ba3caa2f504,child,1475053714.173184,sr
5d9be223d8d7bd4c,b6c53ba3caa2f504,child,1475053714.173259,ss
b6c53ba3caa2f504,3f5edd6b77ed0bf5,parent,1475053714.173304,ss
3f5edd6b77ed0bf5,,main,1475053714.173345,ss
```

## log2span.py

```
for line in read_log(filename):
    span = make_span(*line)
    send_span(span)
```

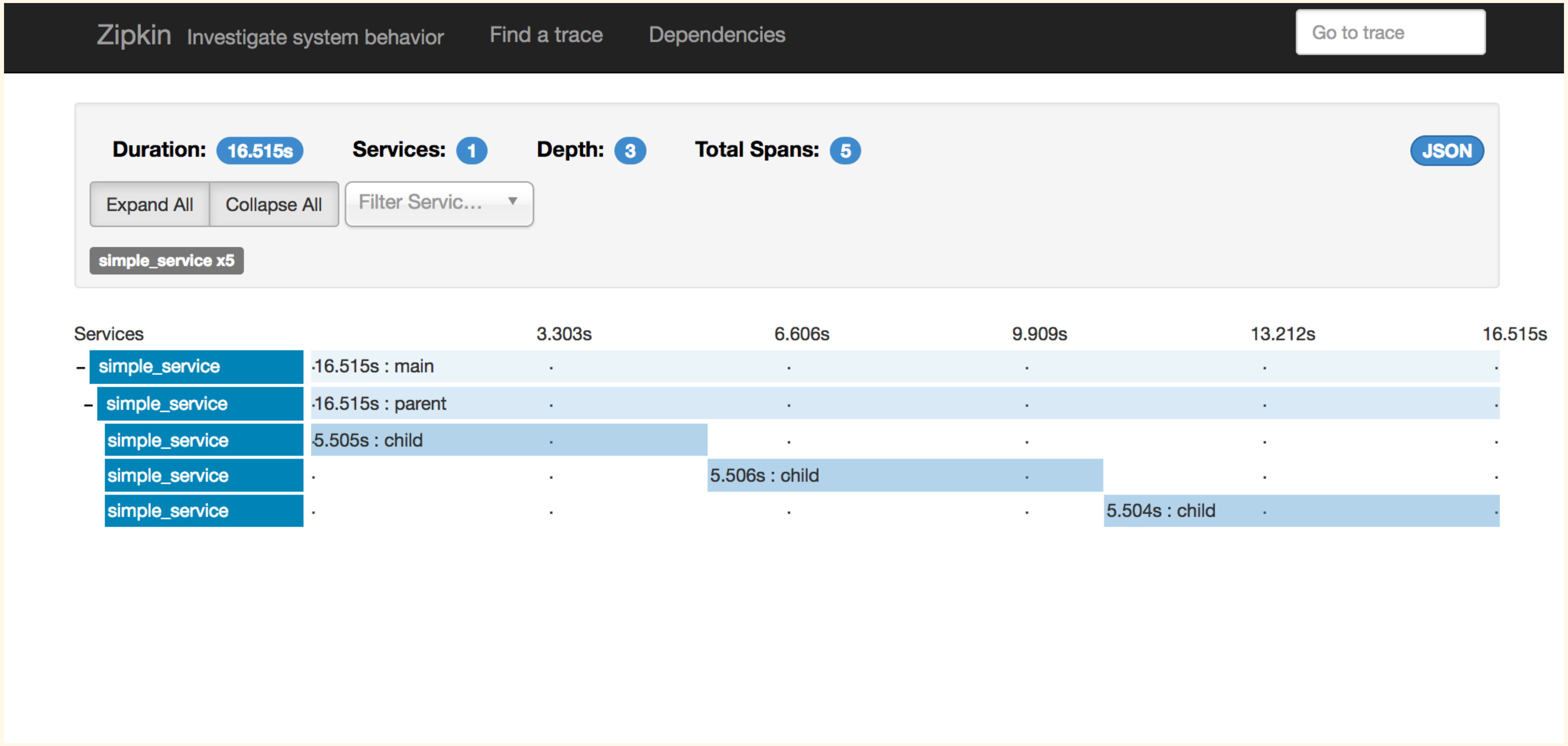
## make_span

```python
def make_span(span_id, parent_id, func_name, timestamp,
      annotation_value, endpoint=DEFAULT_ENDPOINT, debug=True):

  span = Span(
          span_id=span_id,
          parent_id=parent_id,
          trace_id=DEFAULT_TRACE_ID,
          name=func_name,
          timestamp= timestamp * (10 ** 6),
          debug=debug
   )

   annotation = Annotation(
          endpoint=DEFAULT_ENDPOINT,
          timestamp= timestamp * (10 ** 6),
          value = annotation_value
   )

   span.add_annotation(annotation)
   return span
```

## send_span

```
def send_span(span):
    url = 'http://192.168.99.100:9411/api/v1/spans'
    payload = [span.asdict()]
    resp = requests.post(url, json=payload)
    print(resp)
```

## End Result



Duration: 16.515s Services: 1 Depth: 3 Total Spans: 5

simple_service x5

| Services | 3.303s | 6.606s | 9.909s | 13.212s | 16.515s |
|---|---|---|---|---|---|
| – simple_service | 16.515s : main | | | | |
| – simple_service | 16.515s : parent | | | | |
| simple_service | 5.505s : child | | | | |
| simple_service | | 5.506s : child | | | |
| simple_service | | | | 5.504s : child | |

## Conclusion

- Nothing scary about trace functions.
- Tracing can be extraordinarily verbose.
- Capable of deep introspection.

## Conclusion

- Nothing scary about trace functions.
- Tracing can be extraordinarily verbose.
- Capable of deep introspection.
- **Repurposing UI is OK!**

## Further Reading

- **github.com/toffer/talk-custom-tracing**
  - Code Examples
  - Slide Deck

## Thanks!