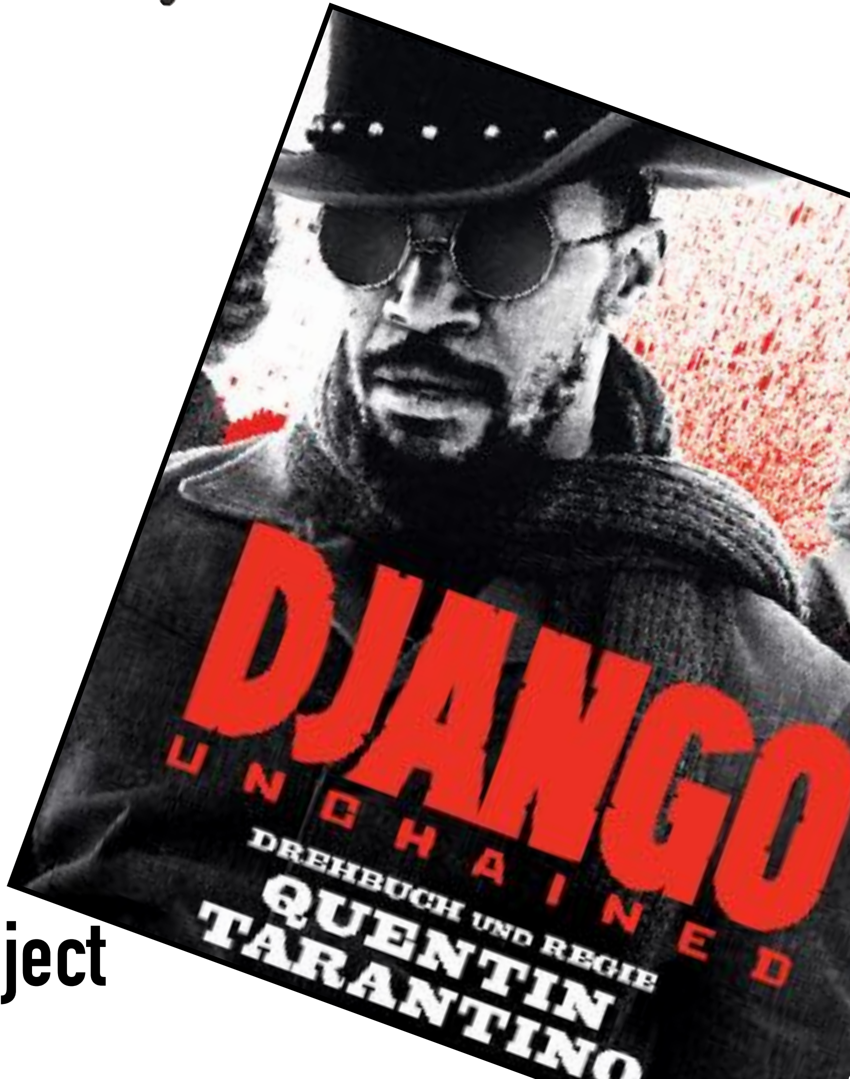
Lightweight Web App Framework for C++

Cjango Unchained

Mengdi Lin

Yasutaka Tanaka

Yi Qi



2017 @ Columbia University CS4995 Project

Abstract

Cjango is a lightweight C++ web application framework. Its coolest feature is dynamic callback reloading: the user can make changes to his server and re-deploy the changes without service interruption, very much similar to Python's Django. Cjango is compatible with HTTP 1.0 protocol. Under the hood, it handles HTTP requests asynchronously, and records >2x performance compared to Django in static web page serving. It also achieves comparable speed with other common C++ web frameworks. High performance combined with a user friendly API, Cjango is the ideal library for anyone who wants to create fast servers without spending too much time learning to use a web framework.

Time: 35 minutes + 10 minutes Q&A

What is Web Application Framework (Framework)?

Web App Framework

- common web frameworks provide 2 basic functionalities
  - routing: routes a http request to the correct user-defined callback function
  - wrapper around low level socket handling that hides the details of packing and unpacking http requests and responses from the user
- an abstraction (framework) for faster&easier web development
- runs an application by `app.run(80)` instead of writing low-level socket handling
- URL mappings to **callback functions** by predefined precedences
  - e.g. if URL is like “/diary/[0-9]{4}/[0-9]{1,2}”, then call `render_diary_page()`
- accesses user’s HTTP Request contents through Cjango::`HttpRequest` class
- automate database schema changes
- etc, etc.

Common Web App Frameworks

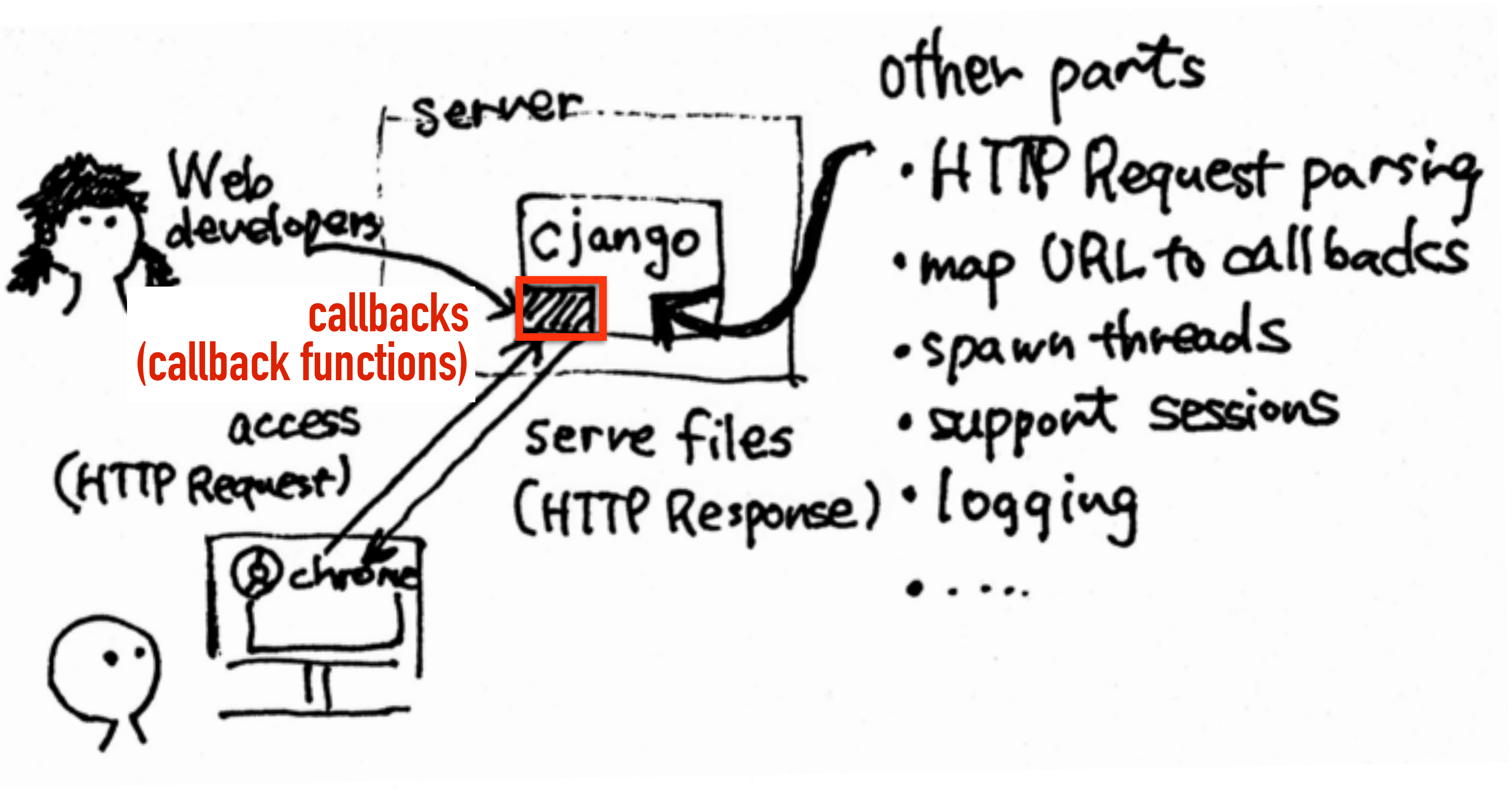
- exist in various languages
- In Python: (20K+ Github stars), (Python, 20K+), …
- In C++: (a **Flask** clone, 3K+ stars), Silicon (1K+ stars)

What is Cjango-Unchained?

Cjango-Unchained (Cjango) is a lightweight C++ web app framework

- provides a Django-like abstraction (framework) on top of C++
- features
  - full compatibility for HTTP 1.0 (GET/POST)
  - **HTTP Session support**
- high-speed
  - response speed: >3x faster than Django (Python) by **async request handling**
  - development speed: boosted by **dynamic callback loading**
    - new callback functions can be loaded at runtime
    - cks can be compiled separately
- user-friendly
  - (Github Star #1) doesn't have a tutorial :(
  - easy-to-use loggers (log-level, category)
  - auto-set compile commands

High-Level Sketch (Abstraction) of Cjango's Benefits



callbacks
(callback functions)

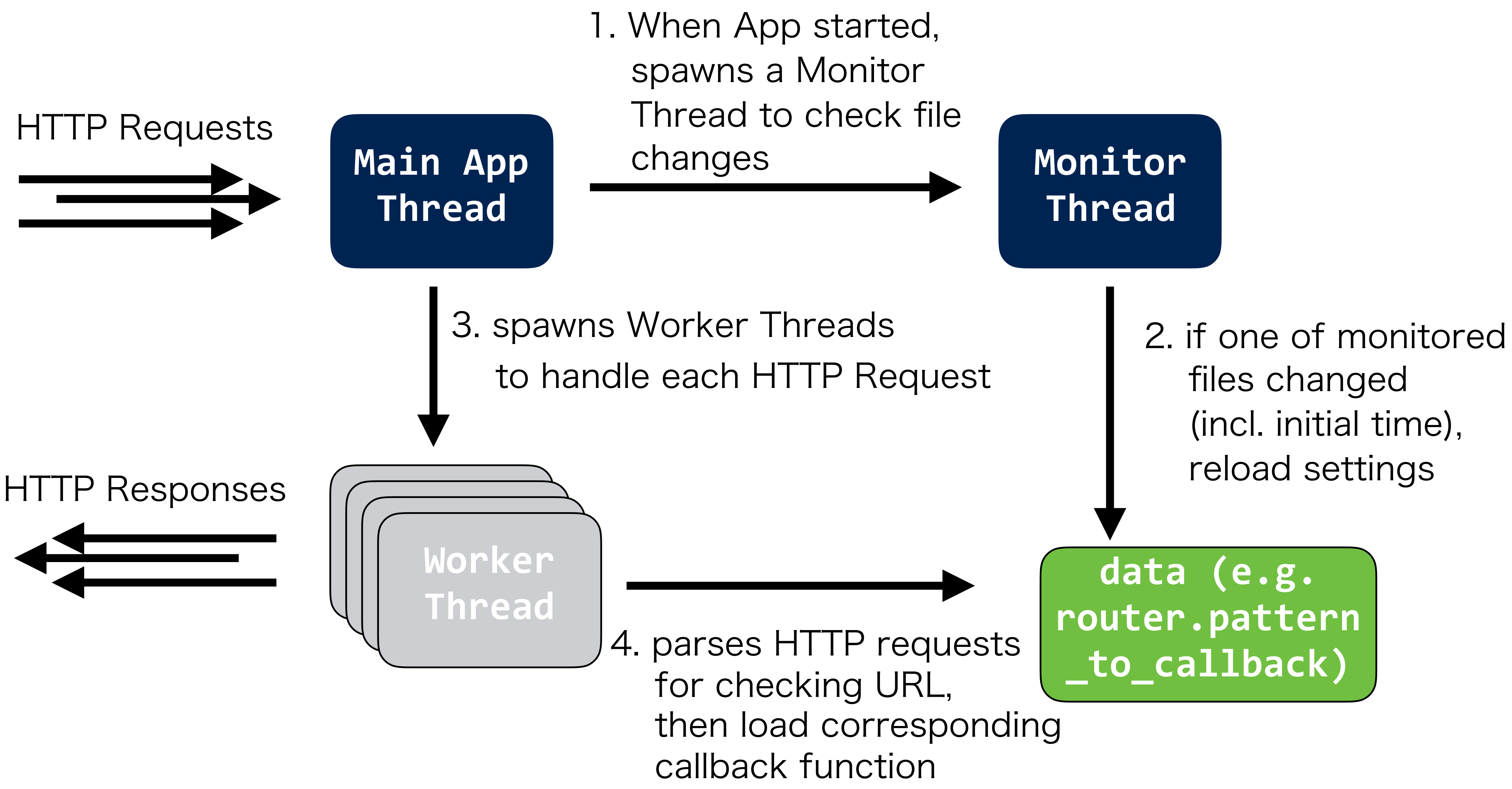
Cjango's Design Philosophy

“ Unchained “

:= Everything should be achieved without frustrating restrictions

- X should be fast
  - prarsing requests
  - HTTP responses
  - function unit tests
  - compilation time (c.f. #1 Crow)
  - dependency installation
    - no “autoconf && ./configure && make && …”
- X can be customized by user needs
  - URL patterns
  - directory hierarchy

High-Level Sketch of How Cjango works



HTTP Requests

Main App Thread

1. When App started, spawns a Monitor Thread to check file changes

Monitor Thread

3. spawns Worker Threads to handle each HTTP Request

2. if one of monitored files changed (incl. initial time), reload settings

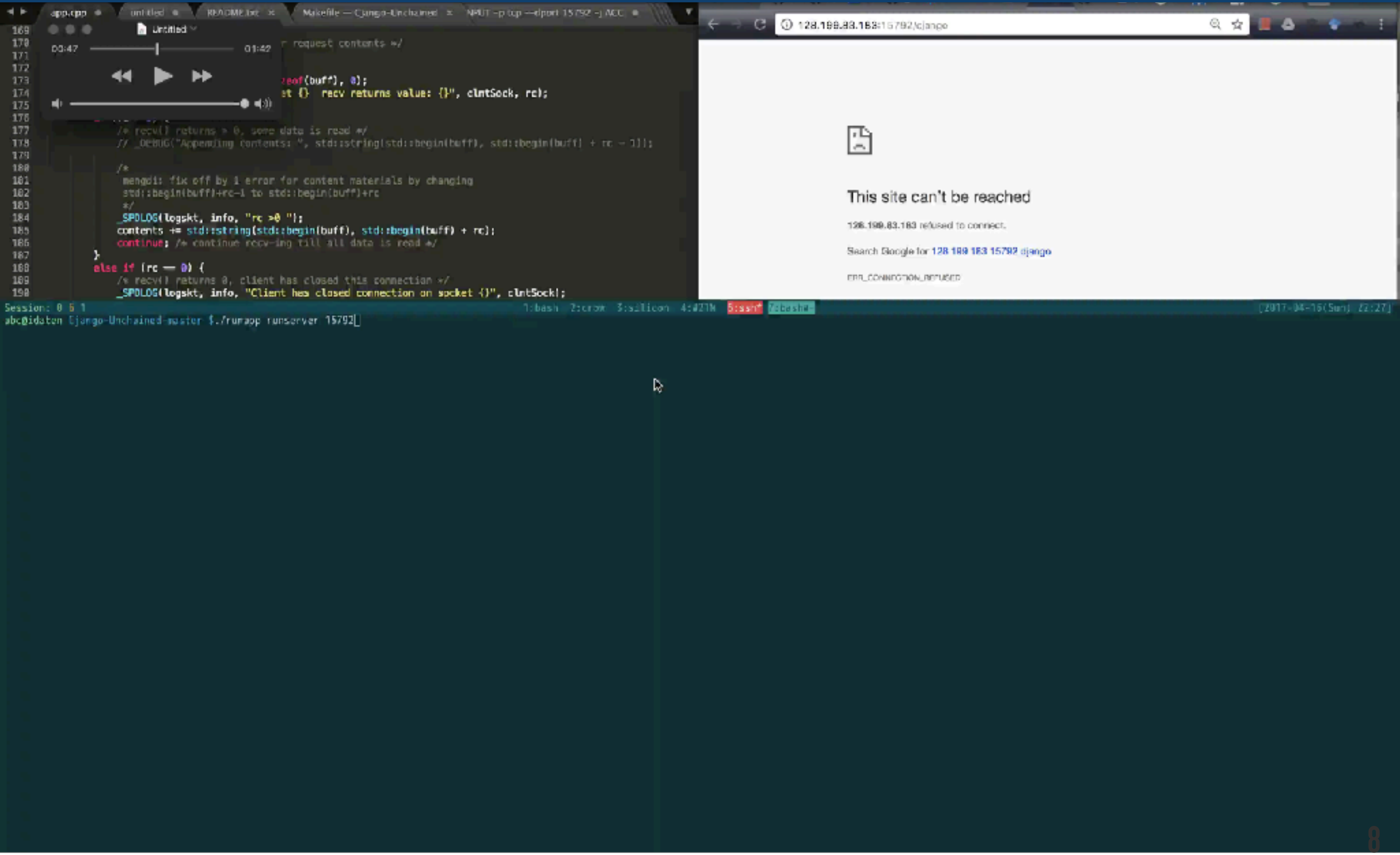
HTTP Responses

Worker Thread

data (e.g. router.pattern_to_callback)

4. parses HTTP requests for checking URL, then load corresponding callback function

Demo: A Static HTML file Served in Debug Mode (30secs)
(each of html, png, gif, and favicon.ico generates HttpRequests for server)



app.cpp
untitled
README.txt
Makefile — Django-Unchained
INPUT -p tcp --dport 15792 -j ACC
Untitled
00:47
01:42

```
r request contents */

zeof(buff), 0);
et {}  recv returns value: {}", clntSock, rc);

        /* recv() returns > 0, some data is read */
        // _DEBUG("Appending contents: ", std::string(std::begin(buff), std::begin(buff) + rc - 1));

        /*
         mengdi: fix off by 1 error for content materials by changing
         std::begin(buff)+rc-1 to std::begin(buff)+rc
         */
        _SPDLOG(logskt, info, "rc >0 ");
        contents += std::string(std::begin(buff), std::begin(buff) + rc);
        continue; /* continue recv-ing till all data is read */
    }
    else if (rc == 0) {
        /* recv() returns 0, client has closed this connection */
        _SPDLOG(logskt, info, "Client has closed connection on socket {}", clntSock);
```

This site can't be reached

128.199.83.183 refused to connect.

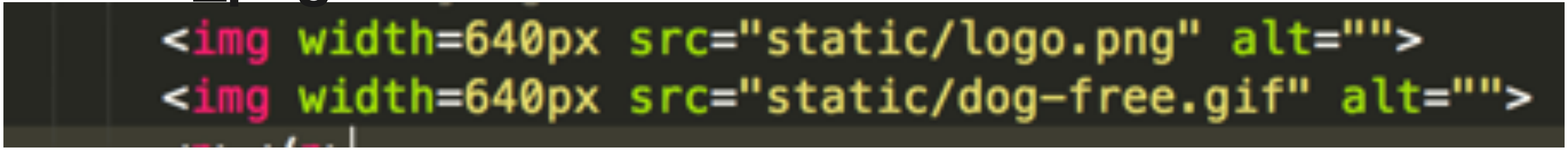
Search Google for 128 199 183 15792 django

ERR_CONNECTION_REFUSED

```
Session: 0 5 1                      1:bash  2:cron  3:silicon  4:#27K  5:ssh*  7:bash#-        [2017-04-16(Sun) 22:27]
abc@idaten Django-Unchained-master $./runapp runserver 15792
```

Static file routings are auto-generated

official_page.html



```
<img width=640px src="static/logo.png" alt="">
<img width=640px src="static/dog-free.gif" alt="">
```

Files under static/ can be accessed without URL mappings

- Cjango generates rules automatically
- users can customize the static file directory path

API Comparisons: Running App

```
python manage.py runserver 8000
```

or

```
./runapp runserver 8000
```

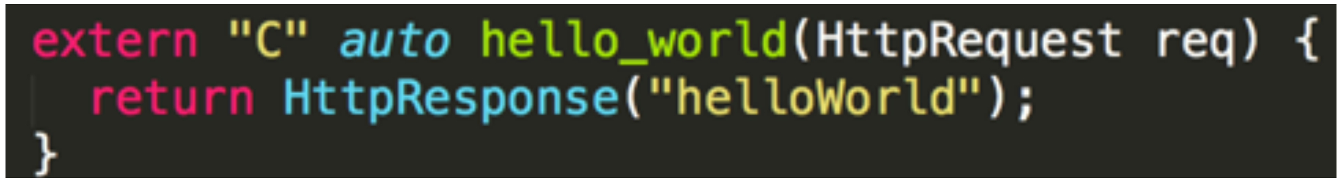
- manage.py is a wrapper script just to call ./runapp inside
  - compatible with django's init command

django

```
python manage.py runserver 8000
```

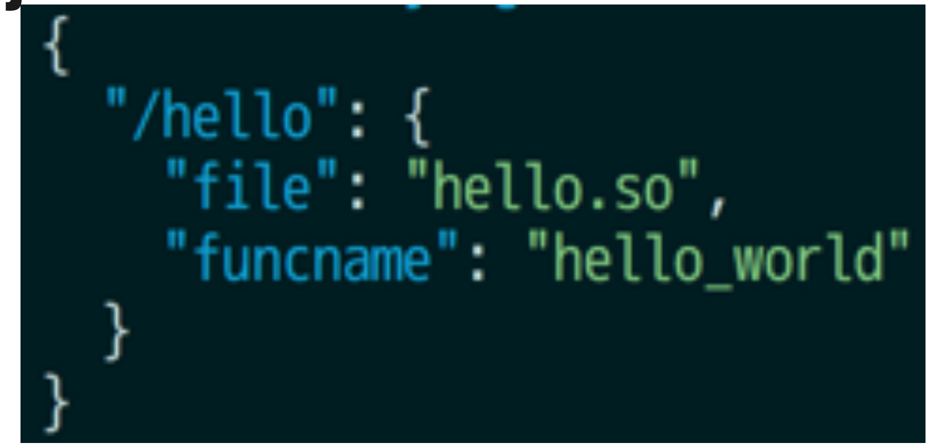
API Comparisons: Setting URL to a callback

hello.cpp → hello.so (by Make rules)



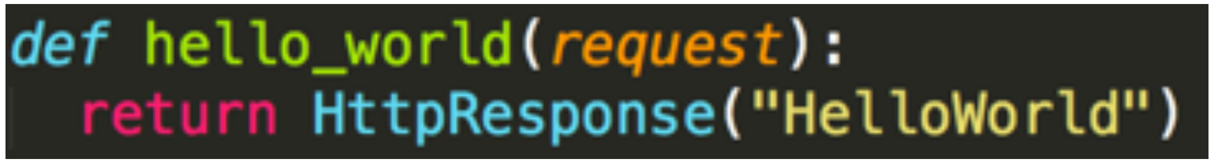
```
extern "C" auto hello_world(HttpRequest req) {
  return HttpResponse("helloWorld");
}
```

urls.json



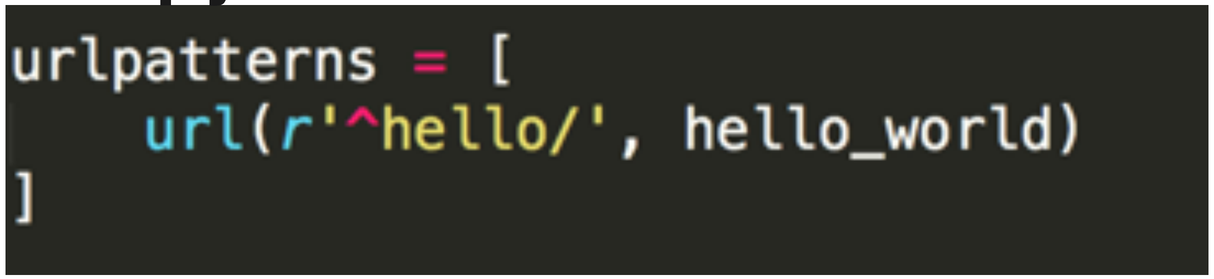
```
{
  "/hello": {
    "file": "hello.so",
    "funcname": "hello_world"
  }
}
```

hello.py



```
def hello_world(request):
  return HttpResponse("HelloWorld")
```

urls.py

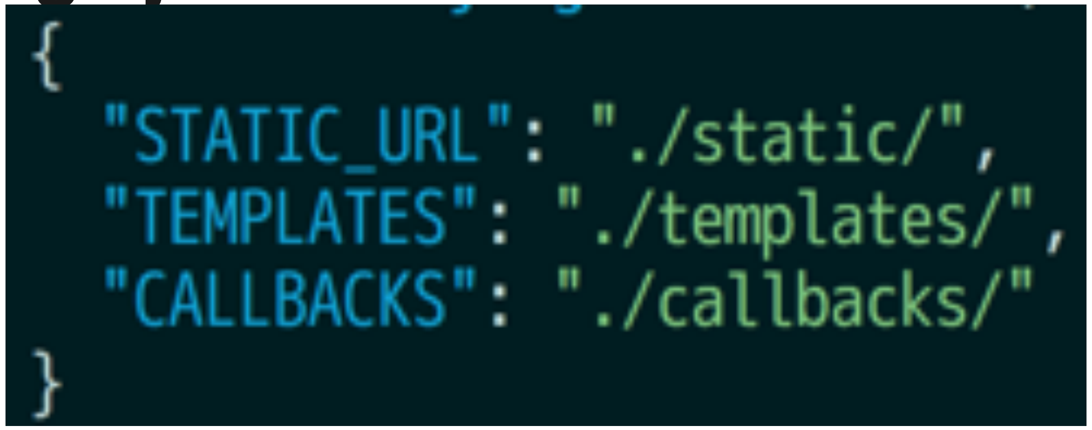


```
urlpatterns = [
    url(r'^hello/', hello_world)
]
```

- extern “C” directive is required for using C’s Dynamic Loading library (`-ldl`)
- When saved, **Mappings in urls.json are automatically reloaded as like urls.py**
  - If hello.so changed, “`touch urls.json`” reloads the new `hello_world()`
- both of urls.json and urls.py can use Regular Expression for pattern

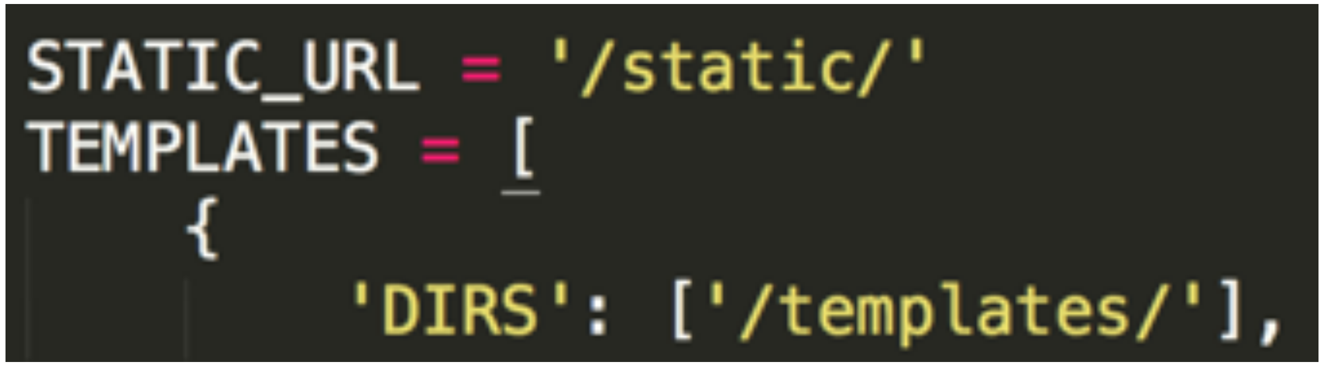
API Comparisons: Setting Custom Static/Template Root Foloders

settings.json



```
{
   "STATIC_URL": "./static/",
   "TEMPLATES": "./templates/",
   "CALLBACKS": "./callbacks/"
}
```

settings.py



```
STATIC_URL = '/static/'
TEMPLATES = [
    {
        'DIRS': ['/templates/'],
```

- Users can set their own root paths
  - templates (fragments of HTML files) are typically reused between apps
- Cjango can also load directory paths for templates and static files
  - **Key observation:** C++ can handle runtime configurations by text files

Explained in detail:
Dynamic Callback Loading

How to reload callback functions

```
#include <dlfcn.h>   // library of C language
// make an alias of a wrapper class for callback function pointer
using callback_t = std::function<HttpResponse(HttpRequest)>;

void *     load_callback(const string& path, const string& func_name) {
    const auto lib = dlopen(path, RTLD_LAZY); // load a .so file
    /*  … error handling here if no such file exists … */
    void *func = dlsym(lib, “callback_hello_world”);
    /*  … error handling if no such func exists … */
    return func;
}

// in Router class
callback_t callback = load_callback(so_filepath, funcname);

callback(request);      // in a worker thread
```

How to reload callback functions

```
#include <dlfcn.h>   // library of C language
// make an alias of a wrapper class for callback function pointer
using callback_t = std::function<HttpResponse(HttpRequest)>;

void *     load_callback(const string& path, const string& func_name) {
    const auto lib = dlopen(path, RTLD_LAZY); // load a .so file
    /*  … error handling here if no such file exists … */
    void *func = dlsym(lib, “callback_hello_world”);
    /*  … error handling if no such func exists … */
    return func;
}

// in Router class
callback_t callback = load_callback(so_filepath, funcname);

callback(request);      // in a worker thread
```

- `dlsym()` reads any typed objects/functions -> generic pointer “void *” used
  - the above code results in “no viable conversion from 'void *' to ‘callback_t'”
  - How should we deal with this in C++?

Approach 1: Use C-style Cast

```
#include <dlfcn.h>   // library of C language
// make an alias of a wrapper class for callback function pointer
using callback_t = std::function<HttpResponse(HttpRequest)>;
using tempcall_t = http::HttpResponse (*)(http::HttpRequest);
tempcall_t load_callback(const string& path, const string& func_name) {
    const auto lib = dlopen(path, RTLD_LAZY); // load a .so file
    /*  … error handling here if no such file exists … */
    void *func = dlsym(lib, “callback_hello_world”);
    /*  … error handling if no such func exists … */
    return (tempcall_t) func;
}

// in Router class
callback_t callback = load_callback(so_filepath, funcname);

callback(request);      // in a worker thread
```

- First convert from `void*` to `tempcall_t`, then to `callback_t`
    - For `(tempcall_t)`, compilers try to apply `const_cast, static_cast, and reinterpret_cast`, in this order
    - The conversion from function pointer type tempcall_t to callback_t is known

Approach 2: Use more specific reinterpret_cast / static_cast

```
#include <dlfcn.h>   // library of C language
// make an alias of a wrapper class for callback function pointer
using callback_t = std::function<HttpResponse(HttpRequest)>;
using tempcall_t = http::HttpResponse (*)(http::HttpRequest);
tempcall_t load_callback(const string& path, const string& func_name) {
    const auto lib = dlopen(path, RTLD_LAZY); // load a .so file
    /*  … error handling here if no such file exists … */
    void *func = dlsym(lib, “callback_hello_world”);
    /*  … error handling if no such func exists … */
    return reinterpret_cast<tempcall_t>(func);
}

// in Router class
callback_t callback = static_cast<callback_t>( load_callback(…) );

callback(request);      // in a worker thread
```

- First convert from `void*` to `tempcall_t`, then to `callback_t`
    - Since we know (in next slide) the successful cast is only reinterpret_cast,
      just use it
    - The conversion from function pointer type tempcall_t to callback_t is

Guilty or not guilty: reinterpret_cast

reinterpret_cast is considered dangerous

- use underlying bit patterns according to the specified type information
  - even if not-HttpRequest object is passed as first argument, continue to execute with the underlying bits

    e.g. `HttpResponse invalid_call(int x)`

    can be compiled and executed as callbacks

However, reinterpret_cast for dlsym() is POSIX-compliant usage

- “Only the following conversions can be done with `reinterpret_cast`, …

  8) On some implementations (in particular, on any POSIX compatible system as required by `dlsym`), a function pointer can be converted to void* or any other object pointer, or vice versa.”

  (Source: cppreference.com, `reinterpret_cast`)

Summary: Dynamic Callback Loading has pros and cons

Pros

- dynamic reloading (removal/addition/change)
    - urls-json can be similar with Django’s urjs.py
- separate compalitation of callbacks
    - other C++ web app frameworks (e.g. Crow, header-only library) suffers long-compile time

Cons

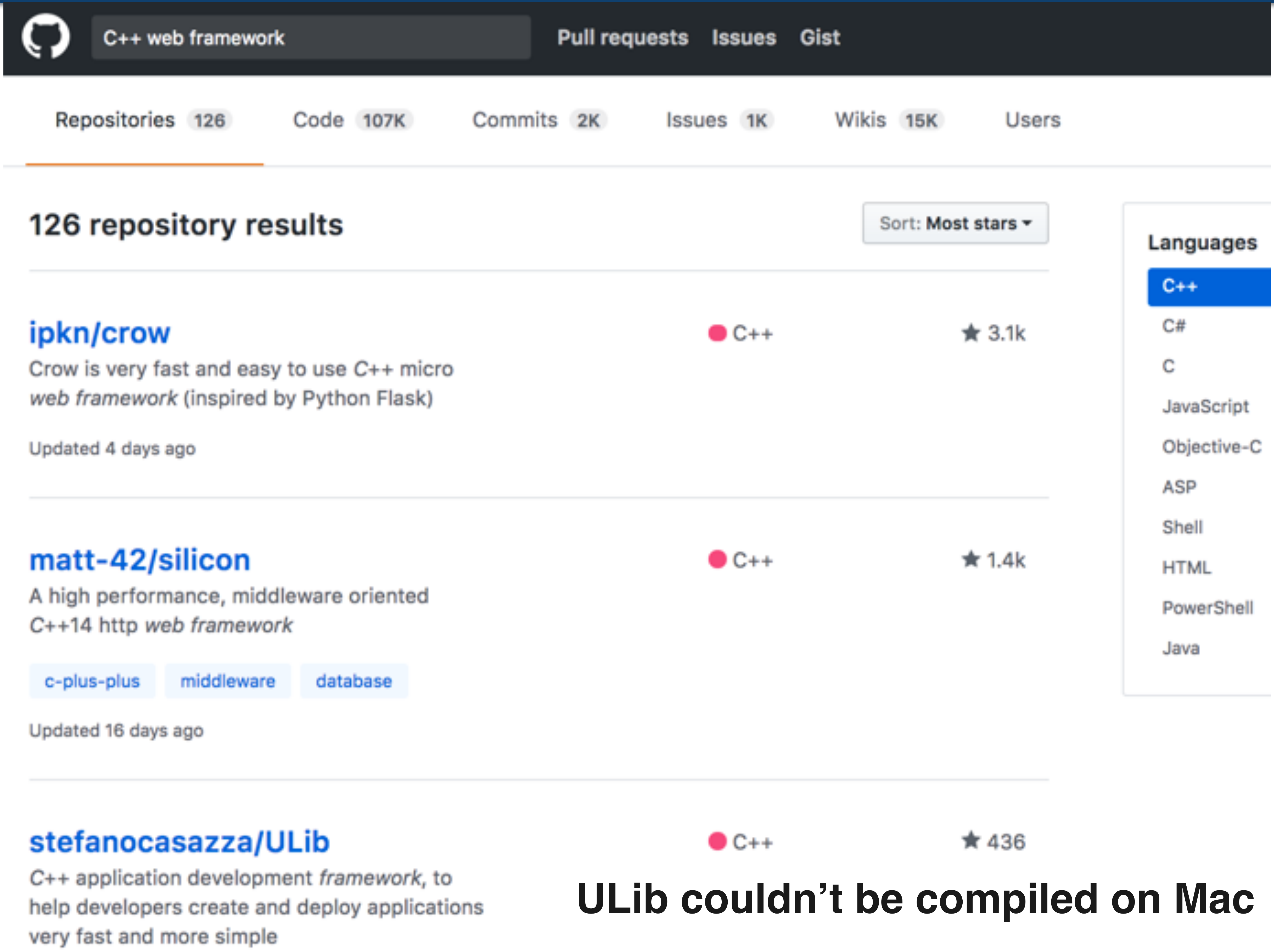
- need for insecure reinterpret_cast
    - cannot check argument/return types even with std::function.target_type()

Cjango’s design: take both

- prepare a compile flag CJANGO_DYNLOAD for disabling the dynload feature
    - **if removed, urls-json is not used and all dynload codes are compiled away**
        - users set each URL-callback mappings by `add_route(“/hello”, call_hello)` in main app files

Performance

Compare with Github's Top 3 C++ Web App Frameworks (and Django)



126 repository results

ipkn/crow ● C++ ★ 3.1k

Crow is very fast and easy to use C++ micro *web framework* (inspired by Python Flask)

Updated 4 days ago

matt-42/silicon ● C++ ★ 1.4k

A high performance, middleware oriented C++14 http *web framework*

c-plus-plus middleware database

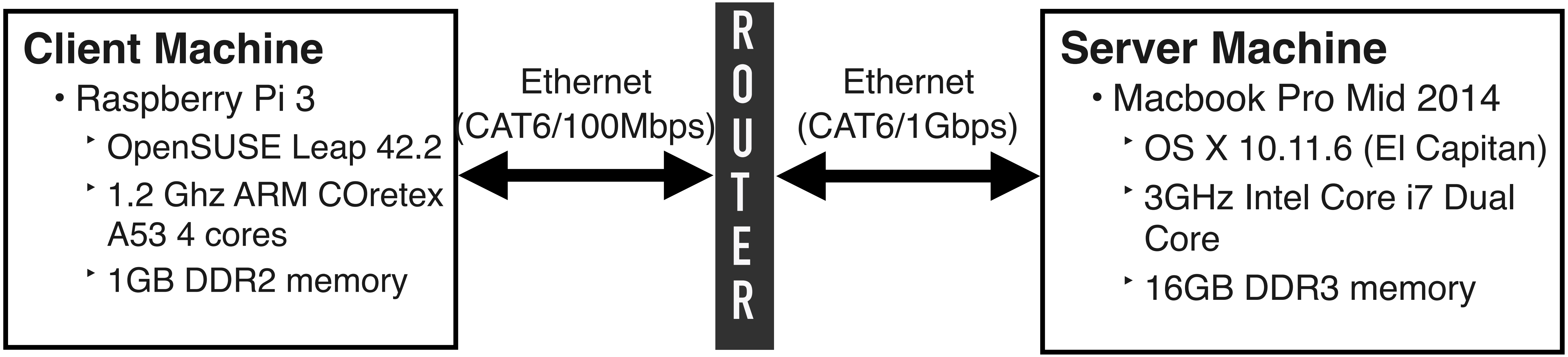
Updated 16 days ago

stefanocasazza/ULib ● C++ ★ 436

C++ application development *framework*, to help developers create and deploy applications very fast and more simple

ULib couldn't be compiled on Mac

Experiment Conditions (Results in Next Slide)



Client Machine

- Raspberry Pi 3
  - OpenSUSE Leap 42.2
  - 1.2 Ghz ARM COretex A53 4 cores
  - 1GB DDR2 memory

Ethernet (CAT6/100Mbps)

ROUTER

Ethernet (CAT6/1Gbps)

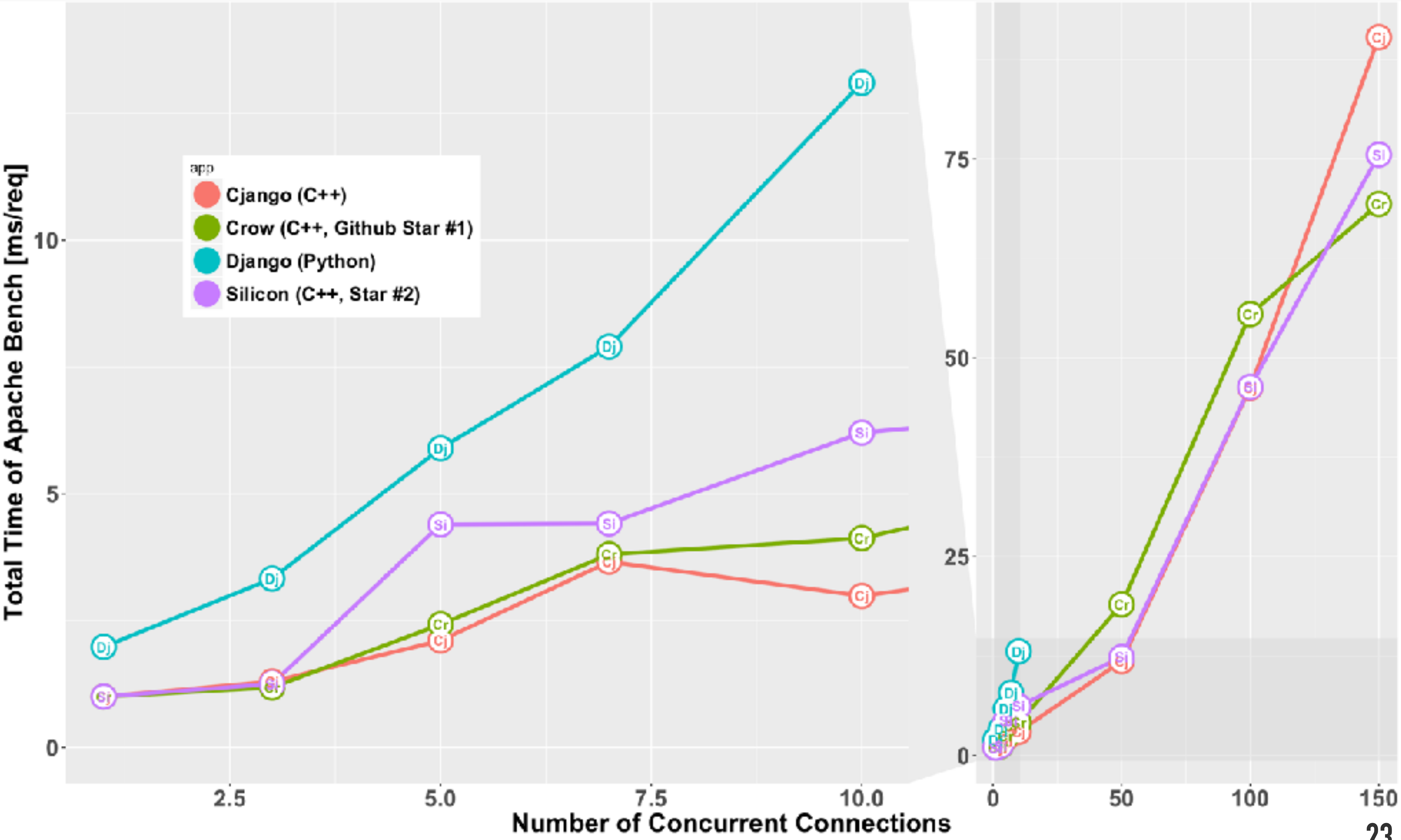
Server Machine

- Macbook Pro Mid 2014
  - OS X 10.11.6 (El Capitan)
  - 3GHz Intel Core i7 Dual Core
  - 16GB DDR3 memory

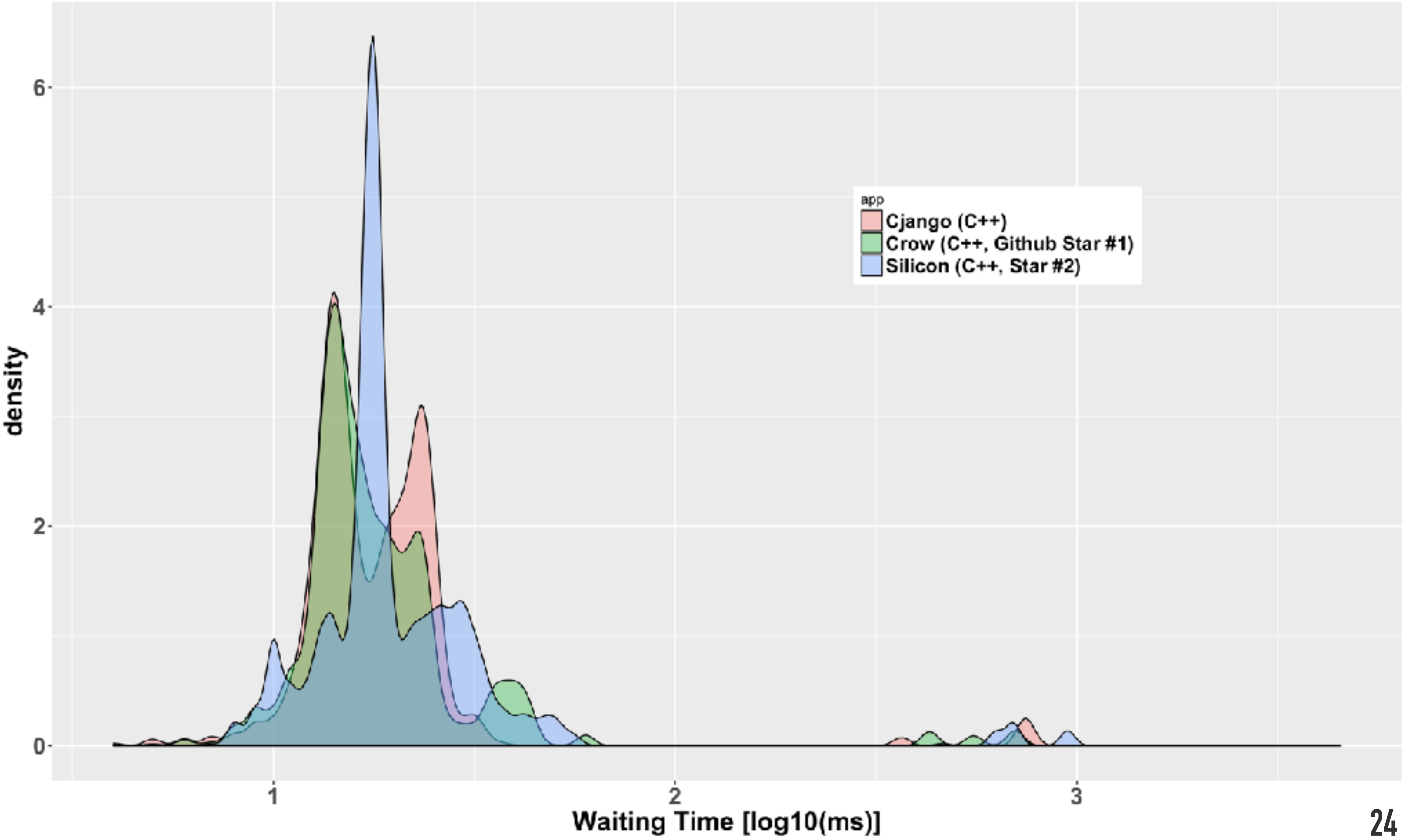
Settings

- One of Django, Cjango, Crow or Silicon apps is running on Server Machine
  - each app serves a simple “HelloWorld” string on port 8000
  - **goal:** measure effects on overheads of Cjango’s dynamic loading, original HTTP parser, original socket handlings
- Client Machine accesses by **A**patch **B**ench (common Http benchmark software)
  - `ab -n 10000 -c {1,3,5,7,10, 50, 100, 150} http://[IP]:8000/`
    - -n : # of requests
    - -c : # of concurrent requests
- Raw data and more details are uploaded at github Cjango-Unchained/src/bench/

Cjango beats Django and has comparable speeds for other C++ frameworks against middle-sized concurrent HTTP requests



150 concurrent request tests have two peaks for “waiting time” (time between the last Request byte sent and the 1st Response byte received on client).



Individual Contributions

Mengdi Lin (team leader)

- came up with "Unchained"
- proposed Django-like web framework for our project
- enlightened us on the http communication mechanisms
- suggested Cjango's catchcopy / direction
- set priorities of developement orders
- Designed Django's main APIs
- write API doc for HttpRequest/Response
- **Implementation**
  - wrote 99% of large codebases under http_parser/
    - HTTP parsing
    - GET/POST handling
    - session support
  - sample demo app for HTTP Session
  - create libhttp_{resonse, request}.so for callbacks
  - write HTTP Session tutorial

Yi Qi

- agreed with the title "Cjango-Unchained"
- shared a few previous attempts for modular servers
- proposed dlopen() functions for minimizing downtime
- found Armadillo's side-by-side comparisons
- proposed json response
- tried two different implementations for socket handlings and found faster one
- introduced easy-to-use _DEBUG macros
- write API doc for App
- **Implementation**
  - App class
  - select() asynchronous HTTP request handling with non-blocking sockets
  - mock application/HTTP request
  - Regular Expression URL matching rules
  - integration test suites

Yasutaka Tanaka

- proposed separate compilation/dynamic loading for callbacks
- Validated the feasibility of dynamic callback loading
- researched competitors
- Measure performance and visualize the results
- Created first draft of slides (~30 pages)
- Draw Cjango logo & dog
- deploy Cjango official site
- write README.md
- write API doc for Router
- **Implementation**
  - Router class
  - Dynamic Loading (Monitor Thread logic, dlopen() logic, appropriate casts)
  - _SPDLOG macros
  - simplefilewatcher customization for our flags
  - Makefile dependency checks and refactoring

Future

Add features targeted on personal use

- http 1.1
- https
- sqlite3
- URL queries/parameters

Learn more:

Cjango is hosted on Github

- mengdilin/Cjango-Unchained
- tutorials
- API documents (by doxygen) are under /src/doc

References

- C++ language core issue reports http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_defects.html#195
    - spec about conversion between object pointer and function pointer (dlsym)
- Crow’s reputation after publicity
    - https://news.ycombinator.com/item?id=8002604
    - Discussions on compile-time URL validation and header-only effectivity

Acknowledgments

Professor

Jonathan

David (his explanation on HexRacer gave us ideas about using Json)

Creators of web app frameworks including Django, Crow, Silicon, TreeFrog

Library Developers of nlohmann/json, simplefilewatcher

The designer of Chain image in logo

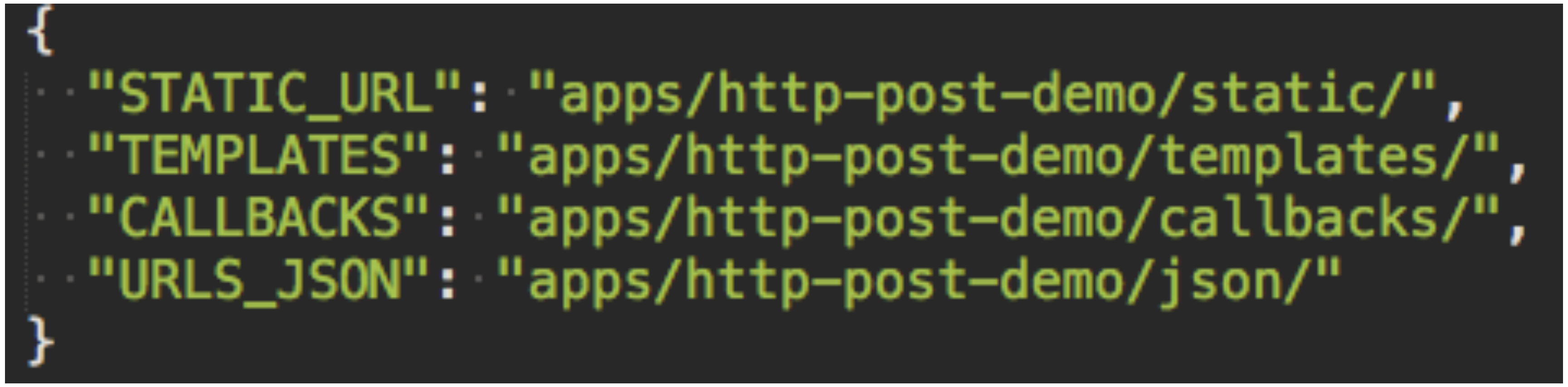
The Documentation writer for Armadillo
- compare notations with Matlab side-by-side

Tutorial: simple http server

- Step 1: directory setup:
  - callbacks/: a directory containing all callback function definitions
  - json/: a directory containing settings.json and urls.json
  - static/: a directory containing all static files that will be referenced from html files (such as images, js files, and css files)
  - templates/: all html files

Tutorial: simple http server

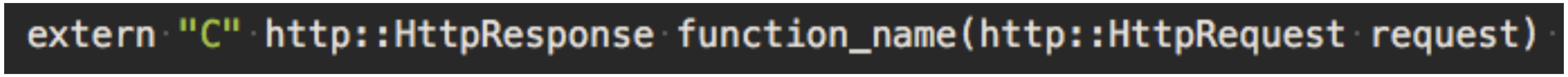
- Step 2: Tell Cjango about your directory setup:
  - Specify the paths to your four directories in json/settings.json file



```
{
  "STATIC_URL": "apps/http-post-demo/static/",
  "TEMPLATES": "apps/http-post-demo/templates/",
  "CALLBACKS": "apps/http-post-demo/callbacks/",
  "URLS_JSON": "apps/http-post-demo/json/"
}
```

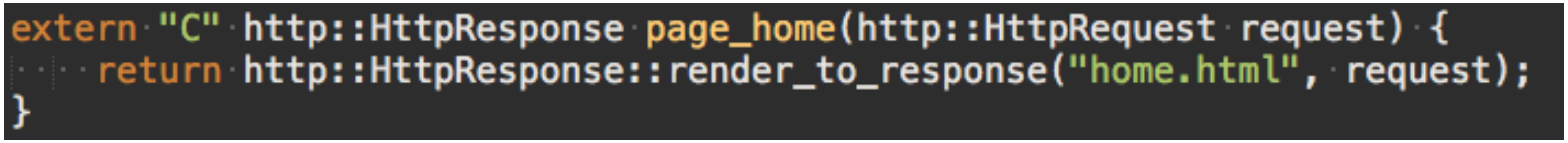
Tutorial: simple http server

- Step 3: Write a callback function
  - all callback functions must have the function signature



```
extern "C" http::HttpResponse function_name(http::HttpRequest request)
```

    - extern “C” is necessary for dynamic reloading
  - a simple callback function called “page_home” that returns “home.html” for a request looks like this:

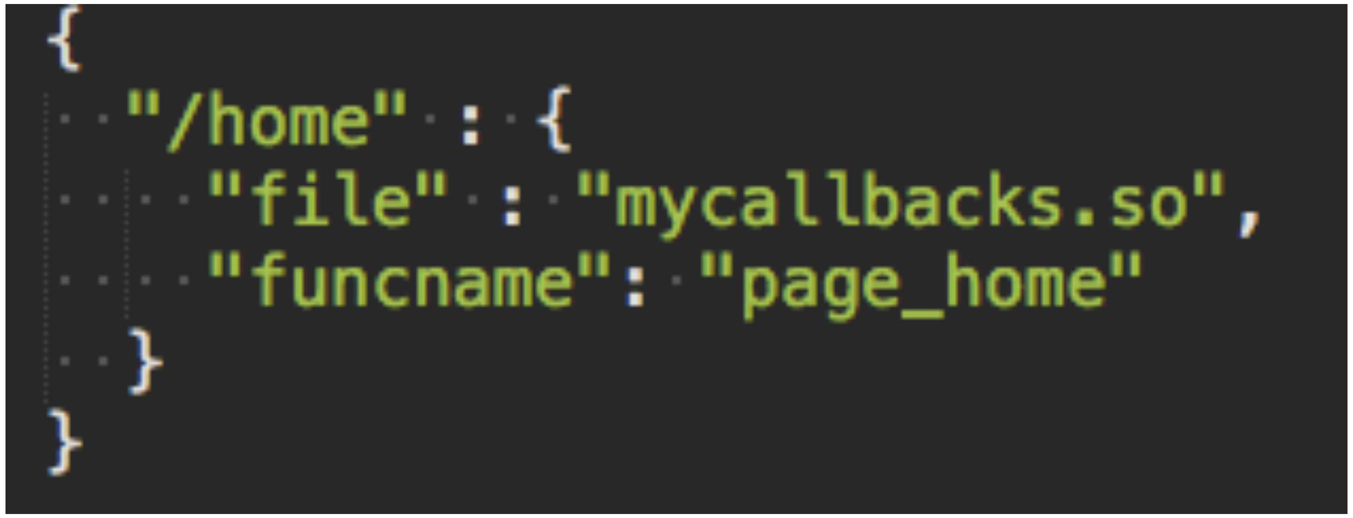


```
extern "C" http::HttpResponse page_home(http::HttpRequest request) {
    return http::HttpResponse::render_to_response("home.html", request);
}
```

    - notice that we only need to specify the html’s file name without any path information. Cjango will find “home.html” in our templates/ directory.
  - compile it into a .so file (we have provided a generic Makefile for users’ convenience)

Tutorial: simple simple http server

- Step 4: Define url mapping
  - Provide a url path “page_home” corresponds to inside json/urls.json



```
{
  "/home" : {
    "file" : "mycallbacks.so",
    "funcname": "page_home"
  }
}
```

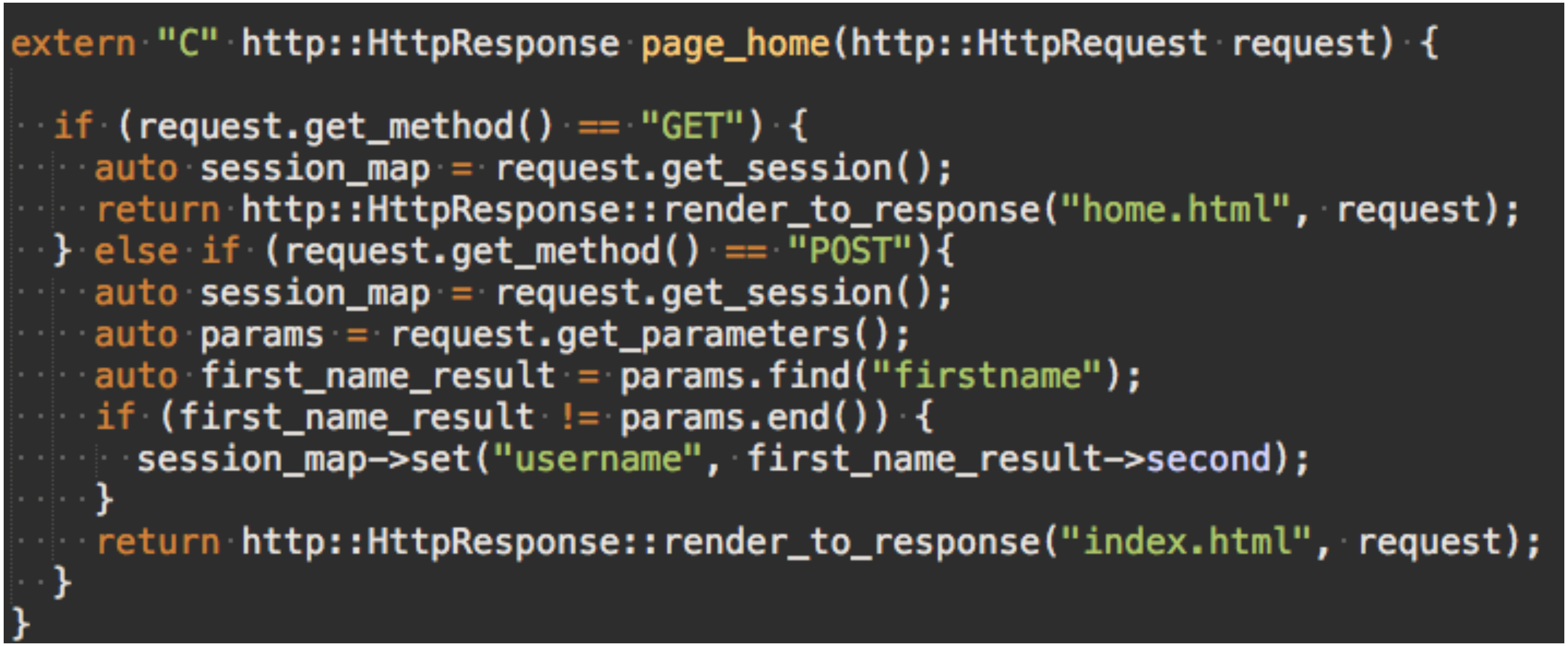
  - Now Cjango will find mycallbacks.so inside callbacks/ directory and run “page_home” whenever a client visits /home path

Tutorial: HttpRequest API

- HttpRequest class provides a helpful interface for retrieving a http request’s fields
  - request.get_method() -> returns http request’s method
  - request.get_meta() -> returns a map of http request’s headers
  - request.get_parameters() -> returns a map of http request’s parameters
  - request.get_session() -> returns a session object associated with the current request

Tutorial: Using HttpRequest In Callback Functions

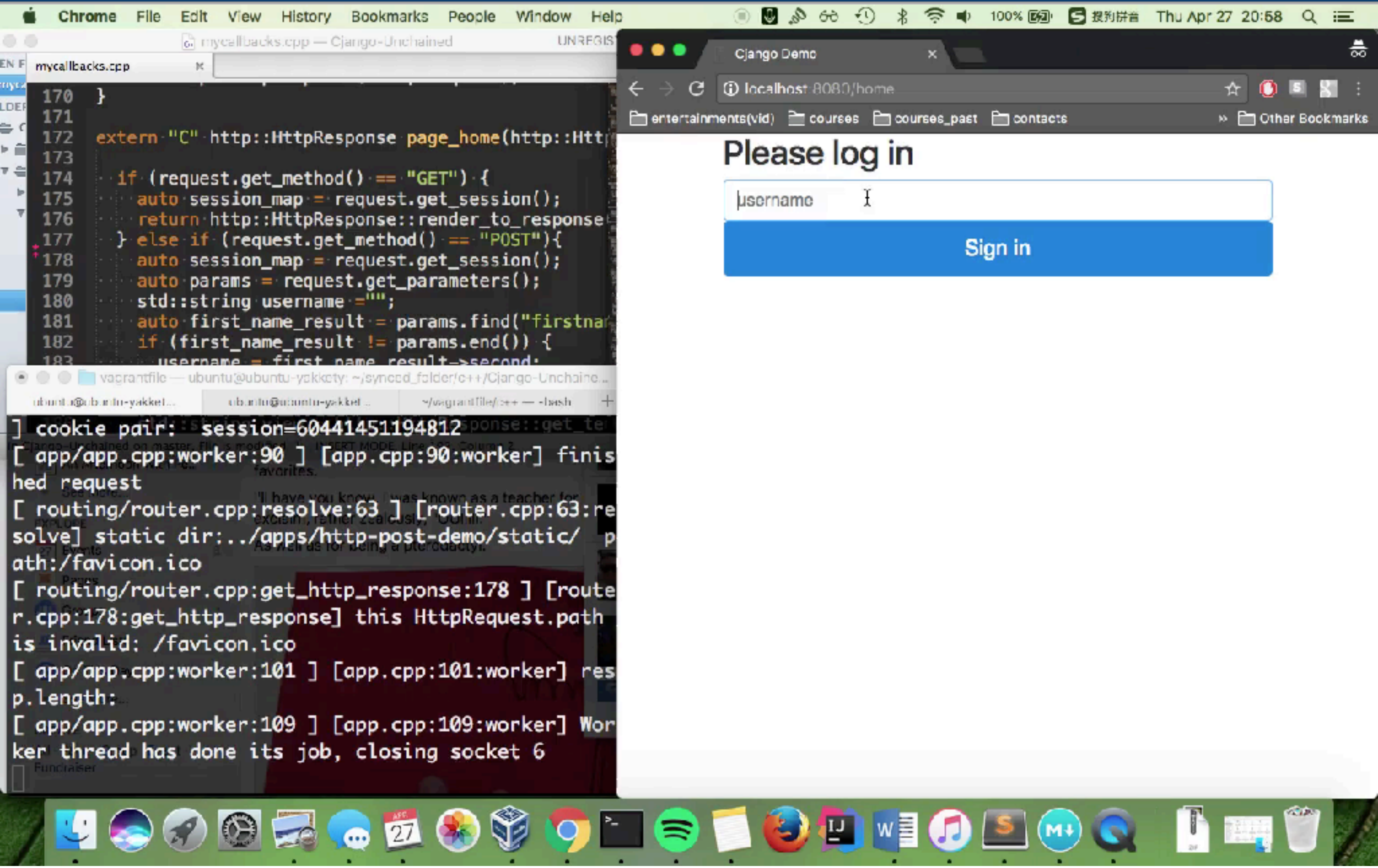
- **Using HttpRequest, our callback functions now can do something more complicated:**



```
extern "C" http::HttpResponse page_home(http::HttpRequest request) {

  if (request.get_method() == "GET") {
    auto session_map = request.get_session();
    return http::HttpResponse::render_to_response("home.html", request);
  } else if (request.get_method() == "POST"){
    auto session_map = request.get_session();
    auto params = request.get_parameters();
    auto first_name_result = params.find("firstname");
    if (first_name_result != params.end()) {
      session_map->set("username", first_name_result->second);
    }
    return http::HttpResponse::render_to_response("index.html", request);
  }
}
```

Demo Time



Chrome File Edit View History Bookmarks People Window Help
Thu Apr 27 20:58
mycallbacks.cpp — Cjango-Unchained
Cjango Demo
localhost:8080/home
entertainments(vid) courses courses_past contacts Other Bookmarks
Please log in
username
Sign in
vagrantfile — ubuntu@ubuntu-yakkety: ~/synced_folder/c++/Cjango-Unchained

```
}

extern "C" http::HttpResponse page_home(http::Htt

  if (request.get_method() == "GET") {
    auto session_map = request.get_session();
    return http::HttpResponse::render_to_response
  } else if (request.get_method() == "POST"){
    auto session_map = request.get_session();
    auto params = request.get_parameters();
    std::string username ="";
    auto first_name_result = params.find("firstna
    if (first_name_result != params.end()) {
      username = first_name_result->second;
```

```
] cookie pair:  session=60441451194812
[ app/app.cpp:worker:90 ] [app.cpp:90:worker] finis
hed request
[ routing/router.cpp:resolve:63 ] [router.cpp:63:re
solve] static dir:../apps/http-post-demo/static/  p
ath:/favicon.ico
[ routing/router.cpp:get_http_response:178 ] [route
r.cpp:178:get_http_response] this HttpRequest.path
is invalid: /favicon.ico
[ app/app.cpp:worker:101 ] [app.cpp:101:worker] res
p.length:
[ app/app.cpp:worker:109 ] [app.cpp:109:worker] Wor
ker thread has done its job, closing socket 6
```