CSC309H1S

Programming on the Web

Winter 2023

Lecture 4: Introduction to Backend Development

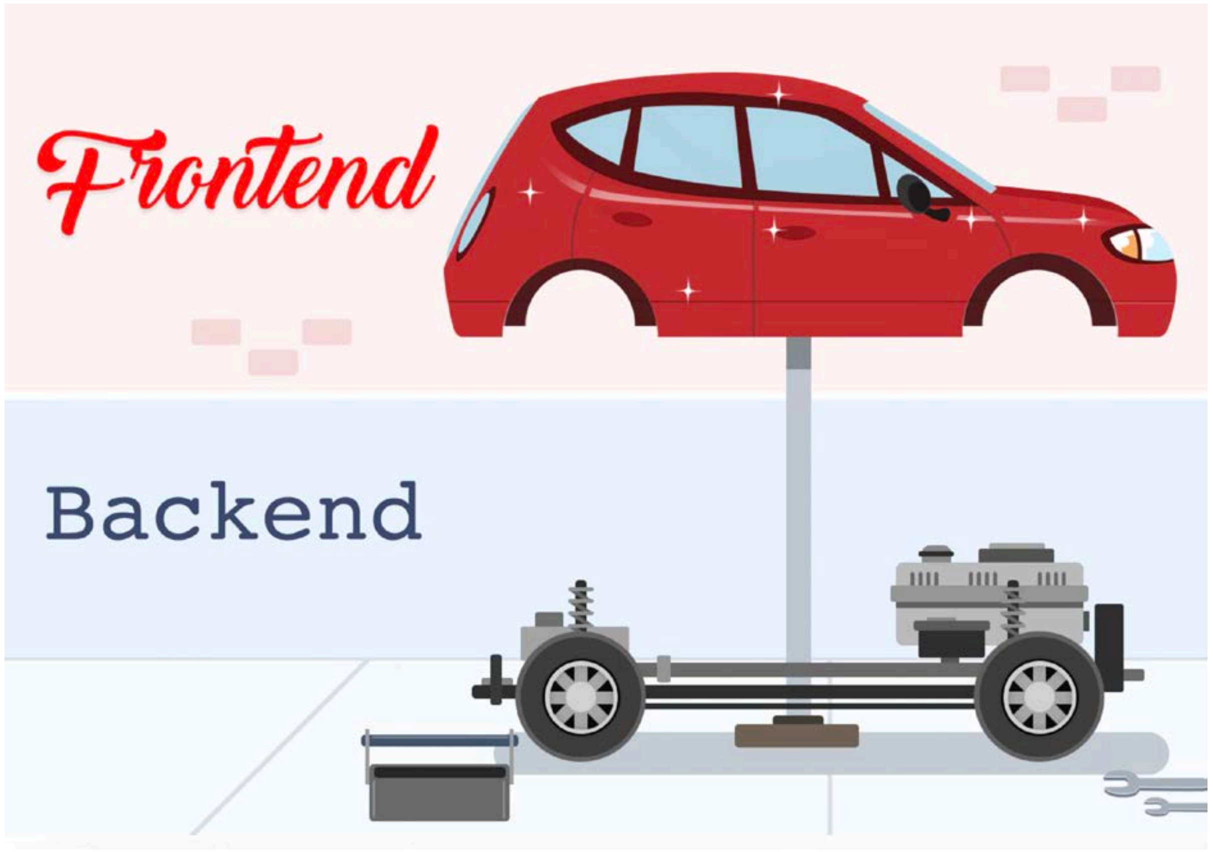
Instructor: Kuei (Jack) Sun

Department of Computer Science
University of Toronto

v1.01

Web Development

- Separation of concern
- Frontend
  - Focuses on *presentation*
  - Part of the *client*
  - Faces the *end-user*
  - Provides user-friendly interface
- Backend
  - Focuses on *data access*
  - Part of the *server**
    - Server can do some frontend work
  - Data storage and business logic



Source: blog.back4app.com

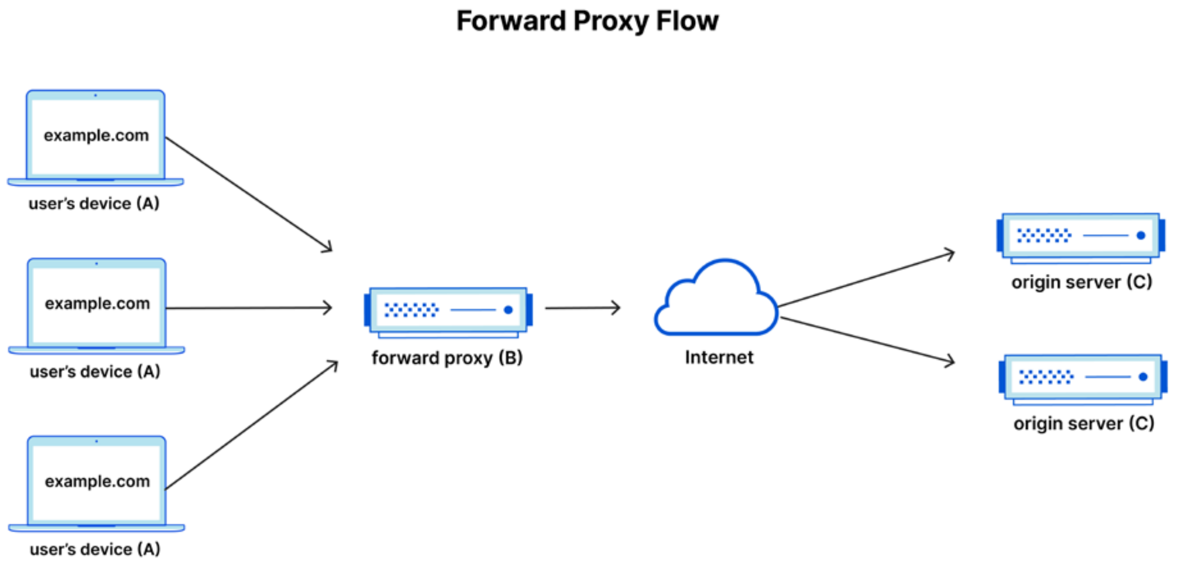
Abstraction in Web Architecture

Source: https://www.reddit.com/r/ProgrammerHumor/comments/m187c4/backend_vs_frontend/

Web Server

- Listens on specific port(s) for HTTP/HTTPS requests
- Examples: Apache, Nginx
- Handles incoming connections
  - Generates a response (dynamic content)
  - Fetches a file (static content)
    - Can be cached in memory for faster subsequent access
  - To act as a proxy between the client and the origin server
    - Forward proxy: sits in front of client devices, before Internet access
    - Reverse proxy: sits in front of origin server, after Internet access
    - Note: this only works for HTTP requests (unlike VPN)

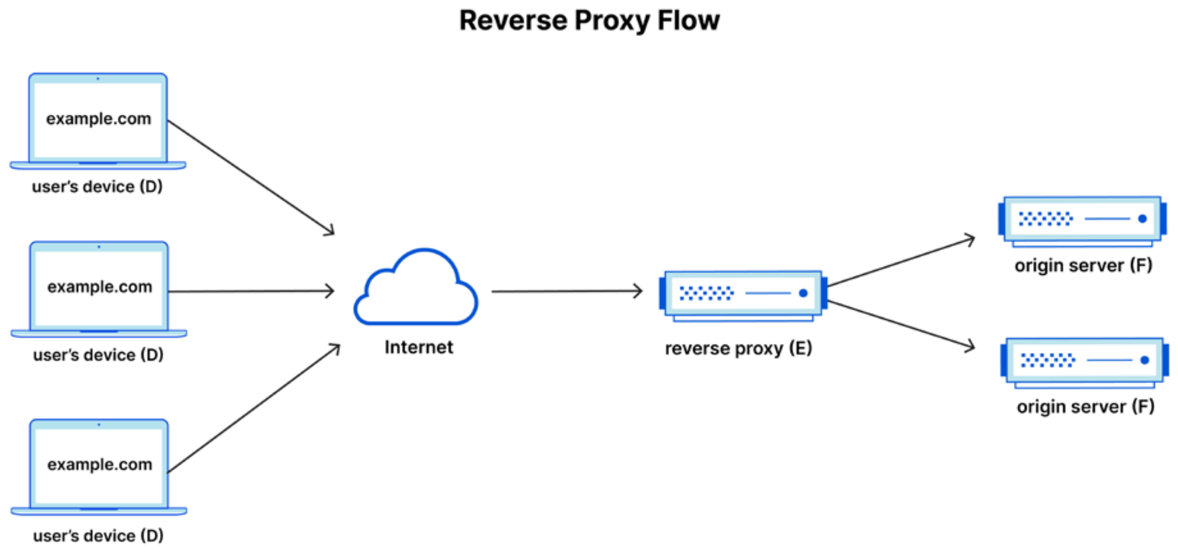
Forward Proxy



- Usages
  - Block or monitor access to certain content, e.g., on a school network
  - Improves security and anonymity by hiding user's IP address
  - Can "sometimes" circumvent regional restrictions

https://www.cloudflare.com/en-gb/learning/cdn/glossary/reverse-proxy/

Reverse Proxy

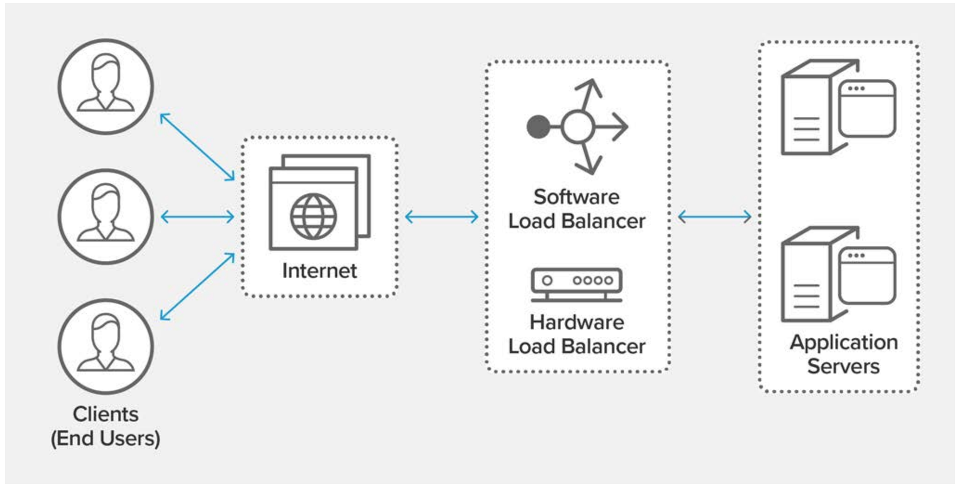


- Usages
  - Caches content for geographically distant web server
  - Acts as a front for security purposes, e.g., encryption, prevent DDoS attack
  - Provides load balancing

https://www.cloudflare.com/en-gb/learning/cdn/glossary/reverse-proxy/

Load Balancer

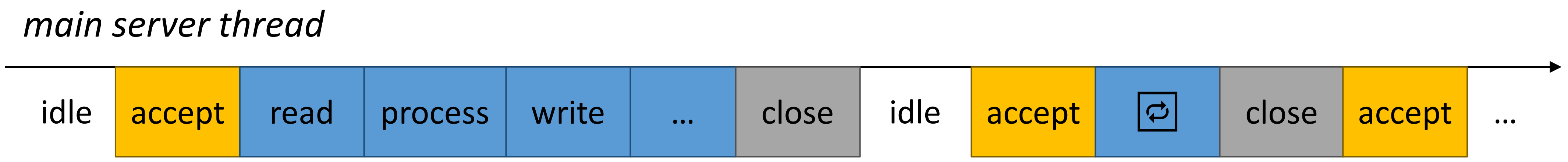
- Popular websites can serve *millions* of concurrent requests!
- Load balancer distributes incoming requests among backend servers
- Ensures all servers have similar utilization
- Allows adding/removing servers based on current demand
  - Reduces energy consumption during times of low traffic



Source: https://www.nginx.com/resources/glossary/load-balancing/

Web Server Architecture

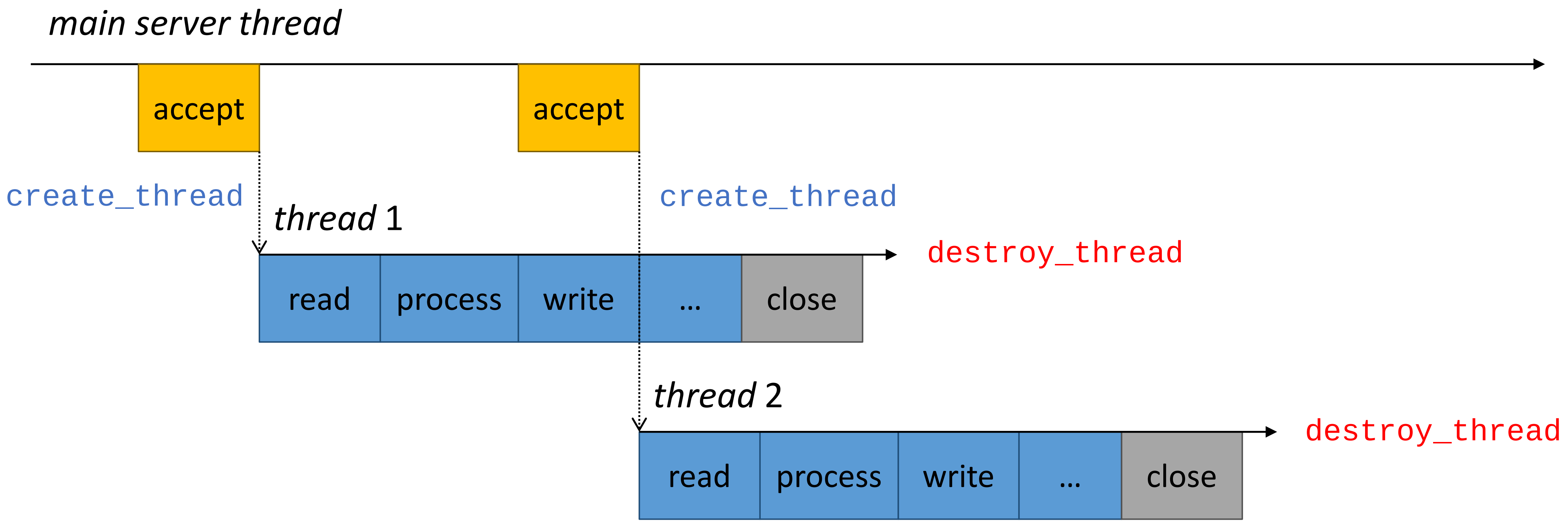
- Single-threaded server



- Cannot only handle one connection at a time!

Web Server Architecture

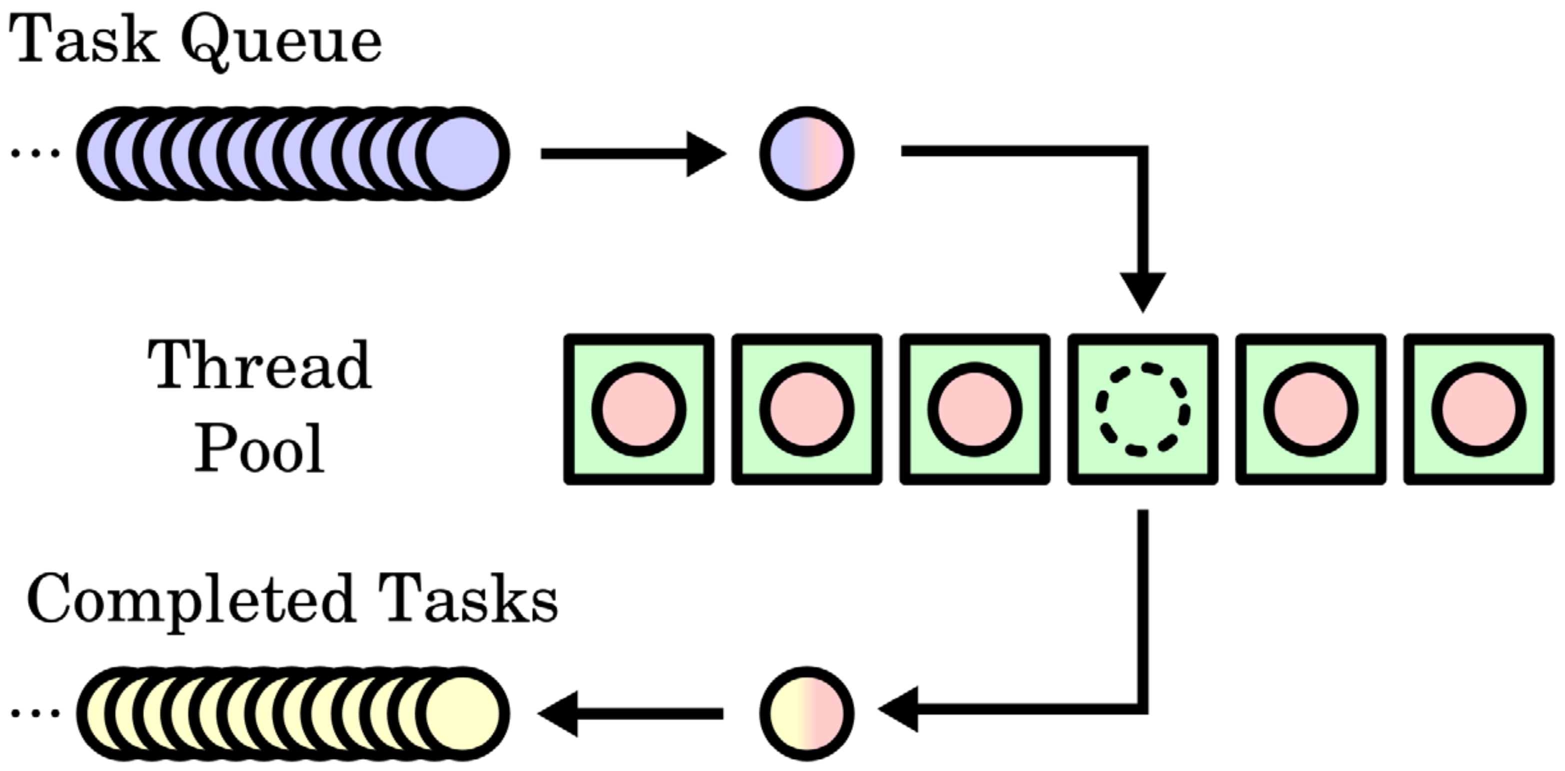
- Multi-threaded server



- Problem: creating threads is expensive and http requests are short-lived.

Web Server Architecture

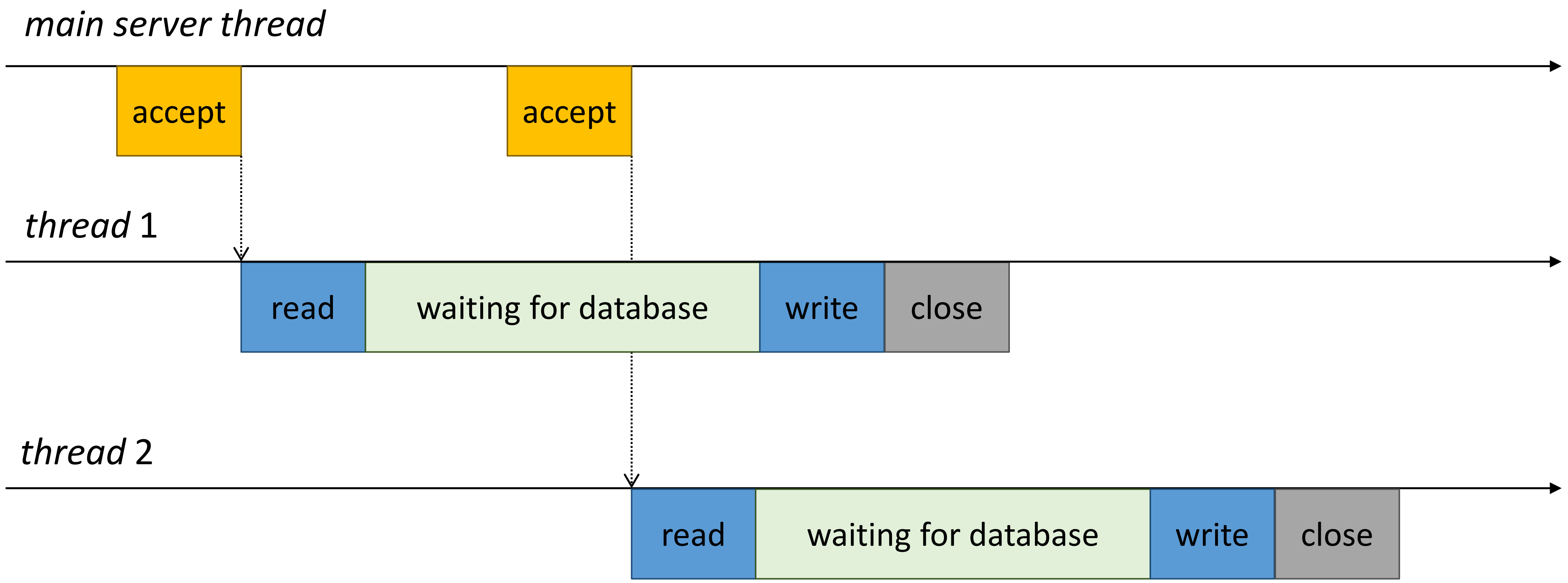
- Multi-threaded server with thread pool



https://en.wikipedia.org/wiki/Thread_pool

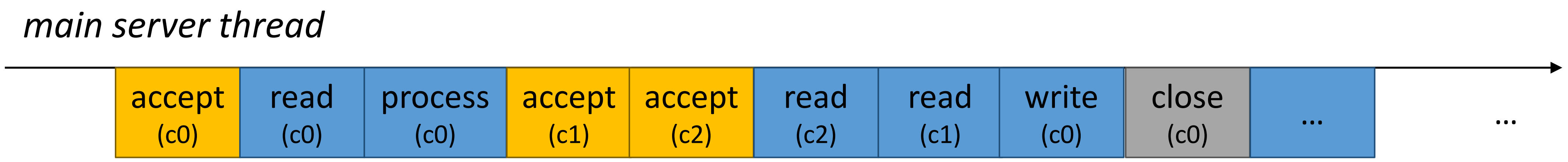
Web Server Architecture

- Problem: threads are frequently blocked waiting for IO



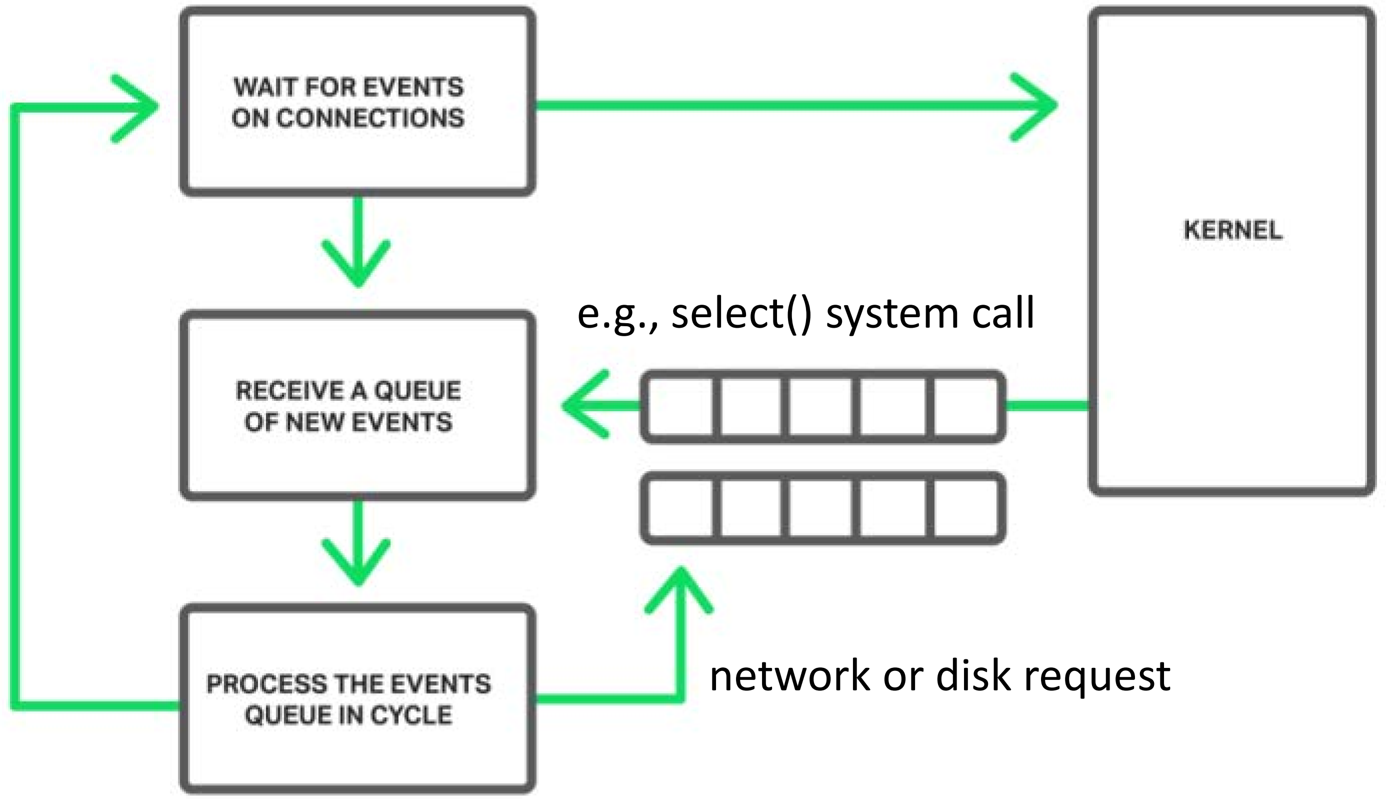
Web Server Architecture

- Event-driven web server



- Events are queued and executed in order
- An http request can be broken up into *states*
- Each state transition is an *event*, processed asynchronously
- No overhead of switching between threads
  - Can be combined with thread pool to utilize more physical CPUs

Nginx Event Loop


WAIT FOR EVENTS ON CONNECTIONS
RECEIVE A QUEUE OF NEW EVENTS
PROCESS THE EVENTS QUEUE IN CYCLE
KERNEL

e.g., select() system call

network or disk request

https://www.nginx.com/blog/thread-pools-boost-performance-9x/

Common Gateway Interface

- Allows web server to run an external program to process requests
  - In early days, to process forms, e.g., POST request
- Separates web server from web application
  - Any web application can use any web server to generate dynamic content
  - Web application can be compiled or interpreted program
- Care is required due to execution of arbitrary code
- Nowadays, there are many similar standards
  - WSGI (web server Gateway Interface): used by Python programs
  - Rack: used by Ruby programs
  - JSGI (JavaScript Gateway Interface): used by JavaScript programs

Programming Languages

- Technically, any language can be used in the backend
  - It just needs a library that understands HTTP protocol
- Python, Java, JavaScript, php, Ruby
- Popular web programming languages are mostly *interpreted*
  - Portable: can run on many operating systems
  - Flexible: does not require compilation
  - Quick and easy to make changes on the fly
  - Bottleneck of most servers is *network*, not code execution
    - RAM access is 10^6 times faster than network access (ns vs ms)
    - Most performance optimization focuses on reducing network latency
      - E.g., using CDN and asynchronous requests

Runtime Environment

- Hardware and software infrastructure for running code
- It is *not* the programming language itself
- Examples:
  - CPython
    - Interpreter that runs the Python programming language
  - Node.js
    - Runs on V8 JavaScript Engine
    - Favorite among web developers
      - Can write both frontend and backend with just one programming language
  - PHP interpreter
    - Runs the PHP programming language

Backend Frameworks

- Libraries on the server-side that helps build a web application
- Avoids doing everything from scratch!
  - Listen on a port, process HTTP request, retrieve data from storage, process data, create HTTP responses, etc.
  - Don’t reinvent the wheel

- PHP: Laravel, CodeIgniter
- Python: Django, Flask, FastAPI
- JavaScript: ExpressJS, Spring
- Ruby: Ruby on Rails

https://www.geeksforgeeks.org/top-10-django-apps-and-why-companies-are-using-it/

Python Project

- Requires use of external packages
- Python's package manager: `pip`
  - Helps install and manage software packages
  - Automatically handles *dependencies*
    - Other packages (and their versions) that are required to use this package
- `pip3 install Django`
  - Command to install latest version of Django (4.1 as of Jan 2023)
- Multiple projects
  - How do I deal with different versions of Django? (or even Python itself)

Virtual Environment

- Manages separate package installations for different projects
- An *isolated* environment with its own version of everything
  - Python interpreter, `pip`, and packages
  - Avoids dependency conflicts and version differences
    - Code for one version of Django may not run for a different version
- `virtualenv`
  - Provides lightweight encapsulation of Python dependencies
  - Note: does not encapsulate the operating system (heavyweight)
- Create a new virtual environment (with a specific Python version)
  - `virtualenv -p /usr/bin/python3.9 venv`

name of virtual environment

Virtual Environments

- `source venv/bin/activate`
  - Activates the virtual environment
  - Note that `venv` is the folder where you created your virtual environment
- `deactivate`
  - Deactivates the virtual environment (if you're in one)
- Packages will not be installed globally via `pip`
- To remove a virtual environment, simply delete the folder (`venv`)
- Keep a text file that includes all the required packages
  - To recreate the virtual environment, simply run:

```
pip install -r packages.txt
```

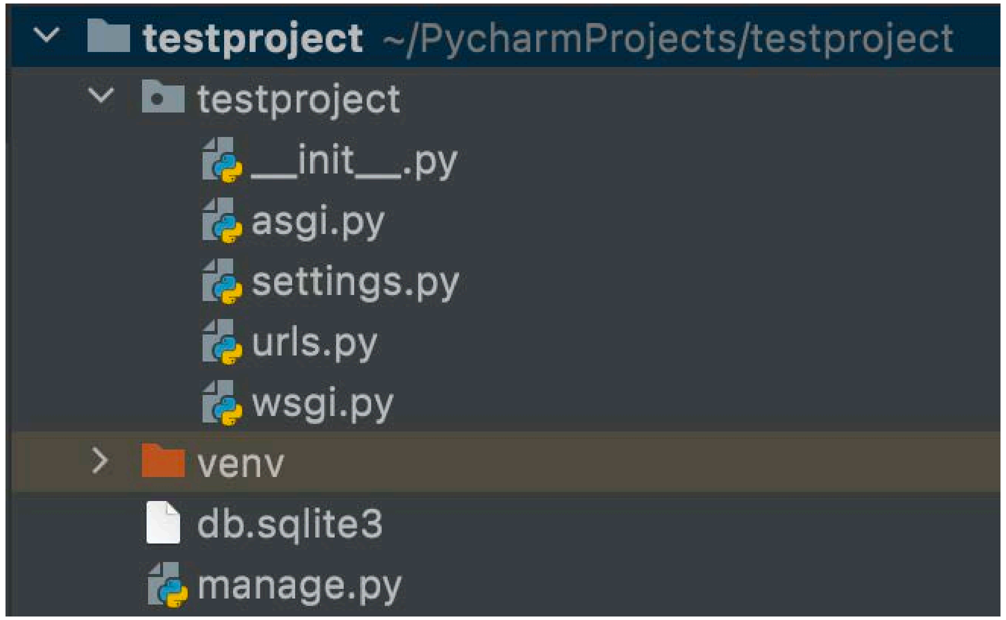
Start a Django Project

1. Create the project folder
2. set up virtual environment and install Django
3. Run the following command:
    - `django-admin startproject <name> .` ← Create the project in the current directory
    - This creates the skeleton code for your project

- You should see the following files created:
    - manage.py: a command-line utility that can do various things
    - A folder that's the same as your project name: contains project-wide settings
- For an alternative tutorial, visit:
    - https://docs.djangoproject.com/en/4.1/intro/tutorial01/ (highly suggested)

Development Server

- Used for testing and development *only*
  - Not suitable for deployment
- `python3 manage.py runserver`
  - Starts the development server
- Your website is accessible at:
  - http://localhost:8000
  - localhost: domain name for this machine
  - 8000: the port number
    - To avoid conflict with actual web server
- Ignore the migration warning for now



```
testproject ~/PycharmProjects/testproject
  testproject
    __init__.py
    asgi.py
    settings.py
    urls.py
    wsgi.py
  venv
  db.sqlite3
  manage.py
```

Django apps

- Django is intended for big projects
  - Can have hundreds of web pages, each with different URL
- Project is organized into *apps*
- An app is a set of related concepts and functionalities
  - E.g., an app to manage accounts, another app to manage products
- `./manage.py startapp <name>`
  - Creates a new app and its folder
    - Contains views.py (today), migration folder, models.py, and admins.py (next week)
- Remember to add the app name to INSTALLED_APPS in settings.py
  - Otherwise, it won't be loaded

Django View

- Code that runs when a specific endpoint, i.e., URL, is requested
  - Can be any callable, e.g., function, class that implements `__call__`
- Example: simple view

```python
from django.http import HttpResponse

def hello(request):
    return HttpResponse("hello")
```

- Function should take an argument, usually named request
- It should return an HttpResponse object

From URL to View

1. Create a file (preferably named urls.py) in the app folder

```
from django.urls import path
from . import views
urlpatterns = [ path('hello', views.hello), ]
```

2. Modify the project's urls.py
   - Add an entry to the list named `urlpatterns`

```
from django.contrib import admin
from django.urls import include, path
urlpatterns = [
    path('test/', include('testapp.urls')),
    path('admin/', admin.site.urls),
]
```

Add this line

- View is now accessible through this URL: `/test/hello`

URL Dispatcher

- Attempts to match URL from top to bottom of `urlpatterns`
- Can capture values from an URL and pass them to the view
- Example:

```
path('hello/<str:name>/<int:age>', hello)
```

- The corresponding view function now takes two extra arguments:

```
def hello(request, name, age):
```

- Exercise
  - Do Question 6 on Quercus

HTTP Request Data

- request.method
  - Tells you which HTTP method was used to access this view
- request.GET
  - A dictionary of key-value pairs from query parameters (or URL parameter)
- request.POST
  - A dictionary of key-value pairs from POST requests
- request.headers
  - The HTTP headers of the request
- Exercise
  - Do Question 7 on Quercus

Sanitization and Validation

- Sanitization
  - Modifies input to ensure it is syntactically valid
  - E.g., escape characters that are dangerous to HTML
- Validation
  - Checks if input meets a set of criteria
  - E.g., Check that passwords match and username is not blank
- Should be checked at frontend for faster error feedback
- Should **always** be checked at backend as well
  - User can bypass front-end restrictions
    - Inspect element and submit form with erroneous input data
    - Handcrafted HTTP request

Processing POST request

- Validation error
  - If data is invalid, should return a 400-level error code
    - 400: Bad Request
    - 401: Unauthorized
    - 404: Not Found
    - 405: Method Not Allowed
- On success, a redirect is usually returned
  - E.g., redirect to profile page or index page after logging in
- Use HttpResponseRedirect or redirect (from django.shortcuts)
  - E.g., redirect('/some/url’)
  - Is there a problem with the above example?

Named URL Patterns

- Django separates URLs that users see from the URLs developers use
- Developers should use named URLs instead of user URLs
  - User URLs may change, causing your redirects to break
- Add name or namespace argument to the path object
- project's urls.py:

```
path('accounts/', include('testapp.urls', namespace='accounts'))
```

- accounts' urls.py:

```
app_name='accounts'
urlpatterns = [ path('', hello, name='hello'), ]
```

- To redirect: `reverse('accounts:hello')`

Django Template Language

- Adds imperative programming features to making HTML files
  - Similar in spirit to PHP, running PHP code inside <?php … *code* … ?>
- https://docs.djangoproject.com/en/4.1/topics/templates/
- Variables
  - Surrounded by {{ and }}, like this:

```
<p>Hello, {{ username }}.</p>
```

- Tags
  - Provides arbitrary logic in the rendering process
  - Surrounded by {% and %}, like this:

```
{% if has_error %} <p class="error">Bad!</p>{% endif %}
```

Template Response

- Create a `templates` folder in the app's folder
- Convention: create subfolder with app name and put html files inside
  - E.g., the template path would be '<app_name>/hello.html'
- Use the `render` shortcut function to easily use Django templates
- E.g., in accounts/views.py:

```
from django.shortcuts import render
def signup(request):
    error = None
    code = 200      # success
    …
    return render(request, 'accounts/hello.html', {
        "error" : error,
        "username" : username,
    }, status=code)
```

template path

Passing template arguments

Cross-site Request Forgery

- Unauthorized commands from trusted users
  - Can be transmitted by maliciously crafted forms, images, and JavaScript
  - Can work without the user's knowledge
  - E.g., hacking a user's browser to secretly deplete his/her bank fund
- Prevention
  - Using CSRF token
    - Add this to your form `{% csrf_token %}`
    - It will generate the following:

```
<input type="hidden" name="csrfmiddlewaretoken" value="KbyUmhTLMpYj7CD2di7JKP1P3qmLlkPt">
```

    - Token value is unique each time the web page is generated
    - Attack becomes unable to authenticate the request without knowing the token

Static Files

- Django can manage static files, e.g., images, CSS, JavaScript.
  - It simplifies the task of locating the files and serving them
- To use a static file, create a folder named `static` (recommended)
  - Put your static files in here, or its subfolders
- Add this to settings.py:

```
STATICFILES_DIRS = [ BASE_DIR / "static", ]
```

- In the HTML file, you can specify a static file like this:

```
{% load static %}    ← Load this template tag
<!DOCTYPE html>
…
<img src="{% static 'me.jpg' %}" alt="me">
```

Static Files

- Django development can serve static files
  - For testing only. Not suitable for production use.
- Add the URLs of the static file to `urlpatterns` in urls.py

```
from django.conf.urls.static import static
from django.conf import settings

urlpatterns = [
    path('accounts/', include('testapp.urls’, namespace='accounts')),
    …
] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

- Exercise
  - Do Question 8 on Quercus

Before you go

- Assignment 1
  - Extension added
  - Due Monday January 30th
  - MarkUs Autotester is now available
    - Your mark will be the same as the autotester output

- Project
  - Students without groups will be assigned one on January 24th
  - Sign up on Calendly to book an interview session for project grading
    - https://calendly.com/csc309-2023s
  - Phase 1 due date is Feb 5th