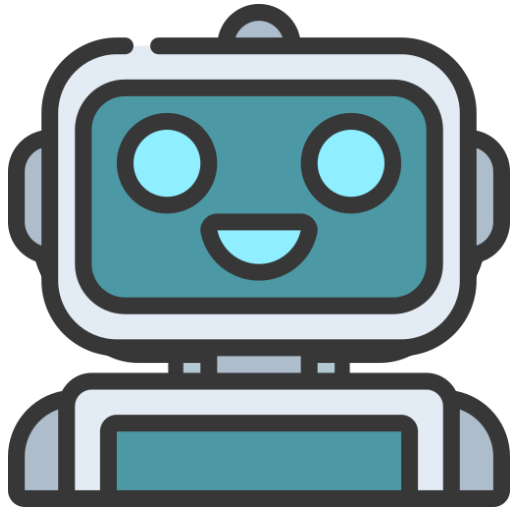
Programming meets the Web:

Internet Robots

Luigi Libero Lucio Starace, Ph.D.

https://luistar.github.io

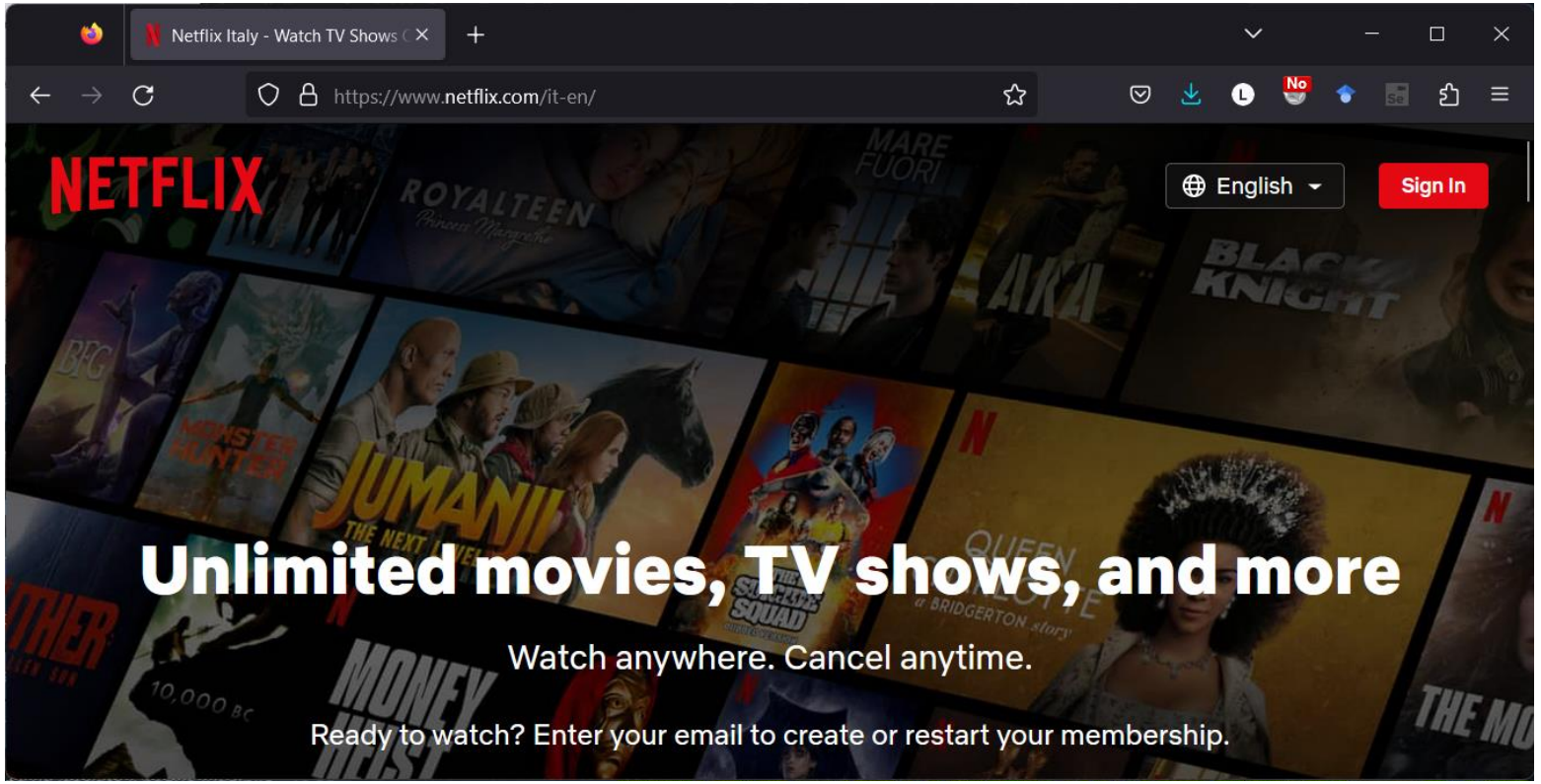
luigiliberolucio.starace@unina.it



May 24, 2023 – Web Technologies

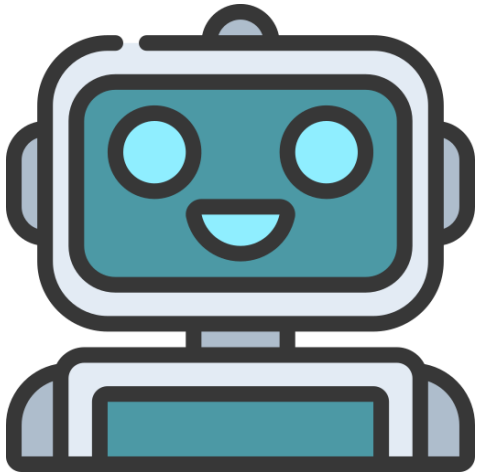
Interacting with the Web

Everyday, we **interact** with the WWW and with Web Apps



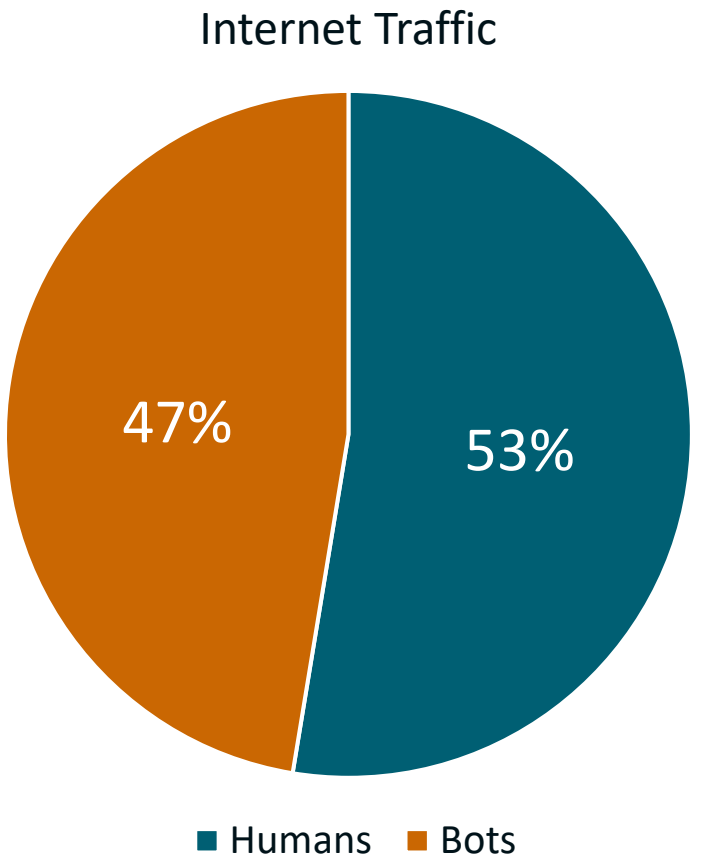
Automatically Interacting with the Web

In a number of cases, it might be useful to **automate** these interactions



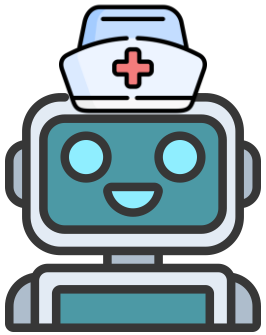
Internet Bots

- Softwares that run automated tasks over the Internet
- More effective than humans, and way more scalable
- In 2022, almost half of the global internet traffic came from Bots [1]

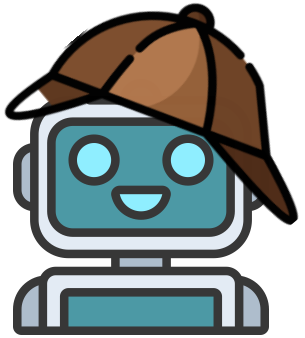


[1] https://www.imperva.com/resources/resource-library/reports/2023-imperva-bad-bot-report

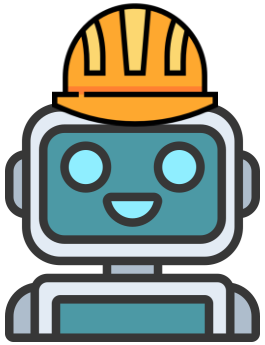
What are the Bots up to? Good Bots



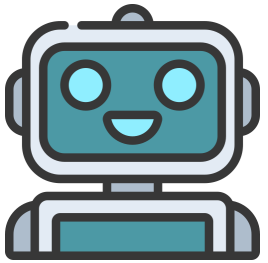
Status Monitoring



Web Indexing

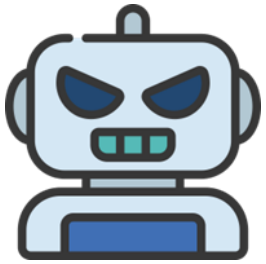


Feed Fetchers

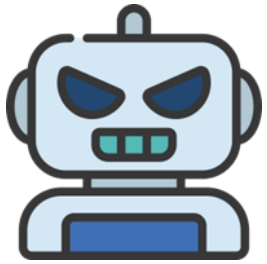


Commercial Bots

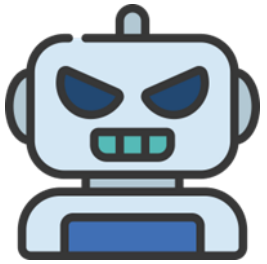
What are the Bots up to? Bad Bots



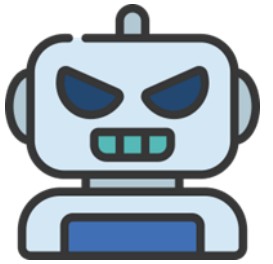
Illegal Scrapers



Spammers



Vulnerability Scanners



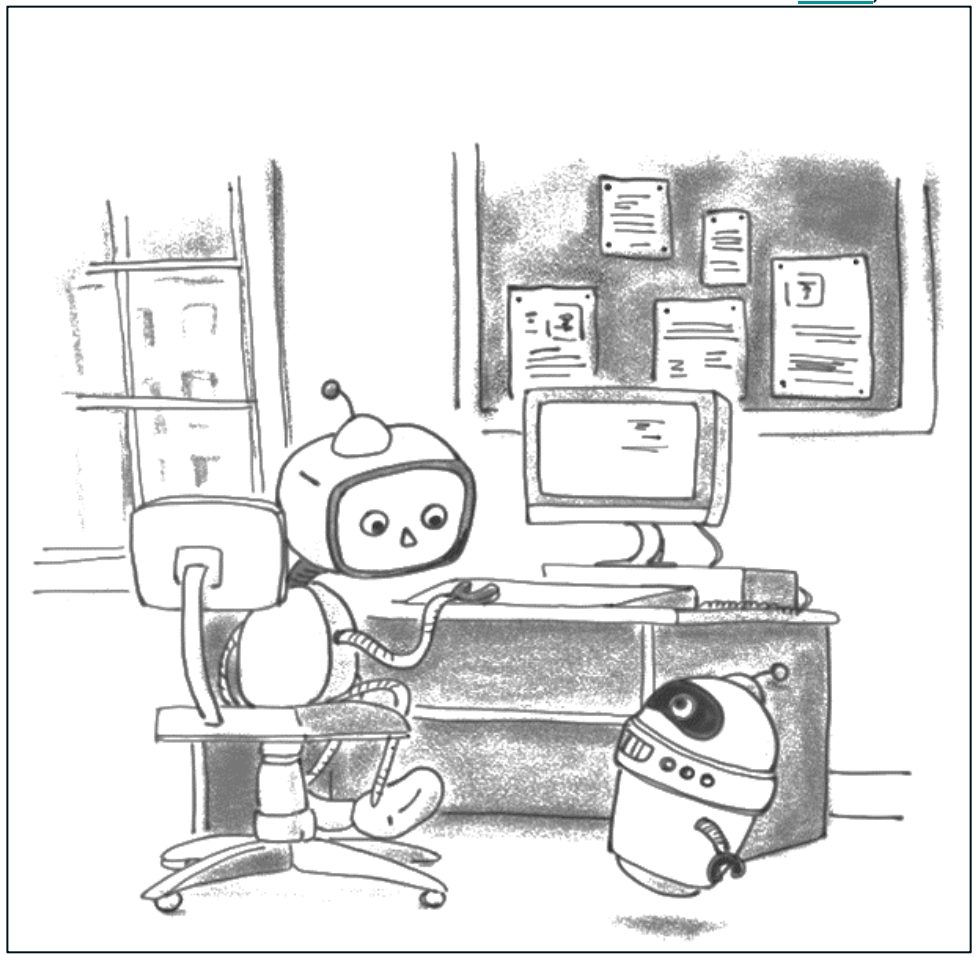
Impersonators

What are the Bots up to? Social Bots

P. Steiner on The New Yorker, 1993.

On the Internet, nobody knows you're a dog.

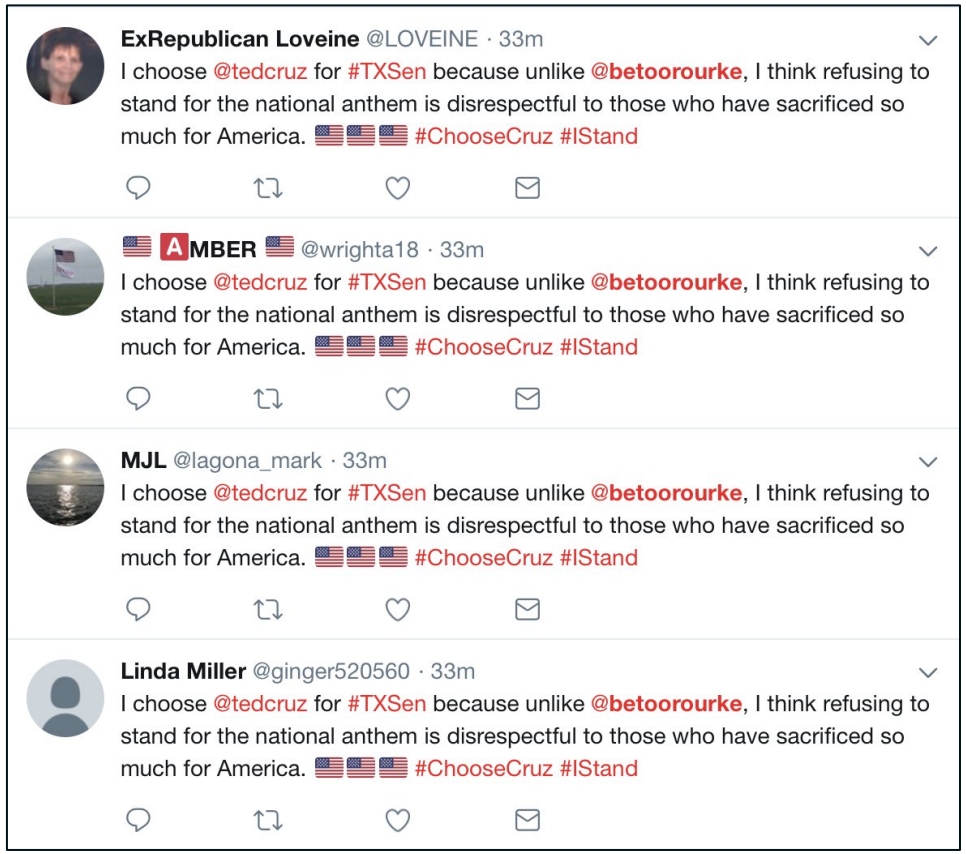
Make Bots Identifiable. ia.net, 2018.



On the Internet, nobody knows you're a bot.

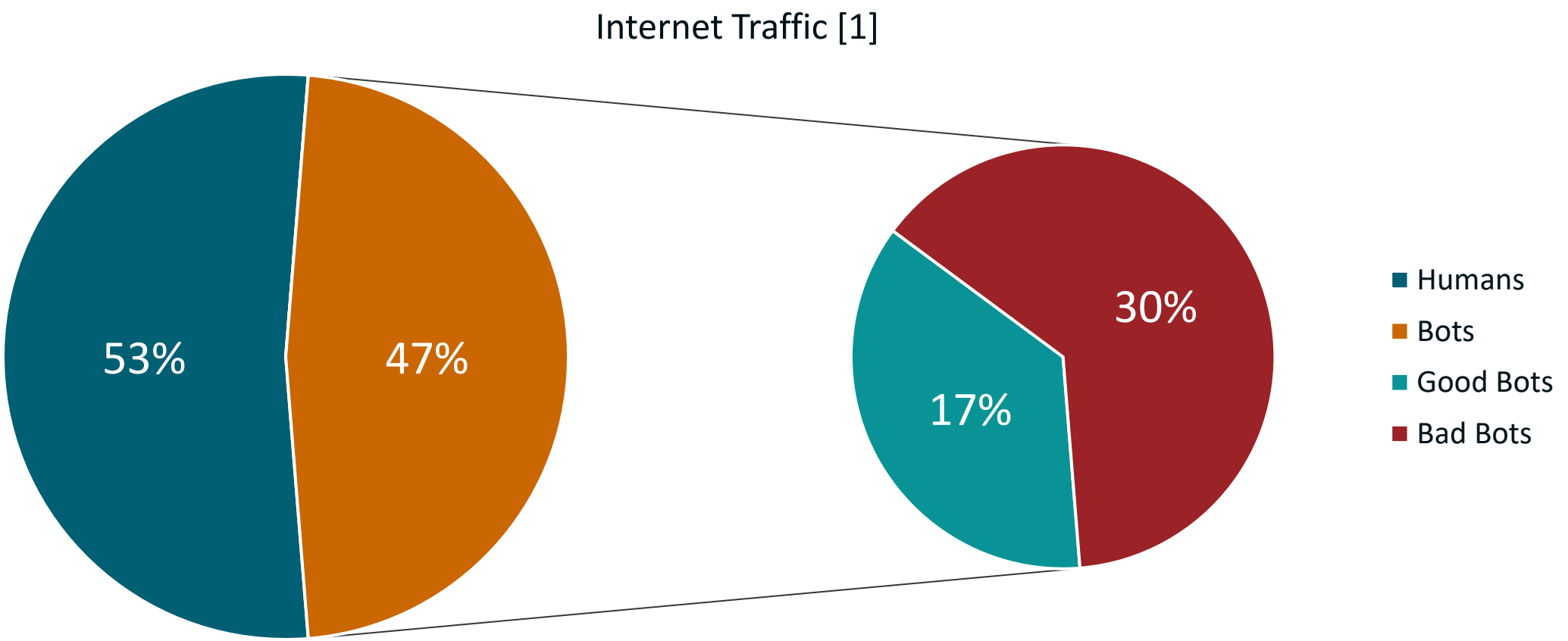
What are the Bots up to? Social bots

- Bots are also laregly used to **impersonate humans** on social networks
- Sometimes for good/acceptable things (i.e., customer care), but mostly to influence public opinion
- This is also a part of how modern wars are fought [1]



[1] Geissler, D., Bär, D., Pröllochs, N., & Feuerriegel, S. (2022). Russian propaganda on social media during the 2022 invasion of Ukraine. https://arxiv.org/pdf/2211.04154.pdf

What are the Bots up to? Good vs Bad Bots



[1] https://www.imperva.com/resources/resource-library/reports/2023-imperva-bad-bot-report

Handling Bots: The Bot Exclusion Standard

- Allows to express rules for Bots that interact with a web site
- `robots.txt` file placed in the root of the hierarchy
- E.g.: https://en.wikipedia.org/robots.txt
- Became a standard in 2022: RFC 9399
- Not a security mechanism! Relies entirely on **voluntary compliance**

```
User-agent: GoodBot
Disallow:

User-agent: BadBot
Disallow: /

User-agent: OtherBot
Disallow: /posts/
Disallow: /articles/
Disallow: /file.html
```

Handling Bots: The Bot Exclusion Standard

- Bot exclusion directives might also be specified using dedicated `<meta>` tags [1] in html documents `<head>`

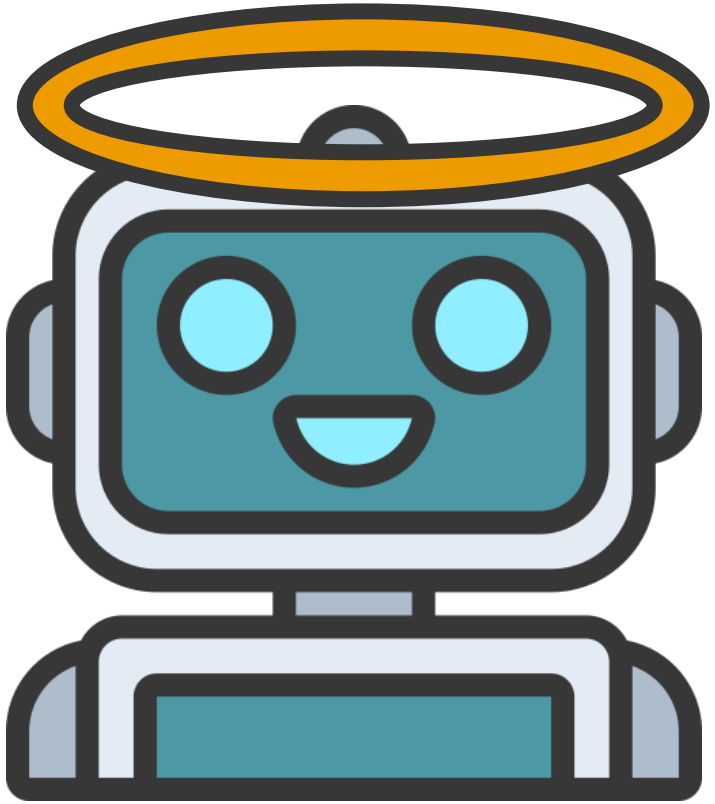
```
<meta name="badbot" content="noindex"/>
<meta name="googlebot-news" content="nosnippet"/>
```

- The indexing of non-html documents (e.g.: pdf files, video, or images) can be blocked using the **X-Robots-Tag: noindex** response header.
- These mechanisms still rely entirely on voluntary compliance as well

[1] https://developers.google.com/search/docs/crawling-indexing/robots-meta-tag

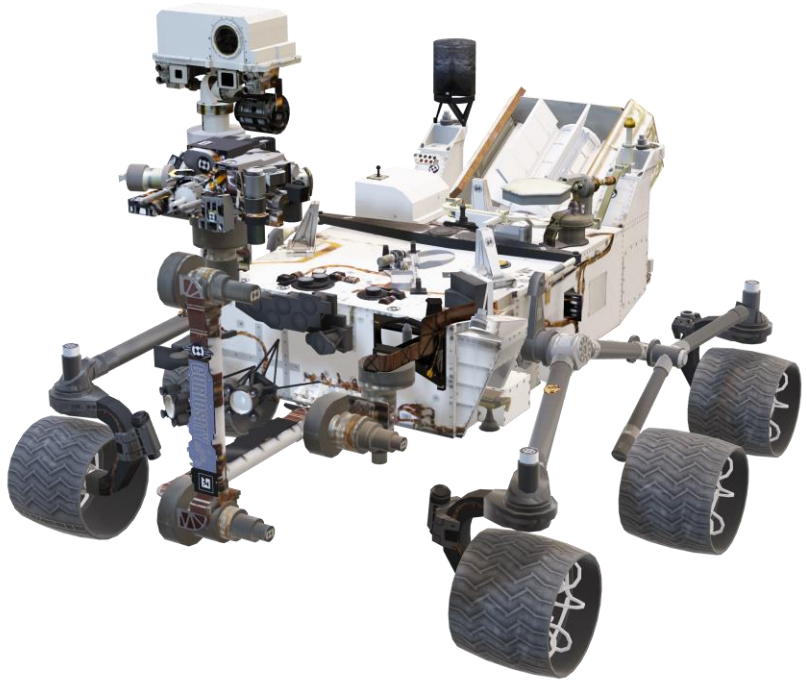
Properties of a Good Bot

- It must comply with national and international (*and Asimov's!*) laws
- It must comply with the Terms and Conditions of websites it interacts with
- It must respect the policies specified in **robots.txt** files
- It should be **polite**
  - Should not put too much load on servers by performing large numbers of concurrent requests



Programming an Internet Bot

- An Internet Bot is a program that interacts with the Internet
- They have their own logic, and perform Web requests (typically http)
- We'll focus on two classes of bots: **Crawlers** and **Scrapers**
- Any other bot is quite similar, with slight changes in the logic



Crawlers

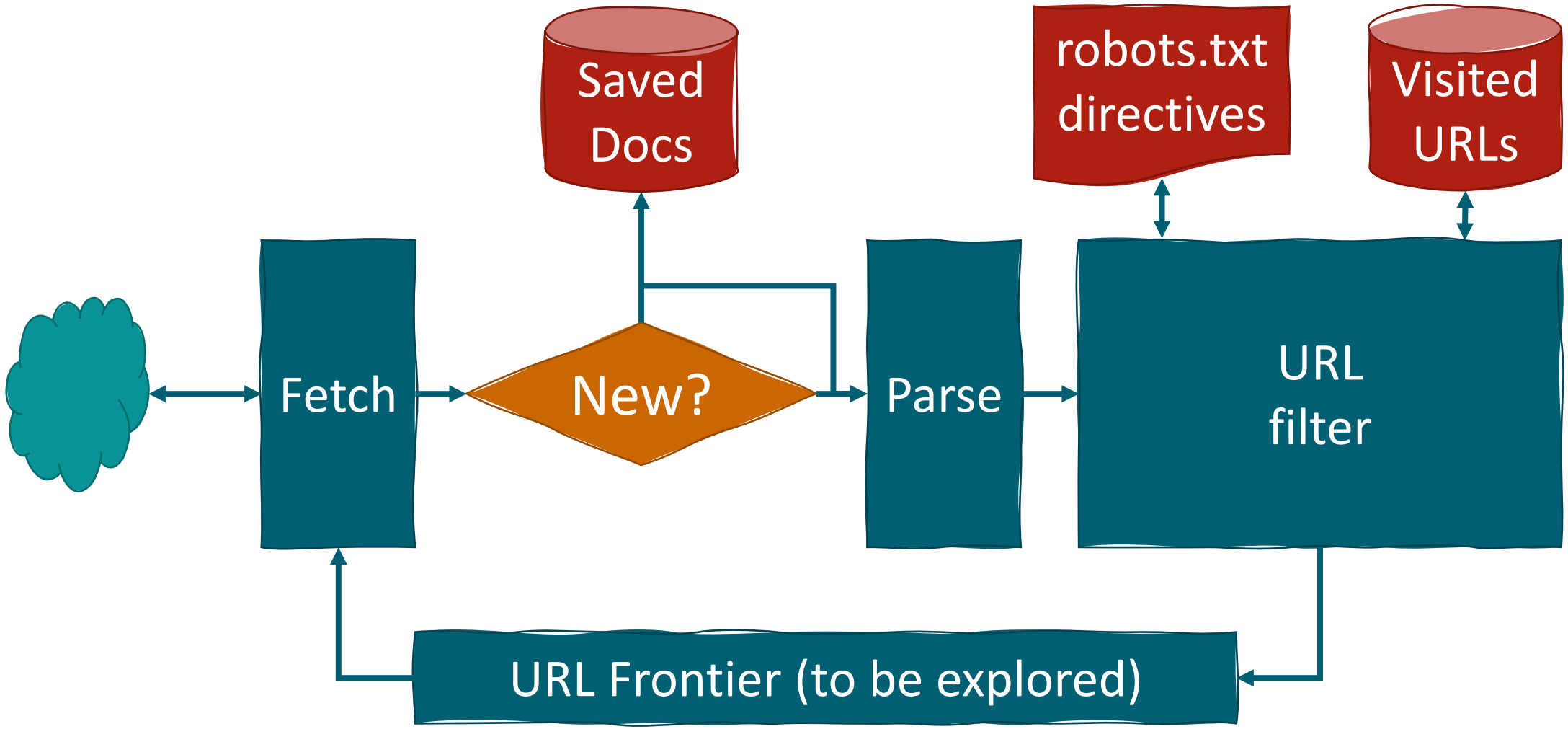
Web Crawling

- **Systematically** explore the web, typically to **retrieve** web pages and add their representations to a local repository
- Bots performing this task are called **Crawlers** (aka **Spiders**)
- Used by search engines to build their indexes
- Used to search for phishing or «dangerous» websites
- Used to infer models of a particular part of the web
- Typically **general-purpose**, work on many websites

Web Crawling – Basic Operation

1. Begin with a set of known seed URLs
2. Add them to a queue
3. Fetch an URL from the queue
4. Get the web page corresponding to the URL
5. Extract URLs that page points to
6. Add the extracted URLs to the queue
7. Return to 3 unless the queue is empty

The Architecture of a Basic Web Crawler



Architecture of a Web Crawler

- Large-scale crawlers are not that simple and require great deals of engineering
- They must be **distributed** on multiple servers
- Typically leverage customized **DNS caching**, **pre-fetching** and resolution to reduce lookup times
- They may never halt

Image generated using DALL-E

Crawling Challenges

- Is the current page «new»?
- Spider **traps**
  - Sets of pages that voluntarily or involuntarily «trap» crawlers
  - E.g.: calendars with dynamic pages
  - Crawlers typically allocate a fixed time budget to crawl a given website

Image generated using DALL-E

Crawling Demo

Implementing a Basic Crawler with Python

Technologies

- Crawling frameworks help us streamline the development of crawlers
- Take care of all the common and boring stuff…
  - Dealing with robots.txt
  - Dealing with parallelism
  - Parsing web pages
- … so you don’t need to reinvent the wheel
  Examples include: Scrapy, Apache Nutch
- Let’s build a crawler with Scrapy

Crawling Demo

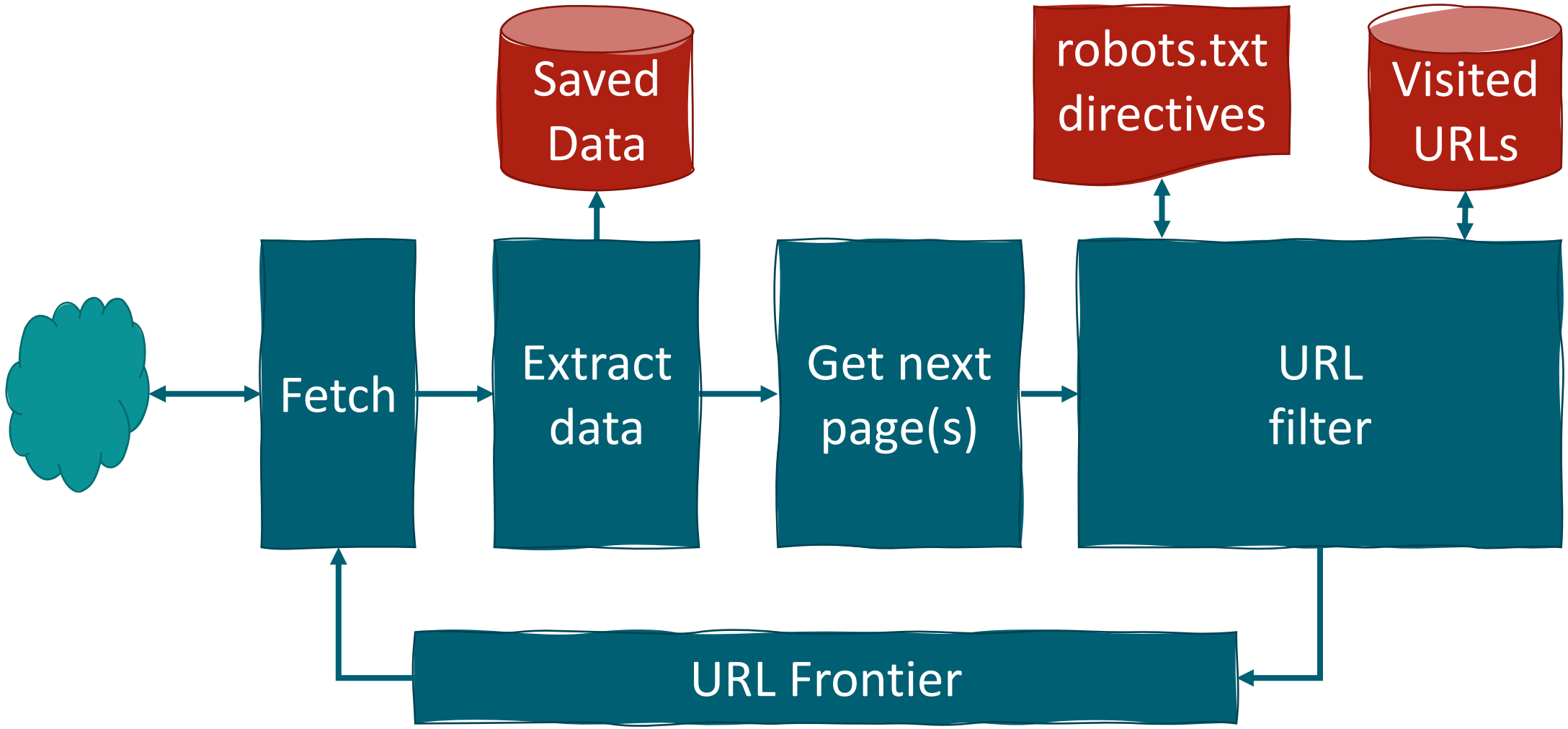
Implementing a Basic Crawler with Python and Scrapy

Scrapers

Web Scraping

- Scraping aims at **extracting data** from human-readable outputs of other programs
- In a Web context, extract data from HTML pages, or APIs
- Extracted data is saved in a local file or database for futher processing
- Bots that specialize in this task are called **Scrapers**
- Scrapers are typically site-specific, not general purpose

The Architecture of a Basic Web Scraper



Scraping Demo

Scraping with Python and Scrapy

Demo: Source Code

- The source code we used in the demos is available on GitHub at https://github.com/luistar/crawling-scraping-examples
- Includes:
  - Crawler «from scratch»
  - Scraper «from scratch»
  - Crawler and Scraper implemented with Scrapy
- Please, don’t DDoS GitHub Pages and my personal website :P

Limitations of the «traditional» bot approach

- We’ve implemented crawlers and scrapers that
  1. Download HTML files (GET http request)
  2. Parse them to extract data and find next pages to visit
  3. Return to Step 1
- This «traditional» approach is simple and efficient
- Has served us well since the beginning of the Internet
- Unfortunately, it doesn’t always work nowadays!


Scraping Dynamic Web Pages Demo

Code: Dynamic Web Pages

- The Dynamic Web Pages shown in the demo are available on GitHub at https://github.com/luistar/dynamic-web-pages-examples
- The repository includes:
  - Web Page featuring AJAX requests (implemented both with XMLHttpRequest and the more modern fetch API)
  - A single-page React app performing an AJAX request
- These dynamic pages are also hosted at https://tecweb-2023-scraping-examples.tiiny.site/, in case you don't want to setup a local web server.

Limitations of the «traditional» approach

- Modern web pages are more and more **dynamic**
- Dynamically **fetch** and update data using **AJAX**
- Depend on **client-side rendering frameworks** such as **React**, **Vue.js**, …
- Getting an HTML document might not be enough

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>A Single Page React App to Scrape</title>
    <script defer src="/static/js/bundle.js"></script></head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
  </body>
</html>
```

Interacting with highly dynamic web pages

- To automatically interact with dynamic web pages we would need to:
  - Fetch all the linked resources (scripts, stylesheets, images, etc…)
  - Actually render the web page (some Javascript code might depend on the rendering (e.g.: `element.getBoundingClientRect`)
  - Simulate user events (e.g.: clicks, scrolls, …)
  - Interpret Javascript code as a browser would
- Basically we would need to re-invent a modern web browser…
- … so, why don’t we just use one?

WebDriver

- W3C recommendation
- A **remote control interface** that enables introspection and control of user agents
- Implemented by modern web browsers (Firefox, Chrom*, etc…)
- Supported by libraries such as Selenium, Playwright, Puppeteer, …

Selenium

- Browser Automation Library: link
- Open-source
- Officially supports: Java, Python, C#, JS, Ruby
- Supports all major browsers: Firefox, Chrome, Edge, Opera, Safari

Getting Started with Selenium

1. Install a Selenium library (examples in Maven/Java)

```
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>4.8.0</version>
</dependency>
```

2. Install the required WebDrivers
   - Check out the official docs
   - Tools like WebDriverManager make the whole process painless

```
import io.github.bonigarcia.wdm.WebDriverManager;

WebDriverManager.edgedriver().setup();
```

Your First Selenium-based Program

- Selenium allows devs to programmatically interact with browsers
- The snippet below creates a WebDriver instance (Firefox)
- Opens Firefox in marionette mode, loads BASE_URL and prints its title

```
public static void main(String[] args){
    final String BASE_URL = "http://localhost:3000/";

    WebDriverManager.firefoxdriver().setup(); //setup Firefox driver
    WebDriver driver = new FirefoxDriver(); //create a Firefox WebDriver
    driver.get(BASE_URL); //load the base url
    System.out.println(driver.getTitle()); //print title of the web page
}
```

Selecting Elements

- Loading a web page and inspecting its title is cool
- … but we won't go far without selecting elements!
- In Selenium, we can do this by using

```
WebDriver driver = new EdgeDriver();
driver.get("http://mypage.org/");

WebElement element          = driver.findElement(selector);
List<WebElement> elements = driver.findElements(selector);
```

- Different **locators** are provided by the By class

Selenium Locators

Locators provided by Selenium include:

Locator	Description
`By.id(s)`	Locates elements that have the given id
`By.name(s)`	Locates elements whose «name» attribute has the given value
`By.className(s)`	Locates elements having the given class name
`By.linkText(s)`	Locates link elements whose **visible text matches** the given string
`By.partialLinkText(s)`	Locates link elements whose **visible text contains** the given substring
`By.js(script,…args)`	Locates elements by evaluating a given JavaScript expression
`By.cssSelector(s)`	Locates elements using a **CSS** selector
`By.xpath(s)`	Locates elements using a **XPath** selector

Locating Elements (examples)

How can we locate:

- The username input field?
  - `By.name("usr")`
- The forgot password link?
  - `By.linkText("password?")`
- The Login Button?
  - `By.id("log")`

```
<html>
  <head><title>Login</title></head>
  <body>
    <form action="action_page.php" method="post">
      <div class="container">
        <label for="uname">Username</label>
        <input type="text" name="usr">
        <label for="psw">Password</label>
        <input type="password" name="psw">
        <button type="submit" id="log">Login</button>
      </div>
      <div class="container">
        <button type="button">Cancel</button>
        <span>Forgot <a href="#">password?</a></span>
      </div>
    </form>
  </body>
</html>
```

CSS Locators (in a Nutshell)

- Here's a brief overview of CSS selectors, by example
- For a complete reference, check out W3CSchools
- They have a nice CSS Selector Tester as well

Selector	Example	Example Description
element	h1	Selects all h1 elements
.class	.btn	Selects all elements with class="btn"
#id	#login	Selects the element with id="login"
element.class	h1.title	Selects all h1 elements with class="title"
element.class1.class2	a.btn.err	Selects all a elements with both btn and err in its class
selector1, selector2	h1, h2	Selects all h1 elements and all h2 elements

CSS Locators (in a Nutshell)

Selector	Example	Example Description
`[attribute]`	`[required]`	Selects all elements with a required attribute
`[attribute=value]`	`[href=#]`	Selects all elements with href=#
`[attribute^=value]`	`[href^=ftp]`	Selects the elements whose href value starts with ftp
`[attribute$=value]`	`[href$=pdf]`	Selects the elements whose href value ends with ftp
`[attribute*=value]`	`[title*=hi]`	Selects the elements whose title value contains hi
`element[attribute]`	`a[target]`	Selects all a elements with a target attribute
`selector1 selector2`	`div h1`	Selects all h1 elements inside div elements
`selector1>selector2`	`div>h1`	Selects all h1 elements whose parent is a div

XPath Locators (in a Nutshell)

- **XPath** stands for **XML Path Language**
- Part of the **XSLT** standard and W3C recommendation
- Designed to navigate through elements and attributes in XML docs
- Uses **path expressions** to select nodes or node-sets

XPath Locators (in a Nutshell)

Selector	Example	Example Description
element	h1	Selects all h1 elements
/	/body	Selects from the (current) root node
//element	//p	Selects all p elements
//element1/element2	//p/a	Selects the a elements that are children of a p
/elem[n]	//p/a[2]	Selects the second a element that is the child of a p elem.
element1//element2	p//a	Selects the a elements that are descendants of a p
elem[@attr]	a[@target]	Selects the a elements having a target attribute
elem[@attr="val"]	a[@class="btn"]	Selects all a elements with class="btn"
elem[text()="val"]	a[text()="hi"]	Selects all a elements whose inner visible text is "hi"
a[contains(@class,'str')]		Selects all a elements whose class attr. contains str

XPath Locators (in a Nutshell)

```
<!DOCTYPE html>
<html lang="en">
<head><title>E2E Web Testing</title></head>
<body>
  <div class="main-content">
    <h1>E2E Web Testing</h1>
    <p>This page is about GUI-level E2E Testing of Web Applications.</p>
    <p>E2E testing <span class="hl">improves</span> the <a href="/qual">quality</a> of Web Apps.</p>
    <a href="/more" class="btn" id="main">Learn more</a>
  </div>
</body>
</html>
```

Absolute (or Raw) XPath locators start from the root node and traverse the entire hierarchy of the page

- E.g., Select the «quality» link: `/html/body/div/p[2]/a`

XPath Locators (in a Nutshell)

```
<!DOCTYPE html>
<html lang="en">
<head><title>E2E Web Testing</title></head>
<body>
  <div class="main-content">
    <h1>E2E Web Testing</h1>
    <p>This page is about GUI-level E2E Testing of Web Applications.</p>
    <p>E2E testing <span class="hl">improves</span> the <a href="/qual">quality</a> of Web Apps.</p>
    <a href="/more" class="btn" id="main">Learn more</a>
  </div>
</body>
</html>
```

Select the «Learn more» button:

- `//a[@id="main"]`
- `//a[text()="Learn more"]`
- `//div[@class="main-content"]/a[@class="btn"]`

Your First Selenium-based Scraper

- Let's scrape the single-page React app!

```
final String BASE_URL = "http://localhost:3000/";

WebDriverManager.firefoxdriver().setup(); //setup Firefox driver
WebDriver driver = new FirefoxDriver(); //create a Firefox WebDriver
driver.get(BASE_URL); //load the base url

// locate the button we need to interact with...
WebElement updateButton = driver.findElement(By.id("update-button"));
updateButton.click(); //... and send a click event to it
// get the articles
List<WebElement> articles = driver.findElements(By.cssSelector("article.article"));
articles.forEach( (article) -> {
    String title = article.findElement(By.cssSelector("h3")).getText();
    String author = article.findElement(By.cssSelector("span.author")).getText();
    String year = article.findElement(By.cssSelector("span.year")).getText();
    System.out.println(title + " by " + author + ", " + year);
});
```

Exercises

1. Implement a basic crawler using Selenium (same functionality of the basic crawler example we discussed).
2. Implement a basic scraper using Selenium (same functionality of the basic scraper example we discussed).
3. Implement a scraper for all the provided dynamic web pages.
   A public mirror is also available at:
   https://tecweb-2023-scraping-examples.tiiny.site/