Web Development and API Design

Lesson 06:
Async Calls to Web Services

Prof. Andrea Arcuri

Goals

- Understand how a SPA can communicate with a backend using **AJAX**, retrieving data in JSON format
- Introduction to the "*event-loop*" model in JS, and how **async/await** can be used to simplify the code dealing with asynchronous behavior
  - e.g., calls to a remote web service, like *REST* and *GraphQL*

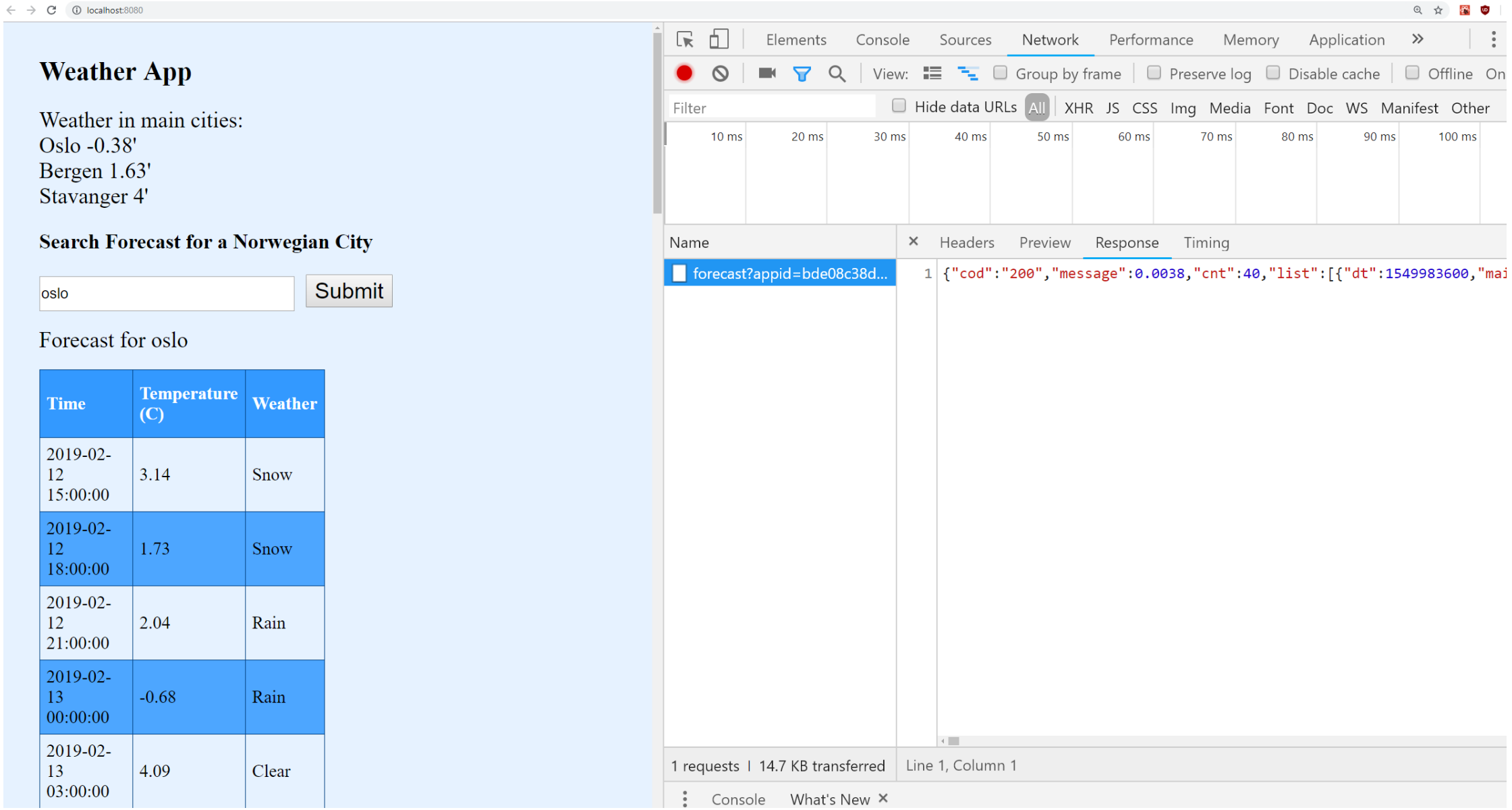
Browser-Server Communications

- Usually based on HTTP over TCP
- Address bar in browser: download that resource, e.g., typically starting from *index.html*
- Then download all other resources used in that HTML file
  - eg, CSS, images and JS files
- User interactions with server: clicking on **<a>** links and submitting **<form>**
  - after such actions, usually would get a new HTML page back from server

AJAX (Asynchronous JavaScript and XML)

- Ability for JS code to start HTTP communications to server
- XML in the name is just for historical reasons… nowadays the main data format is **JSON** (JavaScript Object Notation)
- A SPA can use AJAX to retrieve the data it needs (in JSON), without getting whole new HTML pages
- Once getting JSON data, update HTML in the browser (eg with React)

Example: just fetch data of forecast in JSON, and not a whole HTML page displaying it



Using AJAX

- For JS in the browser, there are 2 main ways to do HTTP calls
- **XMLHttpRequest**: *old* approach using **Callbacks**
  - same as AJAX, the XML in the name is only for historical reasons… you can use it to send/receive any kind of data besides XML, eg JSON
- **fetch()**: more *modern* approach using **Promises**

Third-Party Service Rate Limits

- Commercial web services might provide some *free* options to test them out
- Usually need to create an account
- At each HTTP call, need to provide a **key** to verify it is indeed you that made the request
- If you make too many requests, can be blocked, and asked to buy a commercial license

OpenWeatherMap API usage alert

Inbox ×

OWM Team <robot2@openweathermap.org> Unsubscribe

18 Apr 2019, 22:30

to me

OpenWeatherMap.org

Your API key is blocked.

Dear Customer,
Your OpenWeatherMap API key bde08c38dcc24dfbffc449466cea7e44 **is temporary blocked** due to the continual sufficient exceeding of the calls per minute limit by performing **1986** requests within a minute but the limit for the **Free** account is **60** rpm.

To unblock your API key and restore the normal operation with the weather API we ask you to surpress the loading from your API keys or upgrade your subscription. Contact us info@openweathermap.org for any help with choosing the proper type of the subscription.

Issues with AJAX

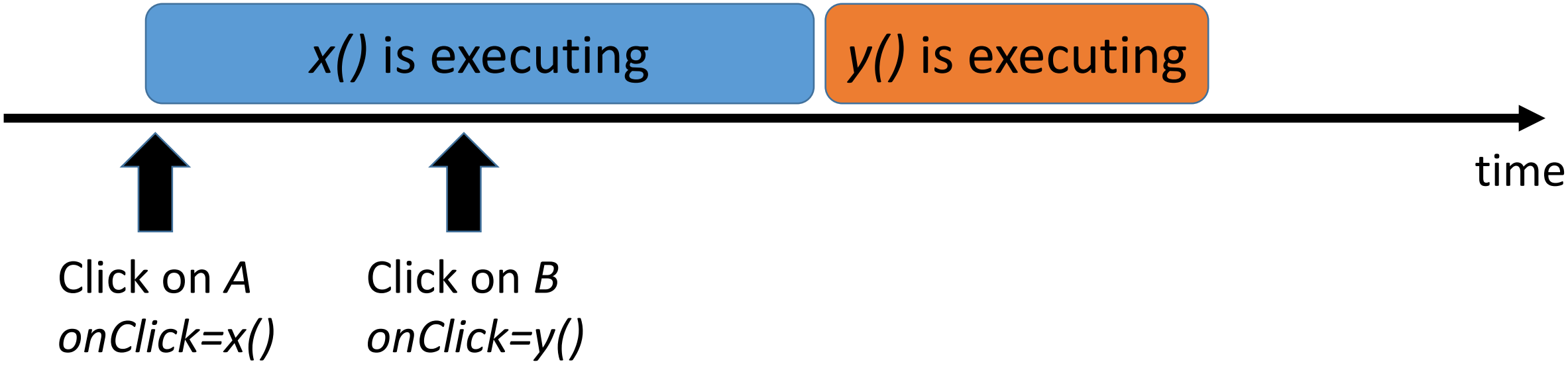
- You make an HTTP call over TCP with AJAX
- Such call could take few milliseconds, or seconds, BEFORE you get a reply from the server
- Even if just 1 ms, might need to do many HTTP calls to render current page (eg fetch data from different servers)
- You do NOT want your app to **freeze** and be unresponsive till server replies
- This is a problem due to how threading is handled in JS

JS Event-Loop Thread

- Following is a very *high-level*, *simplified* story of how the *event-loop* works in JS
  - eg, not going to discuss things like *Service Workers* or the *Job Queue*
- For what concerns you, your JS code is going to be executed on a **single thread**
- Your functions will **run to completion**
  - These are the functions executed when intercepting events like *onClick* and *onMouseOver*

Run To Completion

- Assume a user clicks on 2 buttons (*A* and *B*), executing *x()* and then *y()*, which are registered as *onClick* handlers
- As long as *x()* is running, *y()* cannot start, as there is only 1 thread executing your code for the event handlers



Run To Completion Problems

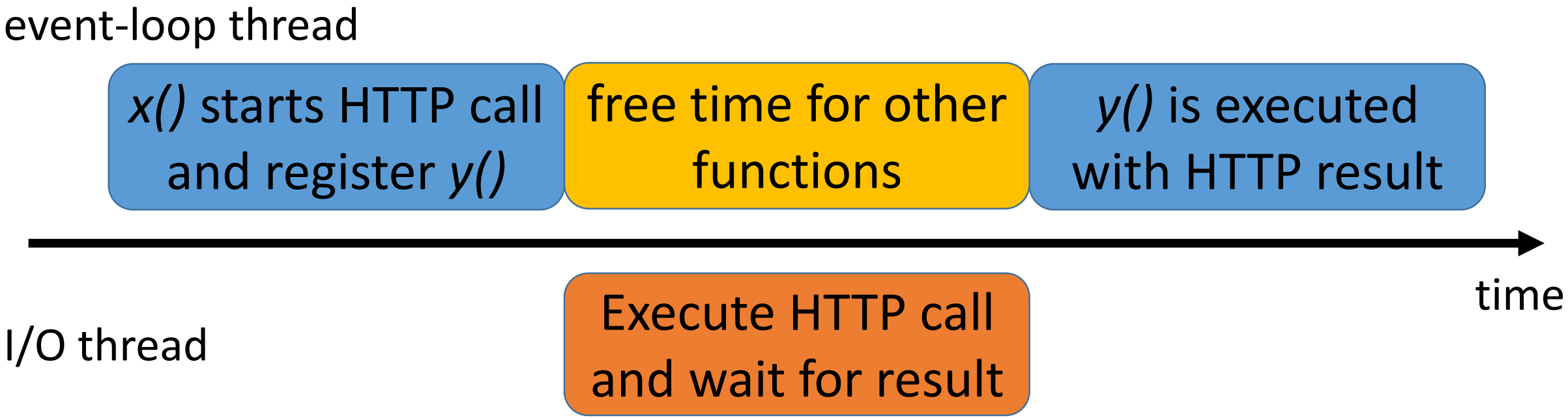
- **while(true){}**
- Code above could completely freeze your app, as no other code could run, as that is an infinite loop and will never end
  - note, you can end up in infinite loops due to bugs…
- Expensive CPU computations in JS can slow down the responsiveness of your app, making it feeling sluggish

AJAX and Run to Completion

- AJAX: (1) execute a HTTP Request; (2) do something when you get the HTTP Response
- Might take many ms before getting back the response
- **Cannot wait on event-loop thread for the response**, otherwise the app would freeze in that period of time
  - i.e., no other code could be executed meanwhile
- 2 solutions: **Callbacks** and **async/await** on **Promises**

Callback

- AJAX call in function *x()* will register a callback function *y()* which will be executed on the event-loop thread when getting results from server
- The HTTP call will be made by an I/O thread, which will schedule *y()*



```
const ajax = new XMLHttpRequest();

//register the "callback" to handle the server's response
ajax.onreadystatechange = () => {
    const payload = JSON.parse(ajax.response);
    //do something with response
};

ajax.open("GET", url);
ajax.send();
```

- Here, the **onreadystatechange** is the **callback** that is going to be executed once we get back the result
- Note: such callback has to be registered BEFORE we **send()** the HTTP request, but will be executed AFTER

Callback Issues

- Callbacks are fine when you make a single request
- When you have many asynchronous communications, each one depending on the others, it can get *very difficult* to see what is going on and the order in which functions are executed
- Often called **Callback Hell**

Promise

- A **Promise** is a JavaScript object
- "*The Promise object represents the eventual completion (or failure) of an asynchronous operation, and its resulting value*"
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise
- A Promise will eventually return the value of the asynchronous computation, and we can **await** until such value is available
- The **fetch()** method making an AJAX request does return a Promise

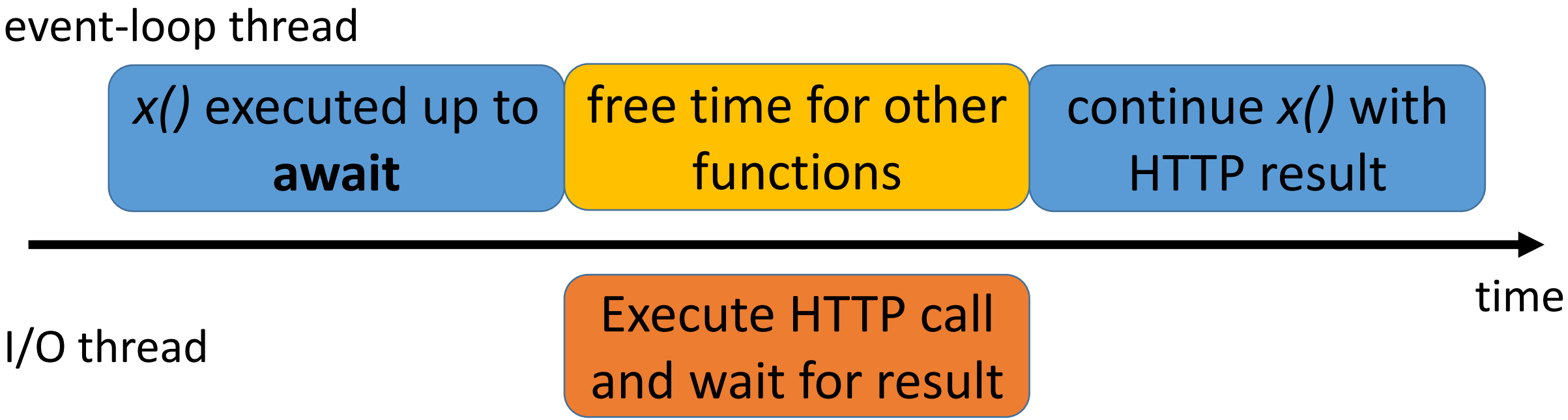
```
async doHttpFetch(url) {

    response = await fetch(url);
    payload = response.json();
    //do something with response
```

- Here there is no callback… we execute operation (*fetch*) and **await** for the results
- Then we continue with the rest of the function
- But what about "**Run to Completion**" in JS???
  - note: the function is declared as **async**

async/await

- **async** functions are split in *execution blocks*, around the **await** commands
  - note, there can be many **await**s inside the same **async** function
- The event-loop thread will execute the functions blocks, and not the whole function
- When I/O thread will get the HTTP response, it will schedule the execution of the second block after the **await**
- The *event-loop thread* is **NOT** waiting during an **await**



Benefits of **async/await**

- It makes code much easier to read, as now the flow of execution looks *sequential*
  - this is particularly true when you have many asynchronous operations in the same function
- No major performance drawback: the event-loop thread is not waiting, and can execute other commands meanwhile we wait for I/O
- Recall that *thread waiting* and *thread-context switches* are **expensive**, because OS operations

Non-Blocking I/O

- This model of a single event-loop thread running your code in blocks is often referred as **Non-Blocking I/O**
- Such model was popularized by *NodeJS*
  - however, most other languages can do the same, e.g., Java, Kotlin and C#
- Very good for CRUD web applications:
  - most operations are CPU cheap, where bottlenecks are in I/O on database
  - can serve many different users without thread-context switches
- However, it is bad for CPU-bound applications
  - as you only have a single execution thread…
  - you could though replicate your app in many running instances, behind a load-balanced gateway (but this is not something we will see in this course…)

Creating a Promise

- For this course, we deal with **Promise**s mainly when we **await** on **fetch()** calls
- But we can create our own **Promise**s
  - we will need to do it for testing purposes
- A Promise requires as input a function, which itself takes as input two functions:
  - *resolve(someValue)*: we will call it when we want to state the Promise is resolved, ie successfully finished. The value we will be what returned to who is **await**ing on such Promise
  - *reject()*: specify that the Promise has failed
  - note: if "*your code*" in the example below is "*resolve(5)*", then the Promise would resolve immediately, giving the value 5 as output

```
new Promise( (resolve, reject) => {/* your code */});
```