HTTP SERVERS
From last time: URLs

http://attu:8080/cse331/test?a=b&c=d#whatever

- **Port** is optional (default is 80 for HTTP)

- Optional “?a=b&c=d” part of path is called **query string**
  - “&”-separated key=value pairs
  - useful for passing arguments to the server-side code…

- **Fragment** is only kept in the browser
  - client can use this to record its place in the document
  - allows back/forward buttons to work on a single page
Server Frameworks

• How do we write a modular HTTP server?
  – need to split up the code into multiple classes

• Usual technique is to route requests using the path
  – use path to choose class that handles the request
  – used in Java, C++, Python, JavaScript, …
  – pass data to class using:
    • query string
    • POST body
    • (part of) path
Spark Java

• Simple library for writing HTTP servers in Java
  – not to be confused with “Apache Spark” — very different!

• Give Spark paths and corresponding classes
  – latter are called “routes” in this library
  – server will read the request path and invoke appropriate class
    • info about the request passed in request object
    • response can be written to response object or returned

• Library handles the event loop
Spark Java

Spark.get("/path", new MyRoute());

• GET request with this path are sent to this object

• Second argument must implement Route interface
  – single required method handle(Request, Response)
  – that means it can also be implemented with a Lambda

Spark.get("/ready", (request, response) -> {
  return "Nah, I’m busy";
});
Example: Hello Server

HelloServer.java
Example: To-Do Server

- Stores a To-Do list
- Clients can retrieve the current list
- Clients can update the list
  - check off an item
  - add a new item
Example: To-Do Server

ToDoServer.java
Spark Java

• Many more features
  – simple things are simple
  – complex things are possible

• Simple version is single threaded
  – makes life much easier
  – medium scale would use threads
  – high scale would not use them (see last lecture)

• Documentation at http://sparkjava.com/documentation
Example: To-Do Servers

Similar approaches work in other languages
  – none of these ideas are specific to Java

Java  server-java
Python server-py
Node.js server-node
HTTP CLIENTS
Client / Server communication

- Original JavaScript API: `XmlHttpRequest`

- Create object call `open` to configure
  - pass in GET / POST, path, and async = true

- Listen for response event
  - `onload` invoked when done
    - `responseText` contains the response body string

- Call `send` to start the request
  - for a POST, pass in the request body
  - for GET, pass `null`
Example: To-Do Client

client-xmlhttp/src/TodoApp.tsx
Debugging

• Network tab in Chrome shows every request
  – full details of request
    • path, headers, etc.
  – full details of response
    • status code, response body, etc.
  – timing information
Client / Server communication

- Original JavaScript API: XMLHttpRequest

- Improved APIs:
  1. fetch (library)
  2. async / await (language)
Fetch

`fetch(url)` returns a Promise

**Promise object**
- `.then(f)` calls f after request completes
- `.catch(f)` calls f after request fails

```javascript
fetch(“localhost:4567/list”)
  .then((resp) => console.log(resp.status));
  .catch((err) => console.error(err));
```
Promise Chaining

What is the point of **Promises**?
- how is `.then(f)` different from `.onload = f`?

Key feature of the library is the ability to chain promises
- `.then` returns another **Promise**
- can use `.then` on it as well

```javascript
fetch("localhost:4567/list")
  .then((resp) => resp.text());
  .then((text) => console.log(text));
```

- `.then` called once status is known
- `.text` called once body is known
**Fetch: GET vs POST**

fetch can be used to send either GET or POST

`fetch(url)`
- starts a GET request
- pass arguments by including a query string ("?a=b…")

`fetch(url, {method: "POST", body: "..."})`
- starts a POST request
Example: To-Do Client

client-fetch/.../TodoApp.tsx
Await: Compiler Help for Promises

Syntax: `await P`

- where `P` is any expression producing a `Promise`

```javascript
async function foo() {
    ... code A ...
    let v = await P;
    ... code B ...
}
```

- acts as if the code pauses at “await P”
- (but other events can continue being processed)
Await: Compiler Help for Promises

```javascript
async function foo() {
    ... code A ...
    let v = await P;
    ... code B ...
}
```

becomes

```javascript
function foo() {
    ... code A ...
    return P.then((v) => {
        ... code B ...
    });
}
```
Await Example

```javascript
fetch("localhost:4567/list")
  .then((resp) => resp.text());
  .then((text) => console.log(text));
```

can be rewritten as

```javascript
let resp = await fetch("localhost:4567/list");
let text = await resp.text();
console.log(text);
```

Second version is more readable for most people.
Await: Compiler Help for Promises

Syntax: \texttt{await P}

\hspace{1cm} where \( P \) is any expression producing a \texttt{Promise}

```javascript
async function foo() {
    \hspace{1cm} \ldots \text{code A} \ldots
    let v = await P;
    \hspace{1cm} \ldots \text{code B} \ldots
}
```

\hspace{1cm} if \texttt{.then} is invoked, \texttt{await} returns that value

\hspace{1cm} if \texttt{.catch} is invoked, \texttt{await} throws that exception
Await: Compiler Help for Promises

```javascript
async function foo() {
    ... code A ...
    try { await P; }
    catch (err) { ... code B ... }
}
```

becomes

```javascript
function foo() {
    ... code A ...
    return P.catch((err) => {
        ... code B ...
    });
}
```
Async

- Functions that use `await` must be declared `async`
  - they no longer finish synchronously

- Compiler has them now return a `Promise`
  - only performs work up until the first `await`
  - Promise encapsulates the work after that

- You can chain code after them with `await`!
Example: To-Do Client

client-async/.../TodoApp.tsx