6.170: Software Studio
Web APIs

Arvind Satyanarayan +
Daniel Jackson
Monday

- Hypertext Transfer Protocol (HTTP)
  - URLs, IPs, and the DNS
  - Request & Response
- Node.js: Server-Side JavaScript
  - Express: Web Server Framework
  - Routes
  - Templates
  - Middleware
  - Sessions & Cookies

Today

- RESTful API Design
- AJAX
- Asynchrony
  - Continuations & Callbacks
  - Promises
The Year is 2000.
API URLs look like this:

/shopping_cart.asp?action=update_qty&user=123&...

/postComment.jsp?entryID=853&text=...

/services.php?method=bid&item=236&...
API URLs look like this:

/shopping_cart.asp?action=update_qty&user=123&...
/postComment.jsp?entryID=853&text=...
/services.php?method=bid&item=236&...

I stands for interface, but early web APIs were poorly designed:

- **Not easily discoverable:** what goes in the path, what goes in the query parameters?
- **Inconsistent:** APIs could be internally inconsistent. Different APIs might have different path/parameter conventions.
- **Difficult to maintain/extend.**
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 299
SOAPAction: "http://www.w3.org/2003/05/soap-envelope"

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
xmlns:m="http://www.example.org">
  <soap:Header>
    </soap:Header>
  <soap:Body>
    <m:GetStockPrice>
      <m:StockName>GOOG</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
Representational State Transfer (RESTful APIs)
RESTful APIs

"Applying verbs to nouns"
A RESTful Request

GET /teas/green

Verb (HTTP Method) Noun aka Resource (URL)
Nouns are Resources

URLs identify a *representation* of a resource.

/teas/green.html
/teas/green.json
/teas/green.xml

Use path hierarchies to imply *structure*, and differentiate between *collections* and *instances*.

/teas
/teas/green
/teas/green/reviews

/teas/345
/teas/345/reviews/5
Verbs take action on resources

The four basic functions of persistent data: create, read, update, delete (CRUD).

These functions map to HTTP methods.

Why bother? Why not just use POST?
Verbs take action on resources

✓ Methods carry different semantics, and can be applied to the same noun:

GET /teas/123/reviews — Get all reviews for tea #123
POST /teas/123/reviews — Create a new review for tea #123
PUT /teas/123/reviews/4 — Update review #4 for tea #123
DELETE /teas/123/reviews/5 — Delete review #5 for tea #123
Verbs take action on resources

✓ Method semantics make it easier to reason about data safety.

<table>
<thead>
<tr>
<th>Method</th>
<th>Safe</th>
<th>Idempotent</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>POST</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>PUT</td>
<td>✘</td>
<td>✔</td>
</tr>
<tr>
<td>DELETE</td>
<td>✘</td>
<td>✔</td>
</tr>
</tbody>
</table>

Safe methods do not change the resource.

Idempotent methods can be called multiple times and always produce the same result.
In-Class Activity: Class Registration API

What resources (nouns) and verbs do we need?

Think (7 minutes)
Pair (7 minutes): ~3 minutes / person.
Share

Submit to: yellkey.com/term
API provides a **contract**

Once the API has been designed, client and server can be developed mostly independently.

Changes to API design break the contract: add versions to API URLs.

1. Manipulate UI
2. RESTful API Call
3. Process Request
4. RESTful Response (usually JSON)
5. Update UI
Asynchronous
(retrieves data from the server in the background, rather than after a page refresh)

JavaScript
(makes the asynchronous call, and process the response to update the page)

XML
(nowadays JSON, but can include any representation of a resource)
let xhr = new XMLHttpRequest();

// Handle the response
xhr.onload = function() {
    let response = JSON.parse(xhr.responseText);
    ...
};

// Make the request
xhr.open("GET", "/teas/green", true);
xhr.send();

Have to register the response handler before making the request. Will be called once the response returns, at some point in the future.

Handle request/response asynchronously (i.e., don't wait for xhr.send() to return).
function get(url, onsuccess, onerror) {
    let xhr = new XMLHttpRequest();

    // Handle the response
    xhr.onload = function() {
        onsuccess(xhr.responseText);
    };

    xhr.onerror = function() {
        onerror(xhr.statusText);
    };

    // Make the request
    xhr.open("GET", "/teas/green", true);
    xhr.send();
}
get("/teas/green", function(text) {
  let response = JSON.parse(text);
  ...}
});
Synchronous vs. Asynchronous

In synchronous communication, the browser sends a GET request to the server, waits for the response, and then proceeds with the next action. In asynchronous communication, the browser sends an XHR GET request to the server, does not wait for the response, and continues with other tasks.
Dealing with Asynchrony: **Continuations**

This callback is called a *continuation*. `get` returns immediately. But when it’s actually done (i.e., the response returns), program execution is passed to this callback function.

```javascript
get("/teas/green", function(text) {
    let response = JSON.parse(text);
    ...
});
```
Dealing with Asynchrony: *Continuations*

```javascript
button.addEventListener("click", function(event) {
  ...
});

get("/teas/green", function(text) {
  let response = JSON.parse(text);
  ...
});

app.use(function(req, res, next) {
  // next is a continuation.
});
```
get("/teas/green", function(text) {
    let greenTeas = JSON.parse(text);
    // can't return list of green teas!
});

The continuation is called at some arbitrary point in the future, outside the normal call stack. So, we cannot capture its result using a normal return statement.
Dealing with Asynchrony: *Side Effects*

Instead, we have to introduce additional *external state*.

```javascript
let greenTeas;

get("/teas/green", function(text) {
    greenTeas = JSON.parse(text);
});
```

And set this state within the continuation. This causes a *side effect*: the continuation is no-longer self-contained.

As code grows more complex, side effects become increasingly difficult to debug.
Dealing with Asynchrony: **Callback Hell**

```javascript
get("/teas/5", function(text) {
   let tea = JSON.parse(text);
   get("/recommendations/" + tea.type, function(text) {
      let recs = JSON.parse(text);
      recs.forEach(function(rec) {
         let div = document.createElement("div");
         div.addEventListener("click", function(event) {
            ...
         });
      });
   });
});
```

The *Pyramid of Doom* aka **Callback Hell**
Dealing with Asynchrony

Have to read code carefully to understand what order functions get called.

```javascript
get("/teas/5", getRecs);

function getRecs(text) {
    let tea = JSON.parse(text);
    get("/recommendations/" + tea.type, buildRecsDOM);
}

function buildRecsDOM(text) {
    let recs = JSON.parse(text);
    recs.forEach(function(rec) {
        let div = document.createElement("div");
        div.addEventListener("click", addToCart);
    });
}

function addToCart(event) {
    ...
}
```
Dealing with Asynchrony: *Promises*

A Promise is a *proxy* object for a value that will be determined some time in the future.

It is returned *synchronously* from an asynchronous function.
Dealing with Asynchrony: *Promises*

A Promise is a *proxy* object for a value that will be determined some time in the future.

It is returned *synchronously* from an asynchronous function.
function get(url, onsuccess, onerror) {
    let xhr = new XMLHttpRequest();

    // Handle the response
    xhr.onload = function() {
        onsuccess(xhr.responseText);
    };

    xhr.onerror = function() {
        onerror(xhr.statusText);
    };

    // Make the request
    xhr.open("GET", "/teas/green", true);
    xhr.send();
}
function get(url) {
    return new Promise((resolve, reject) => {
        let xhr = new XMLHttpRequest();

        xhr.onload = function() {
            resolve(xhr.responseText);
        };

        xhr.onerror = function() {
            reject(xhr.statusText);
        };

        // Make the request
        xhr.open("GET", url, true);
        xhr.send();
    });
}
Chains propagate error

```javascript
get("/teas/5")
  .then(function(text) {
    let tea = JSON.parse(text);
    return get("/recommendations/" + tea.type);
  })
  .then(function(text) {
    let recs = JSON.parse(text);
    recs.forEach(function(rec) {
      ...
    })
  })
  .catch(console.error);
```

Chains: execute async operations one after another by returning a promise within a then.

If an error occurs at any step, the chain is interrupted and we can catch and handle the error.
In-Class Activity: Callback Hell & Promises (~15 mins)

In Pairs. Load example.com in your web browser.

Functionality
1. Set the body background color after 1 second.
2. Set the div background color 1 second after step 1.
3. Set the p font color 1 second after step 2.

Implementation
1. Using nested callbacks
2. Using chained Promises.

Hint
setTimeout(fn, delay);
prop = "color"
elem.style[prop] = "white";
Example Domain

This domain is established to be used for illustrative examples in documents. You may use this domain in examples without prior coordination or asking for permission.

More information...
Example Domain

This domain is established to be used for illustrative examples in documents. You may use this domain in examples without prior coordination or asking for permission.

More information...

```javascript
function setStyle(selector, property, value) {
  return new Promise(function(resolve, reject) {
    setTimeout(function() {
      document.querySelector(selector).style[property] = value;
      resolve();
    }, 1000);
  });
}

setStyle('body', 'backgroundColor', 'salmon')
  .then(function() { return setStyle('div', 'backgroundColor', 'steelblue'); })
  .then(function() { return setStyle('p', 'color', 'white'); });
```

```text
Promise {<pending>}
```