Web App Concepts

Lou Marco

Version 1.0
July 2023

Web App Concepts

In this class, you will learn (course outline):

- What a web app is
- The difference between a web app and a website
- The HTTP request
- Some web app types
- A bit about AJAX
- The *fetch* API (and promises)
- The MVC architectural pattern
- A bit about REST Architecture

Web App Concepts

You:

- Have done some HTML/CSS
- And maybe a bit of JavaScript

This class:

- Lasts 1 day
- Contains 2 labs

Web App Concepts

?

QUESTIONS?

? ? ? ?

?

Web App Concepts

What is a Web Application?

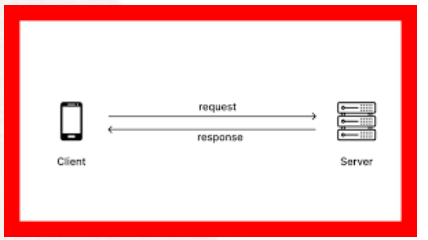
- A *web application* is an app that:
  - Uses the **browser as the user interface**
  - Is stored (hosted) on a *remote server*
    - The **web app code** on the server usually accesses *resources* (databases), making these resources available to the app user

- The user *communicates to the remote server by using a browser*
  - User enters info in browser input fields/forms
  - Browser has code that gathers/bundles user inputs to send to server

Web App Concepts

What is a Web Application?

- The browser communicates with the server by following a *communications protocol*
  - Communication protocols are formal descriptions of digital message formats and rules
    - Without a protocol, a sender may send 8-bit data where the receiver may expect 16 bit, for example



- The protocols used in web apps are **http** and **https**
  - Both allow browsers (clients) and servers to send/receive information from each other
  - The client sends a *request*; the server generates and returns a *response* based on client data
  - More on these protocols soon

Web App Concepts

What is a Web Application?

- The browser often contains HTML that identifies the *server resource to* receive the ***request*** sent by the client
  - A common technique is to code the resource as a value of the *action* attribute of the HTML *form* element

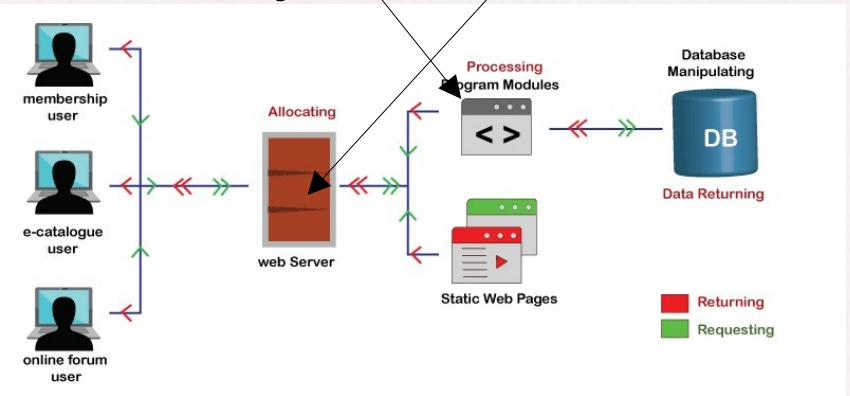
```
<form action="/path_to_resource/resource">
  <label>Enter your name</label> <br />
  <input type="text" name="firstname" placeholder="First name"><br />
  <input type="text" name="lastname" placeholder="Last name"><br /><br />

  <button type="submit">Submit</button>
</form>
```

  - Fill in the text fields, click Submit, the browser bundles user inputs (and other data) to create a **request** that is sent to the server resource *path_to_resource/resource*

Web App Concepts

What is a Web Application?

- The server resource is *executable code* stored in a directory on the web server



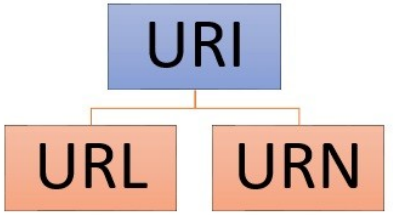
The web server contains code, config files, HTML, CSS, and other files depending on the technology (framework, DB) used to create the web app

- The server resource code extracts user data sent to server as a *request object* and creates a *response object* which server returns to the client
  - The *request/response cycle*

Web App Concepts

What is a Web Application?

- Server resources are identified by a URI
  - *Uniform Resource Identifier*, which may be:
    - URL: specifies a location on the computer network *and technique for retrieving it*
      - Click on a URL hyperlink to access resources
      - URLs typically start with *http:* or *https:*



    - URN: formal naming scheme that identifies a resource but *has no location or access information*
      - *Not clickable;* need apps to locate/access info
      - URNs must start with ***urn:***

  - The *path_to_resource/resource* used in the previous form example is a URL
    - The majority of web apps locate resources with URLs

Web App Concepts

What is a Web Application?

- The only difference between HTTP and HTTPS is that the HTTPS protocol uses *technology to encrypt data to/from client/server*

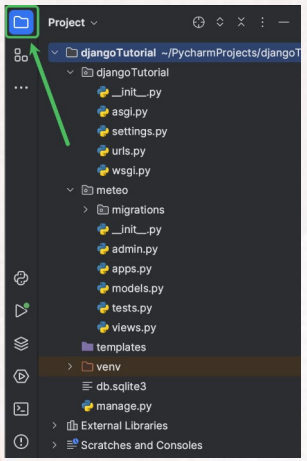
- The server and the web app must be configured to accept HTTPS requests; the client must be configured to accept HTTPS responses from the server
  - Details are beyond the scope of this course

Web App Concepts

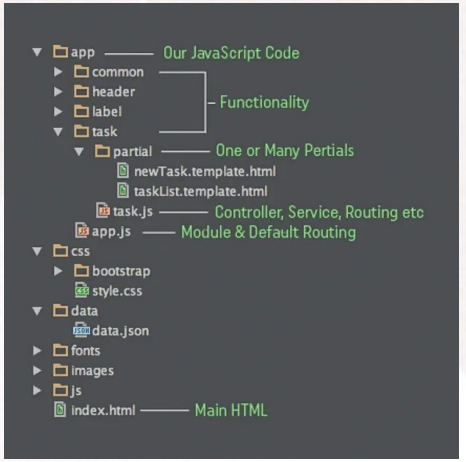
What is a Web Application?

- Web app directory structure/folders depends on the technologies used
  - The *framework* usually dictates the structure

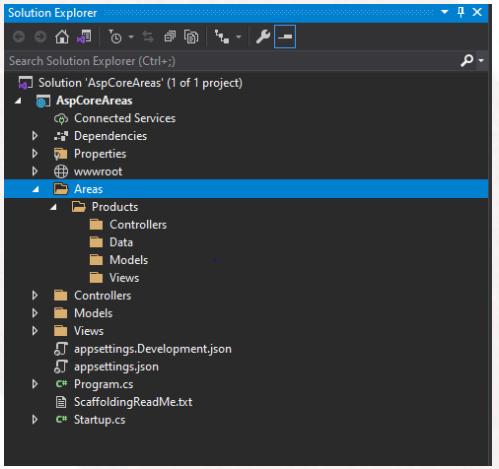
Django



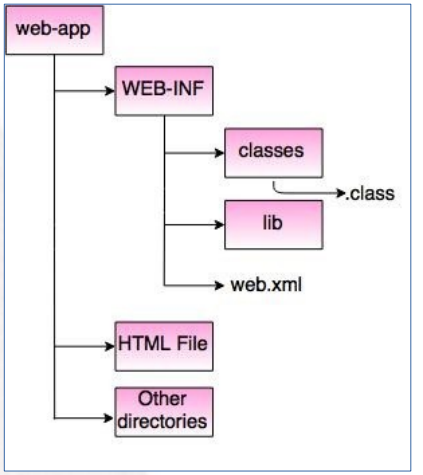
Angular



.NET MVC



Java JEE

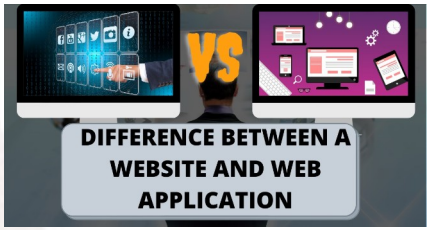


- Not set in stone; considerable flexibility

Web App Concepts

Difference between a *Web App* and *a Web Site*

- Web apps are *designed to be interactive, generating dynamic content*



  - User (via a browser) *requests the web app get (return) information* based on parameters entered by user
    - Requests sent to a *server resource*
    - Returned (response) data is **dynamic**
  - Based on info returned by the app, client *makes additional requests*, retrieving more information

  - Put differently, the web app *implements some business process (use case/user story)*

Web App Concepts

Difference between a *Web App* and *a Website*

- Websites are *designed to present static information*

  - User (via a browser) *requests the website get (return) information* based on parameters entered by user
    - Requests sent to another HTML page
    - Returned (response) data is **static**
  - Once response is received, additional client requests are *usually independent of the previous one(s)*

  - Put differently, the website provides a means to access static information

Web App Concepts

Difference between a *Web App* and *a Website*

WebSite	Web App
Static content delivering same info with same inputs	Dynamic content designed specifically for user interaction
User cannot manipulate/change site info (read-only)	User often has ability to change stored app data (database updates, e.g.)
Development consists of mostly static HTML pages (links to other doc types)	Development uses several technologies, frameworks, programming languages, development processes (**it's an app!**)
Often not requiring any authentication/authorization	Typically **requires** security, including authentication/authorization
Maintenance consists mainly of changing content, adding/removing HTML pages	Maintenance may involve changing HTML pages **and** changes to executable code (**it's an app!**)

Web App Concepts

The HTTP Request

- The client (browser) generates a *request* sent to the server via the *HTTP(s) protocol*
  - Browser has code that bundles inputs from a form plus other data suitable for HTTP transmission

    - Click *submit button*, browser bundles data and sends the bundle to the **URI string coded in the action attribute of the FORM html tag**
    - The browser sends a *full page of data* with the request
  - Later, we'll see that a programmer can *create and submit their own request, bypassing the browser code*
    - And *not send a full page of data (important!)*

Web App Concepts

The HTTP Request

- The HTTP Request contains:
  - The *Request line*, which contains:
    - **The Requested URI (Uniform Resource Identifier)**
    - **The Request method and Content**
    - **Various headers**

    - The three request line components are *what web application developers use*

  - Other HTTP request components are not commonly used:
    - The analysis of source IP address, proxy and port
    - The analysis of destination IP address, protocol, port and host

Web App Concepts

The HTTP Request

- The HTTP *request method* indicates the method to be performed on the resource identified by the requested URI
  - This method is case-sensitive, coded in uppercase

Method	Summary
GET	To fetch a single resource or group of resources
PUT	To update an entire resource in one go
POST	To create a new resource
PATCH	To partially update a resource
DELETE	To delete a resource
OPTIONS	To get information on permitted operations
HEAD	To get metadata of the endpoint
TRACE	For diagnosing purposes
CONNECT	To make the two-way connection between the client and the resource.

The ‘biggies’ for app developers are:

GET, POST, PUT, DELETE, PATCH

Later, we’ll see how to create a request using these methods

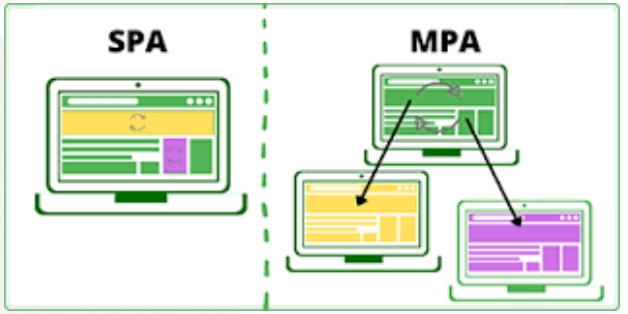
Web App Concepts

Some Web App Types

- Three main classifications:
  - Static web application
    - Built using HTML and CSS to facilitate exhibiting significant content and information
    - Examples: Digital resumes, lead marketing capture pages
  - Dynamic web application
    - Delivers live data based on user requests
    - Uses databases to store all private and public data displayed on the site
    - Sub-categories include:
      - E-Commerce: ‘Online stores’
      - Portals: Offer customized interfaces to specific user groups
      - Content Management (CMS): Content may be changed without using a programming language (WordPress, e.g.)
  - Both types typically send a *full page of data*

Web App Concepts

Some Web App Types

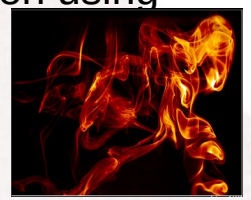


- Single Page web app (SPA)
  - App presented to user as a *single HTML page*
- The single page contains links that:
  - Call code in the page to *create an HTTP request*
    - Minimally including a:
      - Target URI,
      - HTTP *method (GET, POST, etc)*
      - Request *body (data sent to URI)*
    - **Important! Created request does not include a full page of data!**
      - Programmer chooses what data to send
        - Results in improved user experience

Web App Concepts

AJAX

- SPA sends the created request via **AJAX** calls
  - **A**synchronous **J**avaScript **a**nd **X**ML
    - Now, we mostly return data to client using *JSON* these days, not XML
    - Really should call it AJAJ
  - Every link in an SPA, when clicked, generates an AJAX call
- A non-SPA app may have links that generate AJAX calls as well
- All modern web frameworks support AJAX
  - Under the covers, the framework/library loads data into and sends the browser's XMLHTTPRequest object
    - XMLHTTPRequest object particulars not terribly important when using a framework/library when making AJAX calls



Web App Concepts

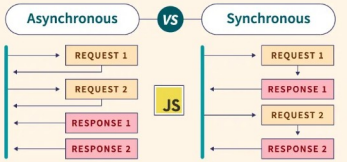
AJAX

- Server *does not know or care* that a request is AJAX or full-page
  - Servers receive/parse request data, generate a response, return response to client
- The API that generates the request and makes the AJAX call contains code that will:
  - Update (part of) the page upon success
    - Using data in the server-returned *response*
  - Take action when the call fails
- The code handling the AJAX call and server response *will NOT cause a full-page refresh*

Web App Concepts

AJAX

- Asynchronous programming is an essential concept in JavaScript (and other languages) that allows code to run in the background without blocking the execution of other code



  - E.G. - Program retrieves data from a remote server can continue to execute other tasks such as responding to user inputs
  - JavaScript on the browser is *single-threaded* but have limited multithreaded capability using *WebWorkers*
    - Anyone interested, check out this link:

https://www.loginradius.com/blog/engineering/adding-multi-threading-to-javascript-using-web-workers/

Web App Concepts

AJAX

- Example asynchronous JavaScript:

```
function fetchData(callback) {
    setTimeout(() => {
      const data = {name: "John", age: 30};
      callback(data);
    }, 3000);
  }

  // Execute function with a callback
  fetchData(function(data) {
    console.log(data);
  });

  console.log("Data is being fetched...");
```

```
Data is being fetched...
{name: 'John', age: 30}
```

SetTimeout() calls a function after a number of milliseconds
After 3 seconds, fetchData calls the anonymous callback function that writes the name/age object to the console

However, the code is executing *asynchronously*
The call by fetchData invoking the anonynous function is **non-blocking**, allowing the message “data is being fetched” to be written before fetchData completes execution

Web App Concepts

The *fetch* API (and Promises)

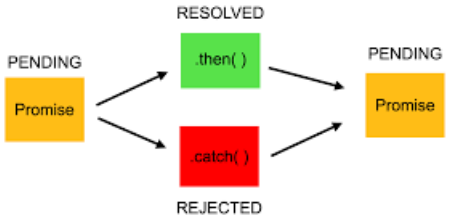
- *fetch* is an interface for making AJAX calls in JavaScript
  - Implemented by modern browsers

- Some in the literature distinguish AJAX and fetch
  - AJAX uses XMLHTTPRequest, fetch does not
  - HOWEVER, both allow a client to:
    - Create a request object
    - Send that request to a server
    - Retrieve/parse the response from the server
    - **Perform the above three steps *asynchronously***

  - We'll call ANY tech that allows for creating a custom request and parse the server response AJAX

Web App Concepts

The *fetch* API (and Promises)



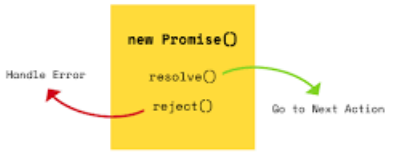
- Calling *fetch* returns a **promise object**, and a server response
- A Promise is an object representing the *eventual completion or failure* of an *asynchronous* operation
- We may use a CTOR to create a promise object (like an object of any class) but we'll spend most of our time *using the promise object returned by fetch*
  - Creating a promise object does not execute its methods
  - We need to *consume (use) the promise to invoke any code in the promise*

Web App Concepts

The *fetch* API (and Promises)

- Create a promise with its CTOR:

```
// This produces (creates) the promise
const aPromise = new Promise(function(resolveFunction, rejectFunction) {
    // Code that either accepts or rejects the promise
    const now = new Date();
    if (now.getHours() == 11)  // Current time 10:39 AM
        resolveFunction();
    else
        rejectFunction();

});
```



Creating the promise object. The code in the anonymous function argument uses two parms; a function to execute when successful (resolved) and one that executes when unsuccessful (rejected)

Basic CTOR:

```
let promise = new Promise(function(resolve, reject) {
    // Make an asynchronous call and either resolve or reject
});
```

Web App Concepts

The *fetch* API (and Promises)

- Use a promise by calling its **then** **and** **catch** **methods**

```
function doWhenResolved() {
    console.log("I execute when promise resolved successfully");
}
function doWhenRejected() {
    console.log("I execute when promise HAS NOT resolved (was rejected)");
}
// This consumes (uses) the promise
aPromise.then (doWhenResolved)
        .catch(doWhenRejected);
// Functions may be coded as anonymous functions
aPromise.then (function() {console.log("I execute when promise resolved successfully");})
        .catch(function() {console.log("I execute when promise HAS NOT resolved (was rejected");});
// Or using fat arrow functions
aPromise.then (() => console.log("I execute when promise resolved successfully"))
        .catch(() => console.log("I execute when promise HAS NOT resolved (was rejected"));
```

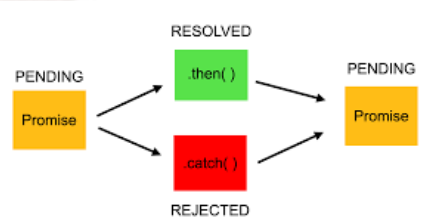
```
I execute when promise HAS NOT resolved (was rejected)
I execute when promise HAS NOT resolved (was rejected)
I execute when promise HAS NOT resolved (was rejected)
```

Promise objects have other methods; consult the docs.

Web App Concepts

The *fetch* API (and Promises)

- A promise object exists in one of three states:
  - Rejected: Code in the promise fails with an error
  - Resolved: (Fulfilled) Code in the promise executes successfully and returns a value
  - Pending: Code in the promise is still executing



- Some use the term *settled* to mean the promise has been resolved OR rejected

- Promises allow for *chaining calls to its then() method*
  - *Each call to then() returns another promise*
  - The *fetch* API often uses this technique; we'll see how and why soon

Web App Concepts

The *fetch* API (and Promises)

- Calling *fetch* returns (creates) a **promise object**
- Code in the *fetch* API calls the then() and catch() methods of the returned promise
- Use fetch to submit an HTTP request to some URI, save returned data, convert/reformat if needed, use in the app
  - Common to convert data to JSON (lots of sites return JSON data!)

Web App Concepts

The *fetch* API (and Promises)

```
const url = 'https://jsonplaceholder.typicode.com/users';

fetch(url)                      // Issue GET (default for fetch) request to above URI
.then((response) => {
    return response.json();     // On resolution, response is returned by URI
})                              // Create ANOTHER PROMISE, 'passing' JSON
.then((data) => {
    let authors = data;
    // List out ALL data returned (without stringify call, output not helpful)
    console.log(`All data returned via fetch: ${JSON.stringify(data)}`);
    // Iterate over returned JSON object, listing contents of two fields
    authors.map(function(author) {
        console.log(`Author Name: ${author.name}\tAuthor email: ${author.email}`);
    });
})
.catch(function(error) {        // Error object returned by fetch if call rejected
    console.log(error);
});
```

code-examples/fetch-ex.html

Let's check it out

Web App Concepts

Lab time!

Look at the PDF in the folder:

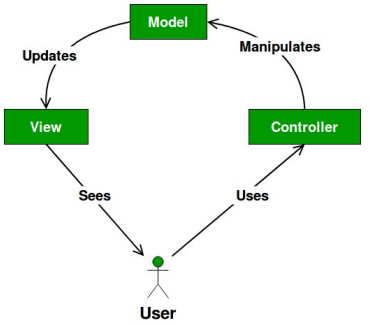
Lab 1 List Author names, email

Named: *Lab - Using fetch.pdf*

And do what it tells ya to do

Web App Concepts

The MVC architectural pattern

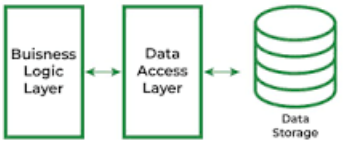


- Design pattern specifies that an application consist of a data **model**, **presentation** information, and **control** information
  - Considered *layers* of a multi-tiered application
- Each of these be separated into different objects
- The holy grail is that code implementing one layer can be changed without affecting code in the others

Web App Concepts

The MVC architectural pattern

- Modern web apps usually have *additional layers:*
  - Data Access layer
    - Responsible for managing the data storage and retrieval of an application
    - ‘sits’ between business-logic layer and data s
    - Allows the business-logic layer to interact with the data storage system without being aware of its specific implementation
    - Responsible for:
      - Connecting to the data storage system and managing the connection.
      - Generating and executing SQL queries or other data access commands to retrieve and store data.
      - Mapping the data from the data storage system to the application’s data objects and vice versa.
      - Handling errors and exceptions related to data access.
      - Providing support for transactions and other data access features



Web App Concepts

The MVC architectural pattern

- Modern web apps usually have *additional layers:*
  - Business logic layer
    - Handles the business logic, business rules as well as calculations

    - ‘sits’ between presentation (view) layer and data access layer
    - Processes and manipulates data before it is presented to the user or stored in the database
    - Responsible for:
      - Validating input data to ensure that it meets the necessary business rules and constraints.
      - Performing calculations and transformations on data, as required by the business logic.
      - Enforcing business rules and policies, such as access control and security.
      - Communicating with the data access layer to retrieve and store data.
      - Handling errors and exceptions

Web App Concepts

The MVC architectural pattern

- The **Model** contains only the pure application data
  - *In theory*, the model contains no logic describing how to present the data to a user
  - *In practice*, apps have cooperating layers used by code in the controller layer to *access data, perform business logic, perhaps a service layer*
  - A model is usually object representations of data stored in a persistent database

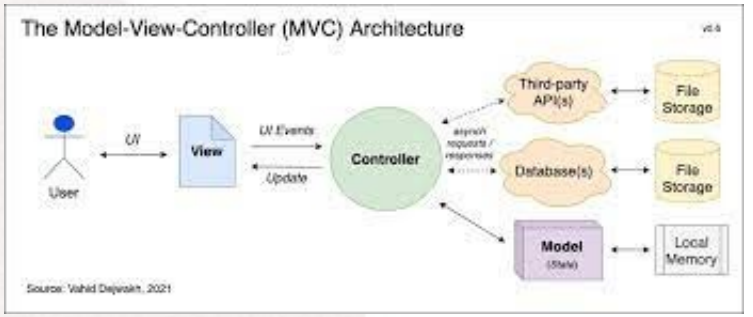
Web App Concepts

The MVC architectural pattern

- The **View** presents the model's data to the user
  - Mostly an HTML page or a server-side component that renders HTML pages
  - Web app frameworks usually use a *template subsystem* to generate HTML pages for views based on data manipulated by controller code
    - The template has references to data combined with static HTML which a server component 'fills in the blanks', renders a response, returns to client

Web App Concepts

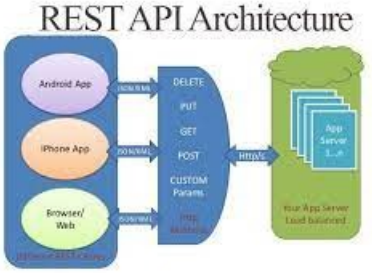
The MVC architectural pattern



- The **Controller** exists between the view and the model
  - Like a ‘traffic cop’ that calls code based on user selection/inputs
    - User selects ‘Create’; controller calls code to create data, etc.
      - Usually using a data access layer, business logic layer
  - Controller *generates a response* in the form of a *view component*
    - Details vary by framework

Web App Concepts

REST Architecture



- **REST** is an acronym for **RE**presentational **S**tate **T**ransfer and an architectural style for distributed hypermedia systems
  - Roy Fielding first presented it in 2000 in his famous dissertation*
- A Web API (or Web Service) conforming to the REST architectural style is a REST API
  - We say an API conforming to the guiding principles and constraints is a *RESTful API*
  - We'll concentrate on how to code a RESTful API, not delve deeply into Roy's six principles

* https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

Web App Concepts

REST Architecture

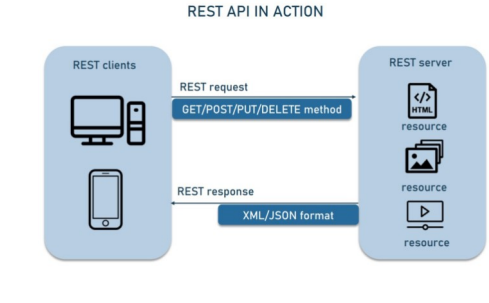
- Key REST Terms:
  - *Client*: A user of the API (browser)
  - *Resource*: Any data that the web APIs return to the client
    - Any information that can be *named*
  - *Endpoint*: perform a specific function, taking some number of parameters and return data to the client
  - *Route*: the “name” you use to access endpoints, used in the URL
    - Endpoint ==> Route + some action

- General flow:
  - Client interacts with the app (usually through an HTML page)
  - App generates a RESTful *route* that requests a *resource*
  - Sends the *route* to a server; the server triggers an *endpoint* that performs what is needed to get the requested *resource*
  - Returns the *resource* back to the *client*

Web App Concepts

REST Architecture

- REST API methods and request structure
  - Any REST request includes four essential parts:
    - An HTTP method
    - An endpoint
    - Headers
    - Body
  - The *HTTP method* describes what is to be done with a *resource*
    - *POST to **C**reate a resource,*
    - *GET to **R**etrieve a resource,*
    - *PUT to **U**pdate a resource, and*
    - *DELETE to **D**elete a resource*
    - *PATCH to Update part of a resource*



Often called CRUD

Web App Concepts

REST Architecture

- REST API methods and request structure
  - The *endpoint*
    - A *route* with an *action*
      - Most often a URI **with parameters** and an HTTP method
    - Location where API can access requested resources

    - Parameters include:
      - Path parameters that specify URL details.
      - Query parameters that request more information about the resource
      - Cookie parameters that authenticate clients quickly

    - This API call contains path and query parameters:

GET /surfreport/{beachId}?time=1400&units=metric&days=3

Fetch the 'surfreport' for the beach identified by *beachid* at *1400* hours for *three* days, using *metric* measurements

Web App Concepts

REST Architecture

- REST API methods and request structure
  - More on endpoint parameters
    - Path parameters are *part of the URL path* and are used to identify specific resources

GET /users/{userId}
userId is a path parameter that represents the unique identifier of a user

- Query parameters are added to the URL after a ? symbol and are used to *filter, sort, modify results returned by an endpoint*

GET /products?category=electronics&priceRange=100-500
category and priceRange are query parameters that filter the list of products based on category and price range

- Query parms are also used to specify sort parms, specify pagination (how many items/page), true/false conditions

.

Web App Concepts

REST Architecture

- REST API methods and request structure
  - Headers
    - Contains additional information about the request
    - Inclusion in the request depends on the language/framework

Header	Description
Authorization	Carries credentials containing the authentication information of the client for the resource being requested
Accept	indicates which content types, expressed as MIME types, the client is able to understand
Content-Type	Indicates the media type (text/html or text/JSON) of the response sent to the client by the server, this will help the client in processing the response body correctly
Cache-Control	This is the cache policy defined by the server for this response, a cached response can be stored by the client and re-used till the time defined by the Cache-Control header

This is a small list. Check out https://en.wikipedia.org/wiki/List_of_HTTP_header_fields for descriptions of the (about) 100 header fields

Web App Concepts

REST Architecture

- REST API methods and request structure
  - Request body
    - Most often used for PUT, POST, PATCH
    - Any data used by REST calls using these HTTP methods her the required data in the resuest body

    - POST Add new data
    - PUT Replace existing data
    - PATCH Update existing data

Web App Concepts

REST Architecture

- Example REST API calls
  - GET request

```
const apiUrl = 'https://api.example.com/data';
const accessToken = 'your_access_token';

const headers = new Headers();
headers.append('Authorization', `Bearer ${accessToken}`);
headers.append('Content-Type', 'application/json');

fetch(apiUrl, {
  method: 'GET',
  headers: headers,
})
.then(response => {
  if (!response.ok) {
    throw new Error('Network response was not ok');
  }
  return response.json();
})
.then(data => {
  console.log('Data:', data);
})
.catch(error => {
  console.error('Error:', error);
});
```

GET resource from apiUri using the headers shown

We'll talk about **response codes** in a bit

Web App Concepts

REST Architecture

- Example REST API calls
  - POST request

```
const apiUrl = 'https://api.example.com/data';
const accessToken = 'your_access_token';

const headers = new Headers();
headers.append('Authorization', `Bearer ${accessToken}`);
headers.append('Content-Type', 'application/json');

const requestBody = {
  username: 'exampleuser',
  email: 'user@example.com',
  role: 'user',
};

fetch(apiUrl, {
  method: 'POST',
  headers: headers,
  body: JSON.stringify(requestBody),
})
// Other code same as shown in previous OH
```

POST adds new data
Include the new data; details depend on language used

Note path and query parms are **usually** absent from POST requests

Web App Concepts

REST Architecture

- Example REST API calls
  - PUT request

```
const apiUrl = 'https://api.example.com/data/123';
const accessToken = 'your_access_token';

const headers = new Headers();
headers.append('Authorization', `Bearer ${accessToken}`);
headers.append('Content-Type', 'application/json');

const requestBody = {
  username: 'updateduser',
  email: 'updated@example.com',
};

fetch(apiUrl, {
  method: 'PUT',
  headers: headers,
  body: JSON.stringify(requestBody),
})
// Other code same as shown in previous OH
```

PUT replaces existing data
Include the new data; details depend on language used

Assume we want to replace data id'd by the URI by **123**, which is replaced with new data or created if not found

Web App Concepts

REST Architecture

- Example REST API calls
  - PATCH request

```
const apiUrl = 'https://api.example.com/data/123';
const accessToken = 'your_access_token';

const headers = new Headers();
headers.append('Authorization', `Bearer ${accessToken}`);
headers.append('Content-Type', 'application/json');

// Other fields in this object on the server are not
// changed
const requestBody = {
  email: 'updated@example.com',
};

fetch(apiUrl, {
  method: 'PATCH',
  headers: headers,
  body: JSON.stringify(requestBody),
})
// Other code same as shown in previous OH
```

PATCH updates existing data
Include the updated data; details depend on language used

Assume we want to update data id'd by the URI by **123, which must exist and is updated with new data**

Web App Concepts

REST Architecture

- Example REST API calls
  - DELETE request

```
const apiUrl = 'https://api.example.com/data/123';
const accessToken = 'your_access_token';

const headers = new Headers();
headers.append('Authorization', `Bearer ${accessToken}`);

fetch(apiUrl, {
  method: 'DELETE',
  headers: headers,
})
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json(); // Not typically needed for a
                            // DELETE request
  })
  .then(data => {
    console.log('Resource deleted successfully');
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

DELETE removes existing data

Assume we want to delete data id'd by the URI by **123**

Web App Concepts

REST Architecture

- Common status (response) codes for HTTP verbs when *successfully* executed:
  - GET 200 (OK)
  - POST 201 (Created)
    204 (No content) Successful, no additional content
  - PUT 200 (OK)
    201 (Created)
    204 (No content) Successful, no additional content
  - DELETE 200 (OK)
    201 (Created)
    202 (Accepted) Server accepted request but resource not yet deleted

These calls may return *errors*; codes in the 400-500 range

https://www.restapitutorial.com/httpstatuscodes.html

Web App Concepts

Lab 2: People project

Read the PDF in Lab 2 – People Project

Starter code in *People Project Starters*