COMP 3015

Week 9: PHP Frameworks, Web APIs

Week 9 Topics

- Web APIs
  - What is an API?
  - What is a web API?
  - Why are web APIs useful?

- PHP Frameworks
  - Laravel

- Building a web API
  - HTTP clients: Postman, Insomnia, cURL, etc.

Part 1: APIs

What is an API?

How two pieces of software can communicate.

eg. PHP provides two APIs to interact with MySQL (mysqli, PDO):

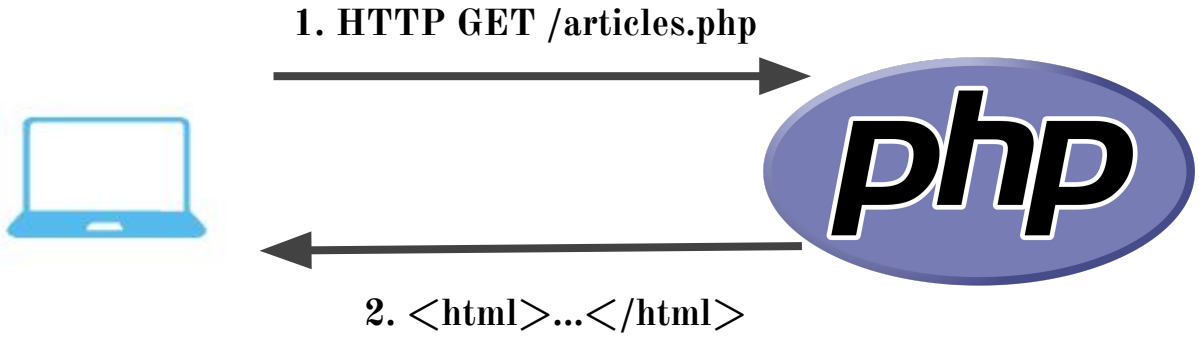
https://www.php.net/manual/en/mysqlinfo.api.choosing.php

Part 2: Web APIs

HTML centric applications: (past ~9 weeks)

Client (browser, in this case)

Your PHP application



What if you build this and then you need to support iOS and Android apps?

Web APIs: Ideas

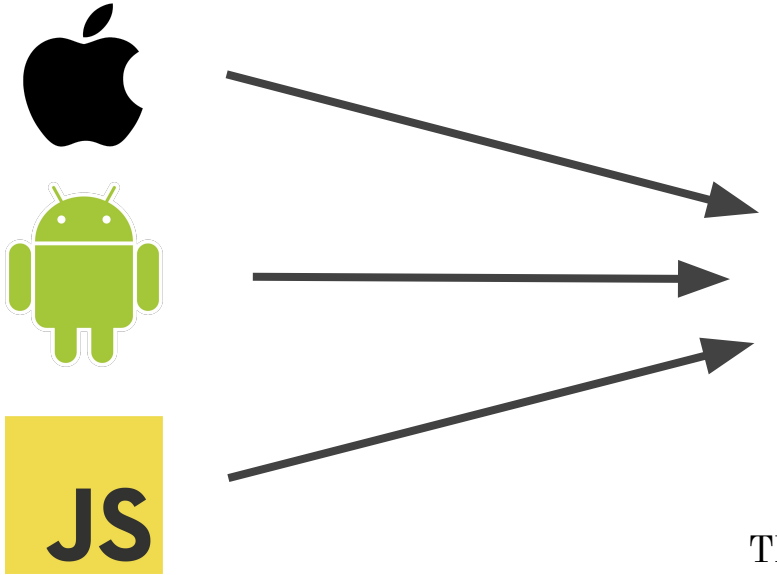
- Build server side applications that can be used for any clients
  - Not limited to interacting with browsers

- Send response data in a platform independent form (HTML needs a parser in the browser to be interpreted)
  - Basically any client can interpret JSON payloads

Web Application Programming Interfaces (APIs)

Lots of types of clients

Notice that any client capable of making requests can interact with a the web API!



Your web API

Think of software like Facebook or various Google applications: iOS, Android and web clients

Web Application Programming Interfaces (APIs)

- Common to build APIs within a company to have one server side system support unlimited types of clients (e.g. iOS, Android, web apps, watch apps, TVs, the next big thing in X years, etc.)

Part 3: RESTful Web APIs

How should we structure web APIs?

One main endpoint?

HTTP GET /main

We could pass data in the URL that says what to do on the server

HTTP GET /main?action=getArticle&id=123

This would work. We have started to use HTTP as a transport layer protocol, though. As our applications increase in complexity we begin to build our own protocol on top of HTTP.

We can do better.

Introducing resources

Idea:

No more **/main** endpoint. Instead, reference the type of resource (eg. article, user) you want to interact with:

HTTP GET /users ← get all users
HTTP GET /articles/<id> ← get the article with :id

Introducing HTTP Verbs: GET, POST, PUT, DELETE

Remember that a verb indicates an action! Think of the endpoints on our servers as being actions that will be carried out.

HTTP **GET** /articles	←	get all articles
HTTP **POST** /articles	←	create a new article
HTTP **PUT** /articles/<id>	←	replace (used to update) the article with <id>
HTTP **DELETE** /articles/<id>	←	delete the article with <id>

- The above should look kind of familiar: we've been building applications roughly following this structure already
- Note that HTTP POST and PUT data is in the body of the HTTP request.
- HTTP DELETE just uses the URL - don't provide a HTTP request body for DELETE

RESTful Web APIs

REST: Representational State Transfer

- Originally described by Roy Fielding in his 2000 PhD thesis: https://www.ics.uci.edu/~fielding/pubs/dissertation/software_arch.htm

- REST is a set of architectural design principles for client-server programs
- No official standard
- Not a protocol
- Academic definitions and applied usage differs
  - We'll start with the applied usage to get an intuitive feeling for RESTful services

RESTful Endpoints Example

HTTP GET /articles	←	return all articles
HTTP POST /articles	←	save a new article
HTTP PUT /articles/{id}	←	update an article (replace an article)
HTTP DELETE /articles/{id}	←	delete an article

HTTP request types

GET ← idempotent, data goes in the URL as query parameters

POST ← data goes in the body of the HTTP message

PUT ← idempotent, data in the HTTP message body

DELETE ← idempotent, requests deletion of a resource

Idempotent: system state does not change by making the same request multiple times. eg 1. you can't delete something twice. eg 2. You can't update a title to "test" twice. Once it's updated once a subsequent update doesn't do anything.

RESTful APIs, optional (but suggested) reading

https://martinfowler.com/articles/richardsonMaturityModel.html

Part 3.5 API endpoint versioning

API endpoint versioning

- Sometimes it’s not possible to make backwards compatible API changes.
- Consider a project where you’re refactoring a system to be centered around companies instead of users. Each company has many users and existing users can invite more users via email.
- Resources are now owned by “**company**” records instead of “**user**” records.
- A production server has the endpoint: **/api/subscription/{user_id}**

API endpoint versioning continued

- The production server has the endpoint: **/api/subscription/{user_id}**

- We need to change it to **/api/subscription/{company_id}**

- **Danger**: we can't simply update our server to accept a company ID instead of user ID.
  - The existing clients (front end web app, iOS, Android apps) will still send the user ID and potentially request a subscription update for the wrong company.

API endpoint versioning continued

The solution is to version our API endpoints:

- **/api/v1/subscription/{user_id}** → **/api/v2/subscription/{company_id}**

- Release process:
  - The server-side update is released first
  - Each client application can selectively update to the **/api/v2/subscription/{company_id}**

- If anything goes wrong, a client application can revert to the **/api/v1/...** endpoint

Part 4: Frameworks

Server Side Frameworks

Java: **Spring**, **Play**

Ruby: **Ruby on Rails**

Node: **Express**

PHP: **Laravel**, **Symfony**, **Laminas**

Lots of web frameworks use very similar patterns/ideas. Learning one of them makes it easy to learn the next one you're interested in.

Frameworks: Why?

- Lots of web applications do similar things
  - Authentication
  - CRUD
  - Cookie handling
  - Sessions
  - Conditional rendering
  - Caching
  - Middleware

- Frameworks provide a common way of handling these things, and speeds up how quickly we can deliver high quality software.
- Why reinvent the wheel?

Client Side JS Frameworks

Outside of the scope of this course

React, and a full stack framework on top of React: NextJS

Vue, and a full stack framework on top of Vue: NuxtJS

- BCIT offers:
  - https://www.bcit.ca/courses/react-and-modern-javascript-comp-2913/
  - https://www.bcit.ca/courses/angular-and-vue-js-fundamentals-comp-2909/

Web Frameworks

Typical Features

- Routing: map a URL to a function e.g. **/posts/1** → **getPost($id)**
- Built-in defences against SQL injection attacks
- Object-Relational Mapper
- Database Migrations
- Middleware e.g. **redirectIfAuthenticated**
  - Automatically run on every request to your application (depending on config)
- Application scaffolding: e.g. Laravel can generate an authentication system (e.g. register, login, logout) with one command

Laravel Intro

- https://laravel.com/
- Open source software: https://github.com/laravel/laravel
- Tutorials: https://laracasts.com/

Laravel: Getting Started

```
$ composer create-project laravel/laravel <project name here>

$ cd <project name here>

$ php artisan serve
```

Laravel: Getting Started

- All applications have an entry point, standard default structure
- The **public/index.php** script is the entrypoint for Laravel applications
- (PSR-4) Autoloading is configured by default

Laravel: Application Scaffolding

$ php artisan make:model <model name> -mcr

-m: model

-c: controller

-r: resourceful controller → particular functions signatures generated for you

or:

$ php artisan make:model <model name> --all

Database Migrations

Used to keep track of database changes (they're files that get tracked by Git), rollback changes

```php
 * Run the migrations.
 *
 * @return void
 */
public function up()
{
    Schema::create( table: 'users', function (Blueprint $table) {
        $table->id();
        $table->string( column: 'name');
        $table->string( column: 'email')->unique();
        $table->timestamp( column: 'email_verified_at')->nullable();
        $table->string( column: 'password');
        $table->rememberToken();
        $table->timestamps();
    });
}
```

Database Migrations (cont.)

Running migrations: **$ php artisan migrate**

```
~/Work/BCIT/COMP3015/Week9/lab-6-solution $ php artisan migrate

  INFO  Preparing database.

  Creating migration table ..................................................... 47ms DONE

  INFO  Running migrations.

  2014_10_12_000000_create_users_table ......................................... 38ms DONE
  2014_10_12_100000_create_password_resets_table ............................... 25ms DONE
  2019_08_19_000000_create_failed_jobs_table ................................... 25ms DONE
  2019_12_14_000001_create_personal_access_tokens_table ........................ 32ms DONE
  2022_11_07_020104_create_articles_table ...................................... 12ms DONE
```

Database Migrations (cont. from last slide)

After running the DB migrations we can get a shell to MySQL and confirm that the tables were created.

```
Database changed
mysql> show tables;
+-----------------------+
| Tables_in_lab_6       |
+-----------------------+
| articles              |
| failed_jobs           |
| migrations            |
| password_resets       |
| personal_access_tokens |
| users                 |
+-----------------------+
6 rows in set (0.00 sec)

mysql>
```

Request-function mapping

Idea: request that is made to our server should be handled by a function.

For example:

HTTP GET /articles maps to **public function getArticles(Request $request) {...}**

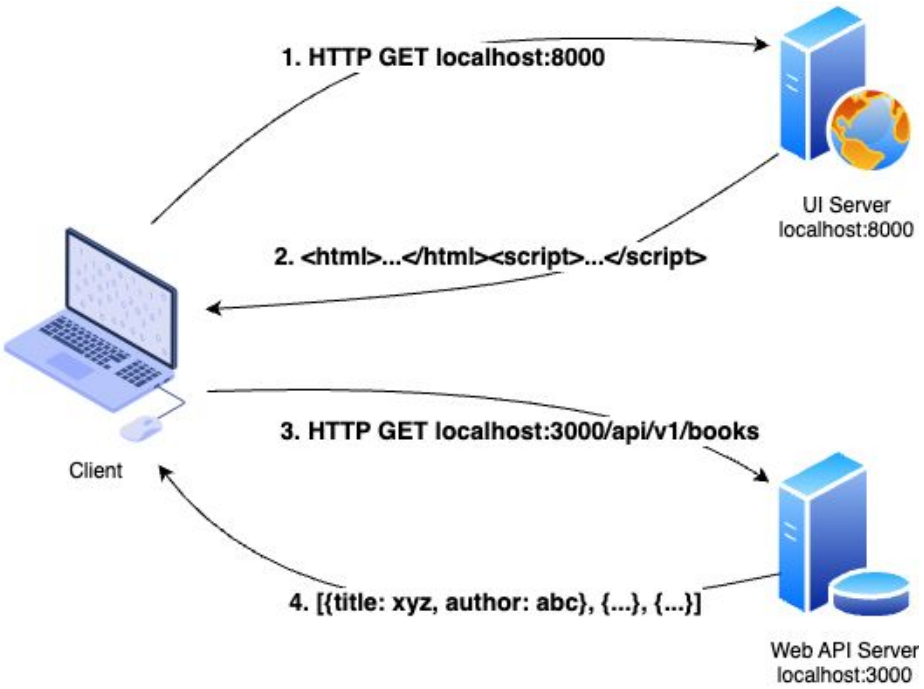
This process is called routing. You can build a router on your own in plain PHP code. Typical implementations use:
https://www.php.net/manual/en/function.call-user-func.php

Optional Reading

- History of REST: https://twobithistory.org/2020/06/28/rest.html

Part 5: CORS (optional)

Web API/UI Server Architecture



NOTE

Laravel now configures CORS to be set up in a very permissive manner.

This is not the case for all frameworks out there.

See **cors.php**

Cross-Origin Resource Sharing

The response was blocked by the browser by the same origin policy.

```
Access to fetch at 'http://localhost:3000/api/v1/books'   index.html:1
from origin 'http://localhost:8000' has been blocked by CORS policy:
No 'Access-Control-Allow-Origin' header is present on the requested
resource. If an opaque response serves your needs, set the request's
mode to 'no-cors' to fetch the resource with CORS disabled.

▶GET http://localhost:3000/api/v1/books          index.html:16
net::ERR_FAILED 200 (OK)

▶Fetch failed loading: GET "http://localhost:3000/api/   index.html:16
v1/books".
```

Same Origin Policy (SOP)

- Security measure implemented by browsers
- Imagine without the Same Origin Policy
  - You're authenticated with your online banking website 🔐
  - You (accidentally) go to a malicious website in another tab
  - The malicious website can do anything you can do on your banking website (via JavaScript) – reading cookies, making reqs on your behalf, etc.
- This is the kind of case that the SOP aims to mitigate

Access-Control-Allow-Origin response header

Access-Control-Allow-Origin:	http://localhost:8000
Connection:	keep-alive
Content-Length:	849
Content-Type:	application/json; charset=utf-8
Date:	Tue, 24 Oct 2023 00:00:43 GMT
Etag:	W/"351-PvhzE3km40cuQB36dJDfsDjStxY"
Keep-Alive:	timeout=5
Set-Cookie:	COMP_3012_lab_3=s%3AMsvCW8MONsEmw-xeSTJictoxLDlmbVeQ.wlEx
Vary:	Origin
X-Powered-By:	Express

▾ Request Headers ☐ Raw

Accept:	*/*
Accept-Encoding:	gzip, deflate, br
Accept-Language:	en-CA,en-GB;q=0.9,en-US;q=0.8,en;q=0.7,la;q=0.6,de;q=0.5
Cache-Control:	no-cache
Connection:	keep-alive
Host:	localhost:3000
Origin:	http://localhost:8000
Pragma:	no-cache

Need to match!

If the Access-Control-Allow-Origin response header value does not match the Origin request header, the response will be blocked from being read.

Note that this is the case for "simple requests". Other requests may trigger a "preflight" CORS request which will occur before sending the actual cross-origin request. This prevents write operations.