CSMC Technologies

Outline

- Overview
- Motivation
- Tutorials
- Resources

Overview

Technologies used by the CSMC

Backend

- CentOS
- Apache
- MySQL
- Docker
  - Alpine
  - Nginx
- PHP
  - Symfony
  - Doctrine
  - Composer
  - Deployer
  - Imagemagick
- JavaScript
  - Webpack
  - Yarn

Frontend

- Twig
- JavaScript
  - JQuery
  - Datatables
  - Fullcalendar
  - Cropper
  - Dropper
- HTML
- CSS
  - Bootstrap
  - Sass

Motivation

CentOS

Why not use a different operating system?

- Was provided to us

Advantages

- Based on RedHat
- Runs only stable versions of packages

Disadvantages

- Software compatibility
- Slow to update

MySQL

Why not use a different DBMS? PostgreSQL? SQLite? Oracle?

- Free/Open source
- Large Community

Advantages

- Scalable
- High performance

Disadvantages

- Does not implement full SQL standard

PHP

Why not use a different language? JavaScript? Java EE? Ruby?

- Familiarity

Advantages

- High Performance
- Large Community

Disadvantages

- Bad reputation
- Interpreted (i.e. not compiled)

PHP

Symfony

- Free/Open source
- Easy to use but powerful
- Lots of features

Doctrine

- Free/Open source
- Easy to use but powerful
- Lots of features
- Removes need for manual SQL queries

Frontend

Twig

- Packaged with Symfony
- Rendered on backend

Bootstrap

- Easy to use
- Looks good
- Lots of features
- Large community

Frontend

JQuery

- Required by Bootstrap
- Easy to use
- Large community
- Lots of features

Sass

- Compiled CSS
- Variables

Docker

Why not use a normal VM? Install everything separately?

- Cumbersome
- Slow
- Hard to change individual parts

Advantages

- Fast
- Smaller than a VM
- Easy to swap packages

Disadvantages

- Still not native speeds
- Some packages slow to update
- Needs Hyper-V on Windows

Git

Why not use another source control? Mercurial? Fossil? Subversion?

- Familiarity/Popularity

Advantages

- Robust
- Freedom
- Many hosting options (e.g. Github, Bitbucket)

Disadvantages

- Confusing to novices
- Freedom

Tutorials

Git

Basic Commands

- clone
- commit
- push
- pull
- merge

Working In a Team

- Choose a workflow that works for the team and the project
- Be consistent and follow the workflow faithfully
- Communicate with your teammates about your work and their work
  - Avoid conflicts
  - Avoid reimplementation

Workflows

- Centralized
- Feature-branch
- Forking

Centralized Workflow

- How does it work?
  - Maintain a single centralized repository
  - Developers push code to centralized repository after completing features on local repositories
- Advantages
  - Easy to setup and understand
- Disadvantages
  - Developers can all push to the same repository (no built-in checks)
  - Potential for a higher chance of conflicts
  - Not very suitable for larger teams

Feature-branch Workflow

- How does it work?
  - Maintain a single centralized repository
  - Each new feature's development is done in a separate branch
  - When a feature is complete, branch is merged with the main branch
- Advantages
  - Clear distinction of individual feature development
  - Lessens chance of conflicts during development of multiple features
- Disadvantages
  - Many branches can be confusing
  - Not very suitable for smaller teams

Forking Workflow

- How does it work?
  - Maintain a centralized repository
  - Each developer maintains a fork
  - Developers push local changes to their fork
  - When feature development is finished, developer makes a pull request to central repository
- Advantages
  - Pull requests give developers the ability to discuss and review other people's contributions
  - Maintains a better record of contributions
  - Each fork can be developed by a single or multiple developers
  - Suitable for teams of all sizes
- Disadvantages
  - Substantial organizational commitment

Best Practices

- Consistency is key
- Commit early and often
- Be descriptive with commit messages
- Maintain communication with your team to avoid conflicts

Docker

Overview

- Container management system
- Abstract different technologies into separate containers
- Quickly and easily install and configure different tools
- Rapidly switch between different tools
- Share configurations with others easily
- Multiplatform/crossplatform

Containers

- A virtual machine
- Runs a minimized OS and a specific tools defined by its configuration
- Can communicate with other containers through networks
- Highly configurable

Images

- Prebuilt configured containers
- Specific tools may have many images
  - For example PHP
    - fpm-alpine3.6
    - zts-stretch
    - cli
    - And many more
- Built for specific versions of tools or set of tools

Docker-compose

- Allows a collection of containers and networks to be defined together in a single file
- Ease the development of multi container systems
- Containers can be built and run in single commands versus a command per container

Commands

- To build containers:
  - docker-compose build
- To run containers:
  - docker-compose up -d
- To run commands in a container:
  - docker-compose exec <container>

Symfony

Overview

- PHP framework
- “A set of reusable PHP components”
  - Security
  - Command line
  - Requests/Responses
  - Controllers
  - Services

Structure

- Model View Controller (MVC)
- Model is defined by entities using Doctrine
- View is defined by templates using Twig
- Controllers are defined by Symfony
- Also have services and commands

Controllers

- Defined by a PHP class
- Each route is defined by a public function with the appropriate annotations
  - A route is a relative URL
  - Represents a request to the server and returns some response

Controllers

```php
<?php

namespace App\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class ExampleController extends Controller {
    /**
     * @Route("/example", name="example")
     */
    public function exampleAction() {
        return $this->render('example.html.twig');
    }
}
```

Controllers

```php
<?php

namespace App\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

class ExampleController extends Controller {
    /**
     * @Route("/example", name="example")
     */
    public function exampleAction(Request $request) {
        // do something with request

        return $this->render('example.html.twig');
    }
}
```

Controllers

```php
<?php

namespace App\Controller;

use App\Entity\User\User;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

class ExampleController extends Controller {
    /**
     * @Route("/example/{id}", name="example")
     */
    public function exampleAction(Request $request, User $user) {
        // do something with request

        // do something with $user

        return $this->render('example.html.twig');
    }
}
```

Controllers

```php
<?php

namespace App\Controller;

use App\Entity\User\User;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

class ExampleController extends Controller {
    /**
     * @Route("/example", name="example")
     */
    public function exampleAction(ExampleService $service) {
        // do something with $service

        return $this->render('example.html.twig');
    }
}
```

Services

- In general, any PHP class that does something
- Example services:
  - Logger
  - Mailer
- Don’t need to be configured (for the most part)
  - Symfony automatically does it
- Just use as an argument in your controller actions

Commands

- Run on the command line
  - php bin/console <command_name> <arguments>
- Used for cron jobs
  - Actions repeated at set intervals, not dependent on users
  - For example, sending weekly emails
- Also used for “one” time actions
  - Clearing Symfony’s cache

Commands

```php
<?php

namespace App\Command;

use Doctrine\ORM\EntityManagerInterface;
use Symfony\Component\Console\Command\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Output\OutputInterface;
use Symfony\Component\Console\Style\SymfonyStyle;

class ExampleCommand extends Command {
    protected static $defaultName = 'app:example';

    private $entityManager;

    public function __construct(EntityManagerInterface $entityManager) {
        $this->entityManager = $entityManager;

        parent::__construct();
    }

    protected function configure() {
        $this->setDescription('Run an example command')
            ->addArgument('example', InputArgument::REQUIRED, 'Example argument');
    }

    protected function execute(InputInterface $input, OutputInterface $output) {
        $io = new SymfonyStyle($input, $output);
        $example = $input->getArgument('example');

        // do stuff with arguments or whatever

        $io->success("Success!");
    }
}
```

Doctrine

Overview

- Object Relational Mapper (ORM)
  - “Translates” an object into a table for a relational database
- Automates basic queries
- Eases creation of complex queries
  - Query builder
  - DQL (probably won’t use)
- Creates the schema from entity definitions

Entities

- PHP classes
- Columns/relationships defined by annotations on member variables
- Everything is handled as if it were a normal class
  - No worrying about foreign keys
- No need to make insert/update/delete queries

Entities

```php
<?php

namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="example")
 */
class Example {
    /**
     * @ORM\Id
     * @ORM\Column(type="guid")
     * @ORM\GeneratedValue(strategy="UUID")
     */
    private $id;

    /**
     * @ORM\ManyToOne(targetEntity="AnotherExample")
     * @ORM\JoinColumn(name="other_id", referencedColumnName="id")
     */
    private $other;

    /**
     * @ORM\Column(type="string", length=255, name="value_column")
     */
    private $value;
}
```

Using Entities in Symfony

```php
<?php

namespace App\Controller;

use App\Entity\User\User;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

class ExampleController extends Controller {
    /**
     * @Route("/example", name="example")
     */
    public function exampleAction() {
        $users = $this->getDoctrine()
            ->getRepository(User::class)
            ->findAll();

        return $this->render('example.html.twig', array(
            'list_of_users' => $users
        ));
    }
}
```

Using Entities in Symfony

```php
<?php

namespace App\Controller;

use App\Entity\User\User;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

class ExampleController extends Controller {
    /**
     * @Route("/example", name="example")
     */
    public function exampleAction() {
        $users = $this->getDoctrine()
            ->getRepository(User::class)
            ->findAll();

        $example = $this->getDoctrin()
            ->getRepository(Example::class)
            ->findByValue("valueofthing");

        $another_example = $example->getOther();

        return $this->render('example.html.twig', array(
            'list_of_users' => $users,
            'other' => $another_example
        ));
    }
}
```

Entity Design

- Good entity design is important to ongoing development and maintenance
- What are entities again?
  - Entities represent the domain
  - Entities have behavior
- What are entities not?
  - Typed arrays
  - Data containers
  - Database tables

Entity Design

- What is “good” entity design?
  - Avoid anemic entities
  - Entities should always be valid
  - Don’t allow collection access from outside the entity
  - Keep collections hidden
  - Avoid setters
  - Avoid coupling
  - Use a Data Transfer Object (DTO) if needed
  - Don’t normalize for the sake of normalizing

Twig

Overview

- Abstracts the PHP code away from templates
- Allows inheritance in templates
- Templates make using different UI components reusable
- Simple syntax and almost all normal programming language constructs available

Example

```
{% include 'shared/base.html.twig' %}
{% block body %}
    {% set b = true %}
    {% if b == true %}
        {% for user in users %}
            <div class="btn">
                {{ user.firstName }}
            </div>
        {% else %}
            {{ other }}
        {% endfor %}
    {% endif %}
    <p>
        Any normal HTML stuff can go in templates
    </p>
{% endblock %}

{% block javascripts %}
    {{ parent() }}
    <script>
        $(function() {
            // do stuff here
        });
    </script>
{% endblock %}
```

How to make a link

- Link
- Link

Resources

Resources

- https://symfony.com/doc/3.4/index.html
- https://www.doctrine-project.org/projects/doctrine-orm/en/2.6/index.html
- https://twig.symfony.com/
- http://api.jquery.com/
- http://getbootstrap.com/docs/3.3/