Introduction to Symfony2

#symfony2workshop

Agenda

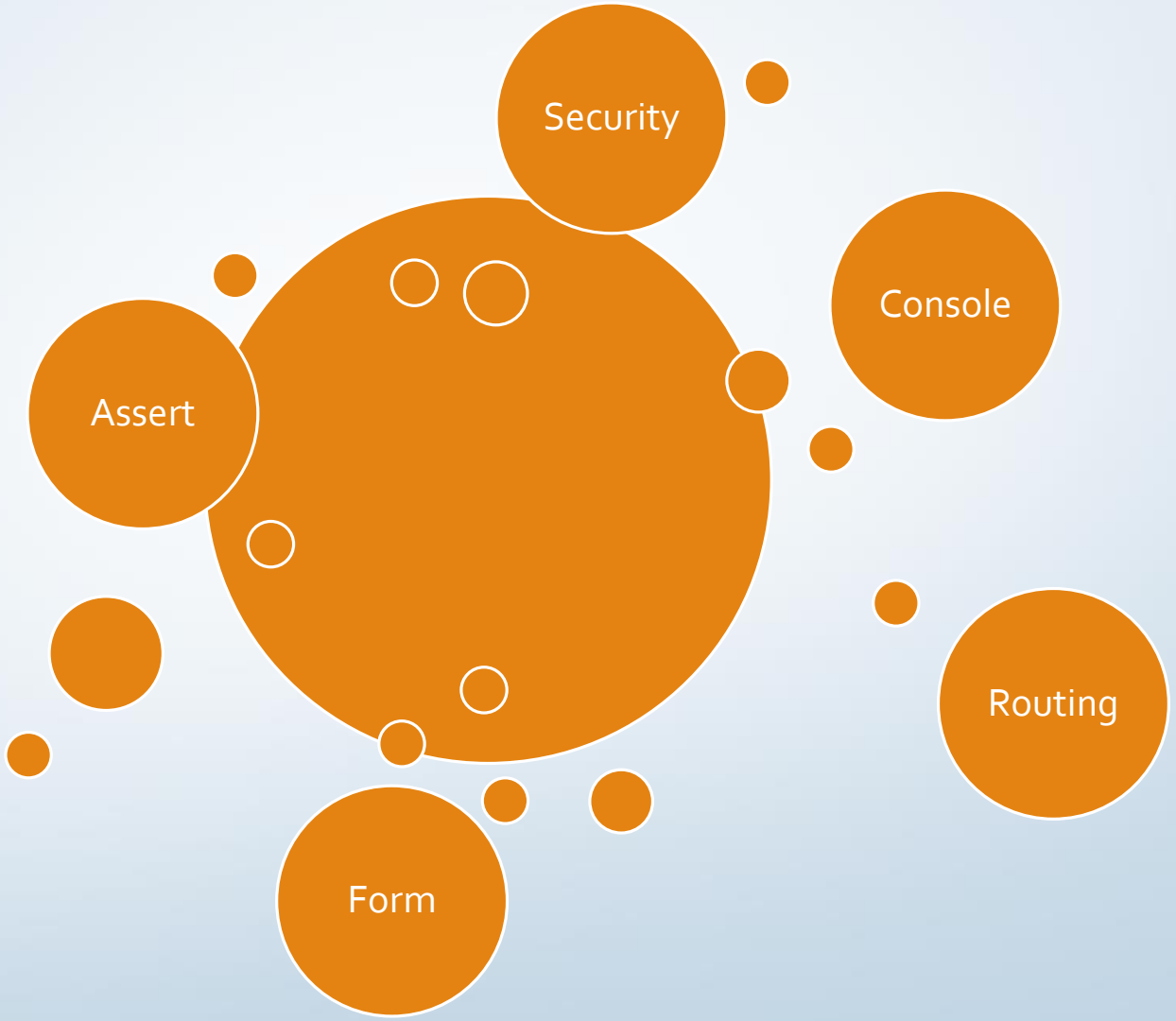
Time	
09:00	Doors Open
10:00	Introduction to Symfony Presentation (1:00 hour)
11:00	Workshop Part 1 (2:00 hour)
13:00	Lunch and Q&A (1 hour)
14:00	Workshop Part 2 (2:00 hour)
16:00	Close

What Symfony **is not**

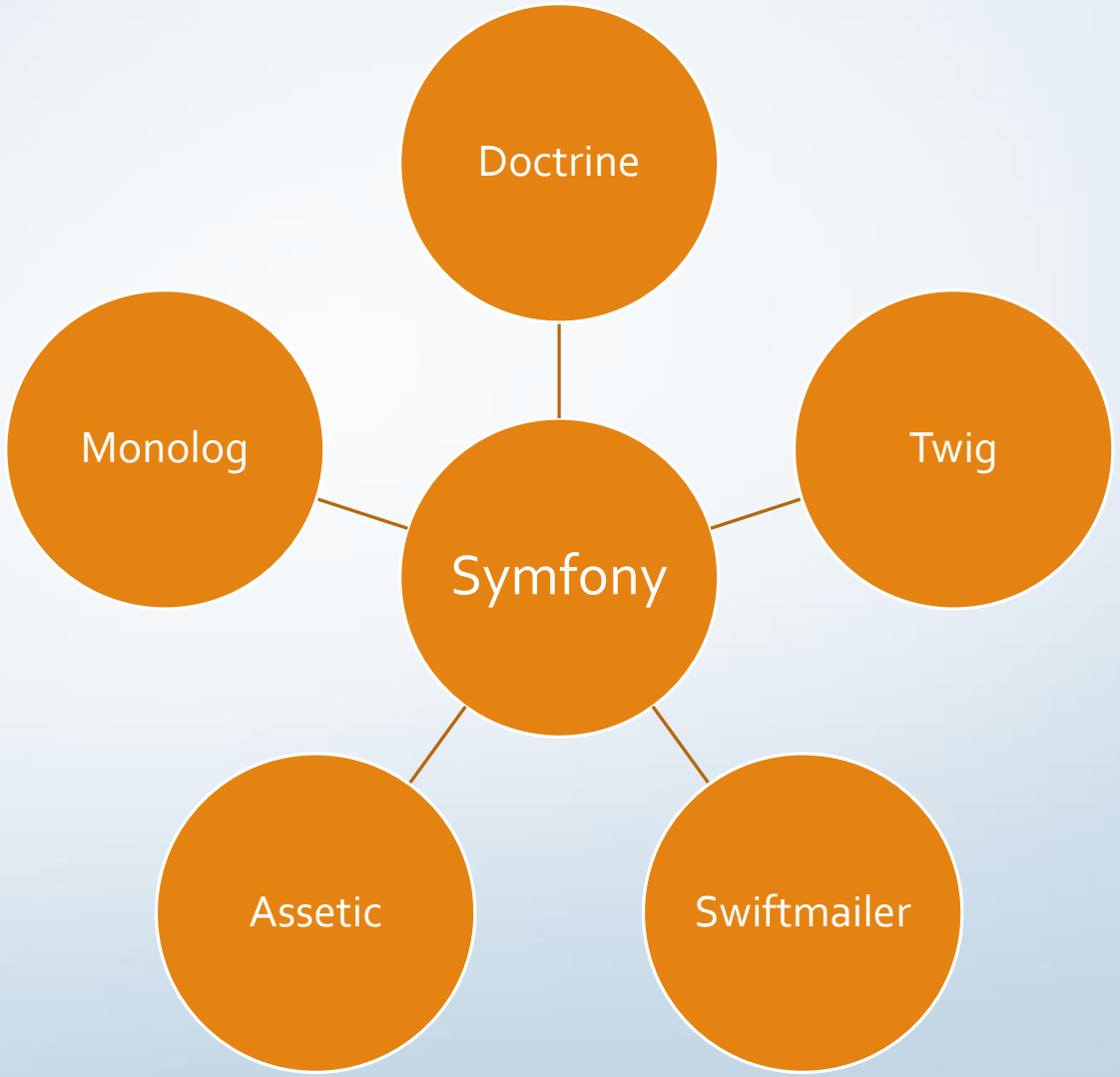
- An MVC framework
- Magic
- Written in French
- “Low Level” Code
- Only suitable for “Enterprise” applications

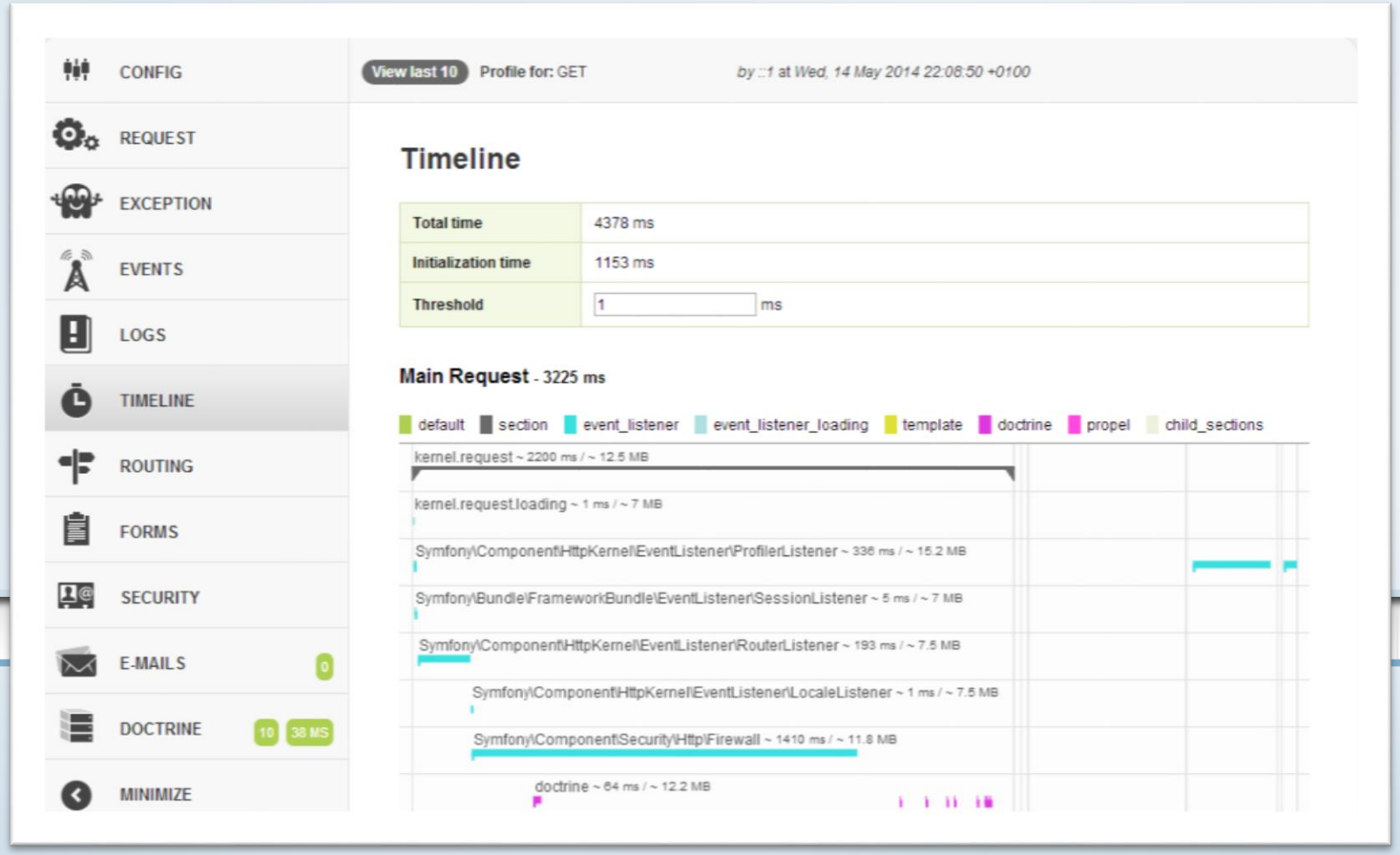
It is… a collection of pretty awesome **decoupled** tools for web app development…

25 to date



combine it with some other **independent** tools and you have a pretty epic HTTP framework





2.4.4

10

The Symfony Profiler

The not-so-secret weapon on every page to inspect every element of the web app

Symfony is also… a methodology

- Think of it to your code like an orthopaedic chair is to your back…
- Lastly it is a community… People that have adopted the methodology and genuinely think this is a positive change

Doctrine

Pretty awesome when it works

Doctrine

DBAL

- PDO “like”, it abstracts the database.
- Also supports management of schema, can create, update and drop columns and tables.
- Supports: MySQL, Oracle, MSSQL, PostgreSQL, SAP Sybase SQL Anywhere, SQLite, Drizzle

ORM

- Uses the DQL to query and manipulate the DB.
- Provides an easy to use model for your controllers and services to manipulate.
- Uses annotations on properties within your PHP model (entities) and relationships between them to create and manage your schema

Documentation

- Available at: http://docs.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/introduction.html
- Documentation is thorough but difficult to digest
- Caution: When Googling Doctrine related information, the results are littered with older information.

Watch your namespaces in annotations

Doctrine Documentation

```
/**
 * @OneToOne(targetEntity="Shipping")
 * @JoinColumn(name="shipping_id",
 *   referencedColumnName="id")
 */
private $shipping;
```

Your Code in Symfony

```
use Doctrine\ORM\Mapping as ORM

/**
 * @ORM\OneToOne(targetEntity="Shipping")
 * @ORM\JoinColumn(name="shipping_id",
 *   referencedColumnName="id")
 */
private $shipping;
```

Using Doctrine to update your SQL Schema

- You have already defined your entities and attached annotations to the properties
- Doctrine can keep your PHP Model and your SQL Schema in sync
- Relationships between entities have already been defined

Entities

- To mark a PHP class as being an entity you use the @ORM\Entity() annotation.
- By default Doctrine will apply defaults such as setting the DB table name as the unqualified class name
- You can override this by specifying your own table name using @ORM\Table() annotation

```
use Doctrine\ORM\Mapping as ORM
/**
 * @ORM\Entity
 * @ORM\Table(name="address")
 */
class CustomerAddress
{
   //...
}
```

Property Mapping

- You can specify that a PHP class property to be persisted to the database by using the @ORM\Column annotation.
- This annotation will assume defaults (type=string), so it's always best to override.
- All built in property types are portable across databases.
- Each property type can have specific attributes depending on the nature of the property.
- If all the built-in types do not fit your application, you can build your own.

```
use Doctrine\ORM\Mapping as ORM
/**
 * @ORM\Entity
 * @ORM\Table(name="address")
 */
class CustomerAddress
{
  /**
  * @ORM\Column(name="postcode",length="8")
  */
    private $postcode
}
```

Available Property Types

Types

- string: Type that maps an SQL VARCHAR to a PHP string.
- integer: Type that maps an SQL INT to a PHP integer.
- smallint: Type that maps a database SMALLINT to a PHP integer.
- bigint: Type that maps a database BIGINT to a PHP string.
- boolean: Type that maps an SQL boolean to a PHP boolean.
- decimal: Type that maps an SQL DECIMAL to a PHP double.
- date: Type that maps an SQL DATETIME to a PHP DateTime object.
- time: Type that maps an SQL TIME to a PHP DateTime object.
- datetime: Type that maps an SQL DATETIME/TIMESTAMP to a PHP DateTime object.
- text: Type that maps an SQL CLOB to a PHP string.
- object: Type that maps a SQL CLOB to a PHP object using serialize() and unserialize()
- array: Type that maps a SQL CLOB to a PHP object using serialize() and unserialize()
- float: Type that maps a SQL Float (Double Precision) to a PHP double. IMPORTANT: Works only with locale settings that use decimal points as separator.

Arguments

- type: (optional, defaults to 'string') The mapping type to use for the column.
- name: (optional, defaults to field name) The name of the column in the database.
- length: (optional, default 255) The length of the column in the database. (Applies only if a string-valued column is used).
- unique: (optional, default FALSE) Whether the column is a unique key.
- nullable: (optional, default FALSE) Whether the database column is nullable.
- precision: (optional, default 0) The precision for a decimal (exact numeric) column. (Applies only if a decimal column is used.)
- scale: (optional, default 0) The scale for a decimal (exact numeric) column. (Applies only if a decimal column is used.)

Associations

- Relationships can be defined between entities by using the following annotations on a property:
  - @ORM\OneToOne()
  - @ORM\OneToMany()
  - @ORM\ManyToOne()
  - @ORM\ManyToMany()

- In addition, the relationship can also be:
  - Unidirectional
  - Bi-Directional
  - Self Referencing
- Depends on whether you want both entities to be able to access the other (not always a good idea)
- Bi-Directional relationships will need an ‘owning’ side

Owning Side?

Doctrine will only check the owning side of an association for changes.

Bi-Directional
One to Many/Many to One

- The inverse side has to use the mappedBy attribute of the OneToOne, OneToMany, or ManyToMany mapping declaration. The mappedBy attribute contains the name of the association-field on the owning side.
- The owning side has to use the inversedBy attribute of the OneToOne, ManyToOne, or ManyToMany mapping declaration. The inversedBy attribute contains the name of the association-field on the inverse-side.
- ManyToOne is always the owning side of a bidirectional association.
- OneToMany is always the inverse side of a bidirectional association.
- The owning side of a OneToOne association is the entity with the table containing the foreign key.

Bi-Directional
Many-to-Many

- You can pick the owning side of a many-to-many association yourself.

Inheritance!

- If you like to use polymorphic classes in your model (we really do), then you'll love how Doctrine can persist these classes to the database either as:
  - A single super table compromising of all the columns required to cover all the child classes
  - Or as separate tables per class, each with the specific columns required for each entity.
- Doctrine will handle all the database actions such as schema updates and CRUD actions to the database.
- This is an advanced topic … for another time.

Getters and Setters

- All your classes will require the standard getters and setters for each property that you wish to use, e.g.:
- $private postcode
- getPostcode()
- setPostcode($postcode)

- IDE's can automatically generate these for you, but beware when doing them on relationship properties as Symfony expects add/remove/get vs. get/set
  - addAddress($address)
  - removeAddress($address)
  - getAddress()
- Doctrine can automatically generate these with the console command:

  $ doctrine:generate:entities
- Although beware when using it on Polymorphic entities.

How to Sync the Model and the Schema

Development

- `$ doctrine:schema:update`

- It's quick and dirty
- It will create **and drop** tables **and data**
- No rollbacks, no version control
- It will only update the schema (no data modifications)
- Do not use for anything but initial development.

Production

- `$ doctrine:migrations:diff`
  `$ doctrine:migrations:migrate`
- Can modify the data (not just the schema) using SQL commands and container aware high level logic migrations
- Version control
- Beware! It does not support transactions during a migration, it a migration fails halfway you could be left with an unstable database. Always test with a recent copy of the production DB before migrating!

Further Reading…

- Doctrine is a very powerful tool, it is also not the most forgiving if you get things wrong.
- The documentation isn't great but if you can overcome this you can be rewarded with some serious time savings.

Twig

Sweeter than Smarty

Twig

- Not just for HTML
- Templates can inherit and override others
- Can be ‘sandboxed’
- Supports
  - Loops,
  - Conditions
  - Filters

Documentation

- Available at: http://twig.sensiolabs.org/documentation
- Easy to understand, concise

Variables

- In a twig template you use {{ }} to echo a statement
- Imported objects can be accessed using the key names so if you imported [‘form’=>$form] into a template you can output $form by using {{ form }}
- If $form is an array you can access key/value pairs using the ‘.’ syntax. {{ form.url }}
- If form was an object you could access the getter methods using the same ‘.’ syntax. {{ form.url }} would call $form->getUrl() and $form->isUrl() on the object.

Expressions

- You can use expressions to modify variables, twig supports a wide range of common operators
- {{ 1 + 1 }} = 2
- {{ 1 and 0 }} = false
- {{11 % 7 }} = 4
- {{ 1 in [1, 2, 3] }} = true

- There are countless operators and expressions, the documentation is very easy to digest and is worth a read.

Macros

- Macros are a way of defining reusable code within a template.
- Beware at using them as a twig should only really be used at displaying data, keep business logic outside of the templates.
- We find them useful for example with disabling links (stripping the <a> tags from commands that the user does not have permission to use.

Filters

- On any expression you can apply a filter that will take the result of an expression and output the modified result. Filters are used by the '|' syntax.
- There are a number of built-in filters, although you can always create your own.
- Most of the built-in filters are shortcuts to PHP functions, although it's preferable to use them in the twig templates rather than using the PHP functions in the controllers/services.

- Examples on $form['price'] = 9.432599
- {{ form.price|number_format(2) }} = 9.43
- {{ form.price|abs }} = 9
- {{ form.price|round(1,'ceil') }} = 9.5

Conditions

- Conditions work pretty much the same way in PHP, except using the twig syntax.
- Use the {% %} syntax to escape from the template into twig

```
{% if users|length > 0 %}
    <p>User is logged in</p>
{% else %}
 <p>User is not logged in</p>
{% endif %}
```

Loops

- Loops again work pretty much the same way in PHP, except using the twig syntax.

```
<ul>
    {% for user in users %}
        <li>{{ user.username }}</li>
    {% else %}
        <li>No users</li>
    {% endfor %}
</ul>
```

Service container

Because life isn’t always easy

Imagine

- We have a simple logging class that logs to a text file
- Fancy new Logging as a Service website comes along
- We now want to log to this LaaS website but our log class is directly used in multiple classes and multiple projects

Dependency injection

Tightly coupled - bad

- Dependent upon other classes/modules
- Less flexible to change
- Cannot be shared amongst projects easily
- Changes cause a "ripple effect" because of fragile design and deeply tangled code

Loosely coupled - good

- Remove dependencies of other classes/modules
- Flexible to change, different implementations can be substituted – e.g. logger
- Functionality can be easily shared
- Separation of concerns leads to better, more flexible design decisions
- Easier to test

Service Container

- Acts as a registry of classes
- Loads classes based on an ID, rather than the actual class
- The class behind an ID can be substituted for other classes with enhanced or different functionality
- Injects the dependencies for each class directly, meaning the calling class doesn't have to pass them in itself.
- Classes are no longer tightly coupled