SPENDEE s.r.o., PHP backend developer test

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

Task details

In PHP, write an underlying implementation of a controller acting as a gateway to a JSON REST API web service.

In order for the task to be completed **the supplied code is not required to run**, you MUST however use all **Required components** (see below) and MUST follow all required criteria (specified with the MUST keyword).

Hint: If you want the supplied code to actually run, use any framework of your choice allowing for dependency resolution and injection - such as Nette or Symfony. Otherwise (when you do not care about the code running) simply consider a scenario where the code you are programming would be a part of a larger framework environment which does allow dependency resolution and injection.

The web service is used to download information about a watch from within the system and MUST return data as a JSON. The retrieval of the entity SHOULD NOT produce any side effects propagated to the user. It SHOULD also make use of the standard HTTP code definitions (see RFC 7231). The response MAY include additional information on an error.

Resource location:

```
/watch/{id}
```

Example output JSON of the Watch entity:

```
{
  "identification": 1,
```

```
  "title": "Watch with water fountain",
  "price": 200,
  "description": "Beautifully crafted timepiece for every gentleman."
}
```

You MAY nest the entity structure in a custom wrapper providing more information about the request, but the entity format MUST NOT be changed.
In addition to required components you SHOULD use **service components** and MAY create unlimited number of your own components.

For the top-level controller component you SHOULD use the following class, but you MAY use any controller variant suitable for a framework of your choice (if you had decided for the code to run):

```
class WatchController
{
    public function getByIdAction($id)
    {
        // write underlying implementation according to the specification
    }
}
```

The id MUST be only positive whole numbers. The server SHOULD handle invalid values based on good REST API practices.

Task constraints

A user sends a request to the REST API to download information about a watch identified by an id. The system MUST first try to find the watch in cache (cache expiration is unlimited, ie. once a watch is cached it remains cached forever). When the watch is not present in the cache the system shall load the watch from a slightly *slower source*: either MySql database or an external XML file. Both of these sources are exposed as interfaces in the **required components** section. Should the *slower source* successfully respond with data it MUST be inserted into cache and returned to a user.

The actual *slower source* is not known during development of the API, thus you SHOULD provide a way to quickly switch from one source to another using a configuration. Also take into account that in future there may be other new data sources.

For simplicity you can consider caching to a file, but you MUST account for the future where caching mechanism might be exchanged for something more performant, such as Redis.

Required components

You MUST use the following components representing data sources within your solution of the task as data providers - you SHOULD NOT implement the interfaces. Excluding any of the following components will render your task incompleted.

You MUST NOT modify any of the components (both Required and Service). The following are considered modifications:

- replacing the **interface** keyword with a **class**,
- changing a name of an **interface** or a **class**,
- changing **method** input parameter(s) - i.e. adding, removing or changing data type,
- changing expected **method** return data type.

Hint: If you want the code to run, you actually have to implement the **MySqlWatchRepository** and **XmlWatchLoader** interfaces, but the task can be completed without doing so if you simply want to demonstrate your programming skills and do not want the code to run.

Both of the data sources contain exactly the same data set represented in different formats. But because of the evolution of the project each data source returns a different data type and specifies a method with a different name.

MySqlWatchRepository

```
interface MySqlWatchRepository
{
    /**
     * @param int $id
     *
     * @return MySqlWatchDTO
     *
     * @throws MySqlWatchNotFoundException Is thrown when the watch could
     *                                     not be found in a mysql
     *                                     database, eg. watch with the
     *                                     associated id does not exist.
     *
     * @throws MySqlRepositoryException    May be thrown on a fatal error,
     *                                     such as connection
     *                                     to a database failed.
     */
    public function getWatchById(int $id): MySqlWatchDTO;
}
```

Searches the database for a watch. If a watch with the associated id exists returns it as a **MySqlWatchDTO** (see below), or throws an exception when the watch does not exist.

MySqlWatchDTO

```
class MySqlWatchDTO
{
    /**
     * @var int
     */
    public $id;

    /**
     * @var string
     */
    public $title;

    /**
     * @var int
     */
    public $price;

    /**
     * @var string
     */
    public $description;

    /**
     * @param int    $id
     * @param string $title
     * @param int    $price
     * @param string $description
     */
    public function __construct(
        int $id,
        string $title,
        int $price,
        string $description
    ) {
        $this->id          = $id;
        $this->title       = $title;
        $this->price       = $price;
        $this->description = $description;
    }
}
```

XmlWatchLoader

```
interface XmlWatchLoader
{
    /**
     * @param string $watchIdentification
     *
     * @return array|null
     *
     * @throws XmlLoaderException May be thrown on a fatal error, such as
     *                            XML file containing data of watches
     *                            could not be loaded or parsed.
     */
    public function loadByIdFromXml(string $watchIdentification): array;
}
```

Obtains and parses a XML file containing data about watches. When no exception is thrown and the watch is not found, the null value is returned. Otherwise the method returns an associative array having the following structure:

```
[
    'id'    => 'INTEGER',
    'title' => 'STRING',
    'price' => 'INTEGER',
    'desc'  => 'STRING'
]
```

Service components

You MAY use any of the following components. Not using them might still lead to the task being considered completed.

```
class MySqlRepositoryException extends RuntimeException { }
class MySqlWatchNotFoundException extends MySqlRepositoryException { }

class XmlLoaderException extends RuntimeException { }
```