# Pull requests

This project does continuous integration with [Travis](https://travis-ci.org/WhichBrowser/Parser-PHP/) and [Coveralls](https://coveralls.io/github/WhichBrowser/Parser-PHP). This means that every pull request must have the proper PSR-2 code standards and working unit tests. Please make sure your pull request passes all tests before submitting it.

To test your coding style run:

```

vendor/bin/phpcs --standard=PSR1,PSR2 src/

```

Then try the PHPunit unit tests:

```

vendor/bin/phpunit

```

And finally to test the WhichBrowser analyser:

```

php bin/runner.php check --show

```

# Unit tests

This project uses both unit tests for the public API and it's own testing tool for testing if the analyser properly works.

If you contribute to the public API, please add revelant tests. The API currently has 100% code coverage and any PR should maintain this number. The tests can be found in the `tests/unit/Model/` directory and are checked using PHPunit. This should be fairly simple.

To run the unit tests:

```

vendor/bin/phpunit

```

The data for the analyser can be found in the `tests/data/` directories. These are Yaml files containing an entry for every single test that is executed. Currently these directories contain 5000 different tests resulting in almost 100% code coverage. Every entries contains a number of keys, which represent the different parameters of the Parser object and an extra key which contains the result. If you add a new test, you should not provide the result yourself, but instead just provide the parameters and let the testing framework generate the result.

A very simple test entry looks like this:

```

-

headers: 'User-Agent: Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0; Xbox)'

```

Once the results if generated by it will look like this:

```

-

headers: 'User-Agent: Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0; Xbox)'

result: { browser: { name: 'Internet Explorer', version: '9.0' }, engine: { name: Trident, version: '5.0' }, device: { type: gaming, subtype: console, manufacturer: Microsoft, model: 'Xbox 360' } }

```

To run the tests:

```

php bin/runner.php check --show

```

And to rebase the tests to generate the new results:

```

php bin/runner.php rebase

```

# Data files

A large part of the device model information is maintained in the data files. You can find these files in the `data/` directory. The files are little more than just arrays with the model number as key and another array a the value for each entry. The value contains a manufacturer, a model name and optionally the device type. If no device type is provided we assume it is a mobile phone. The device types that you are allowed to use are defined in the `src/Contstants/DeviceType.php` file.

```

DeviceModels::$ANDROID\_MODELS = [

'LG-L30L' => [ 'LG', 'F70' ]

];

```

In the example above, the model number is `LG-L30L`, the manufacturer is LG and the model name is F70. And because no device type is provided, it is a mobile phone.

Things can get a bit more complicated, because the key of the array can also be a regular expression. To mark this, the final character of the key must be an exclamation mark.

```

DeviceModels::$ANDROID\_MODELS = [

'LG-H8(10|11|12|15|18|19)!' => [ 'LG', 'G4' ]

];

```

Because the G4 uses multiple model numbers, like LG-H810, LG-H811, LG-H812 and so on, we do not want to include a seperate line for every single permutation. So we match all different model numbers with just a single regular expression.

The actual regular expression that is executed will look like this: `/^LG-H8(10|11|12|15|18|19)/iu`.

As an optimalisation it is also possible to use subsets. One regular expression will be tested to see if the whole subset must be skipped or looked at. A subset is marked by two exclamation marks.

```

DeviceModels::$ANDROID\_MODELS = [

'LG-F[0-9]{3,3}!!' => [

'LG-F320!' => [ 'LG', 'G2' ],

'LG-F340!' => [ 'LG', 'G Flex' ]

]

];

```

And finally, it is also possible to use backreferences to match multiple models at the same time:

```

DeviceModels::$ANDROID\_MODELS = [

'Aspire ([0-9]{4,4}[A-Z]?)!' => [ 'Acer', 'Aspire $1', DeviceType::DESKTOP ]

];

```

This will both match the "Aspire 5750G", the "Aspire 4738G" and many more similar models.

On more thing you need to be aware of, all of the keys are not tested against the raw value extracted from the useragent string, but instead is pre-processed. The code for this can be found in the `DeviceModels::cleanup()` function in the `src/Data/DeviceModels.php` file.

Indices

-------

In order to reduce the time needed for model lookups, WhichBrowser uses an index. These files are located in the `data/indices/` directory and are automatically generated by the `update-indices.php` script. This script will evaluate every single regular expression and will try to create a key for every single possible value that matches the regular expression. If the regular expression is too complicated or has too many possible values it will place it in a secondary list of \*difficult\* regular expressions.

After making a change to the data files you \*\*need\*\* to regenerate the indices, otherwise the new device model will not be picked up by WhichBrowser.

To update the indices of all relevant data files:

```

php bin/update-indices.php --all

```