# How I learned to stop worrying and love testing

*(because I had no other choice)*

## Programme

- ➔ Purpose of testing
- ➔ Plan ahead of implementation
    - ◆ Test-Driven Development
- ➔ Benefits of extensive testing
- ➔ Obstacles and misuses
- ➔ Automated vs. manual testing

## Why should I test my code?

## What testing is about

- ➔ Confirmation of the known
  - ◆ Check for proper and improper executions of an application
- ➔ **Documentation** of system behaviour
- ➔ Agile Modeling (AM): "test with a purpose"
  - ◆ **Why you are testing** something and **on what level** its need to be tested
- ➔ **You should be in control**, not Skynet
  - ◆ Plan for future changes and improvements

**Testing a system** is more important than having a **perfect system**

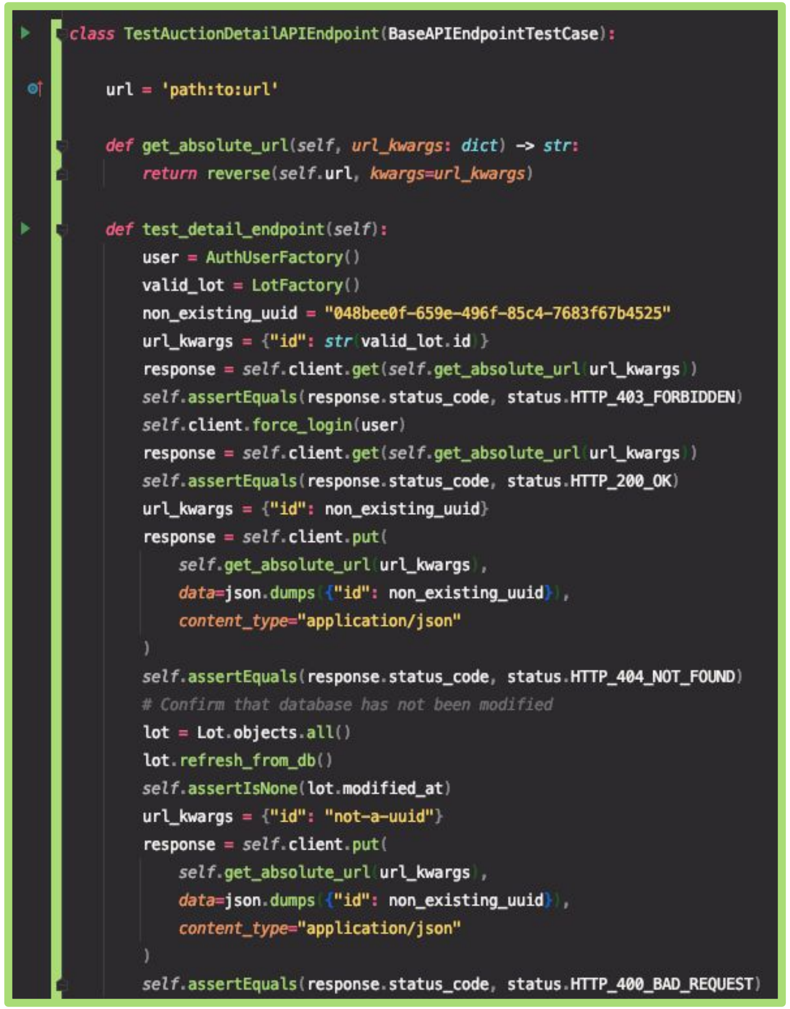
## Why is testing so painful?

## What makes testing harder and slower?

- ➔ Changing or **adding too many tests at once**
  - ◆ Incremental implementation
  - ◆ Modular components
- ➔ **Unclear purpose** (tests don’t make sense)
  - ◆ Checking for multiple behaviours at once
  - ◆ Split test into more scenarios

## What makes testing harder and slower?

- **Hard to change** or extend
    - Split and refactor existing tests
    - Change or remove mocks and patches
- **Functionality is hard to test** (most common)
    - Symptom of a component in need for re-architecture
    - Use appropriate testing tools
    - Improve your **testing skill-set**
    - Test from a higher level perspective

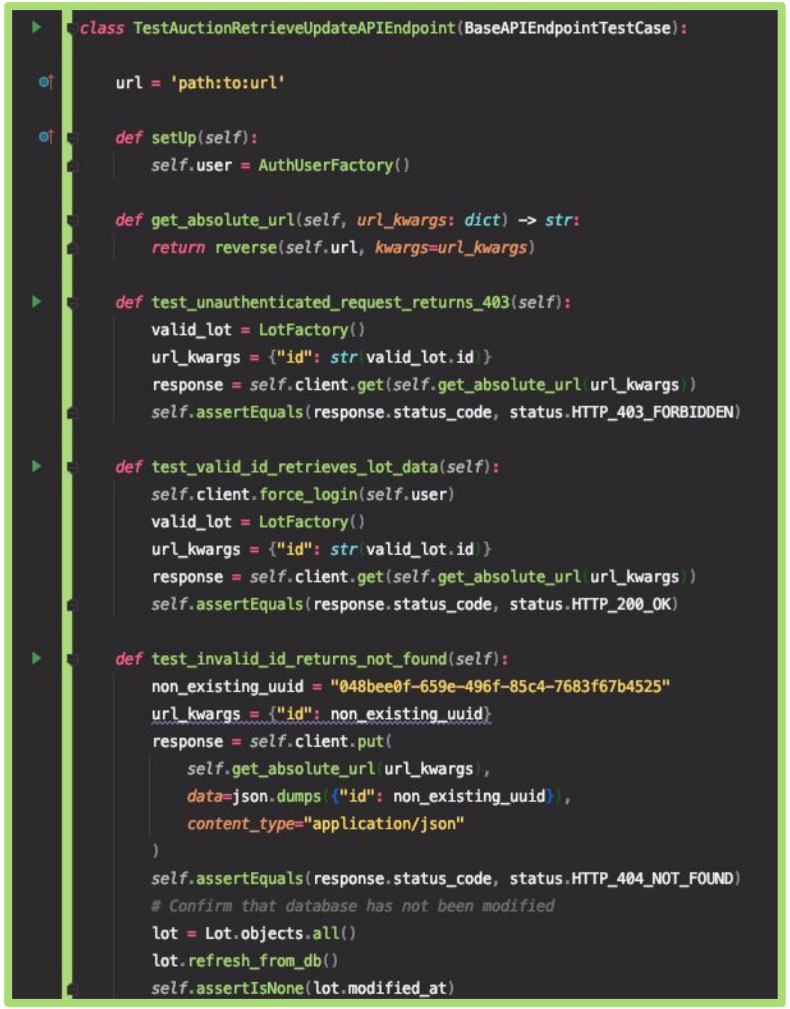


```python
class TestAuctionDetailAPIEndpoint(BaseAPIEndpointTestCase):

    url = 'path:to:url'

    def get_absolute_url(self, url_kwargs: dict) -> str:
        return reverse(self.url, kwargs=url_kwargs)

    def test_detail_endpoint(self):
        user = AuthUserFactory()
        valid_lot = LotFactory()
        non_existing_uuid = "048bee0f-659e-496f-85c4-7683f67b4525"
        url_kwargs = {"id": str(valid_lot.id)}
        response = self.client.get(self.get_absolute_url(url_kwargs))
        self.assertEquals(response.status_code, status.HTTP_403_FORBIDDEN)
        self.client.force_login(user)
        response = self.client.get(self.get_absolute_url(url_kwargs))
        self.assertEquals(response.status_code, status.HTTP_200_OK)
        url_kwargs = {"id": non_existing_uuid}
        response = self.client.put(
            self.get_absolute_url(url_kwargs),
            data=json.dumps({"id": non_existing_uuid}),
            content_type="application/json"
        )
        self.assertEquals(response.status_code, status.HTTP_404_NOT_FOUND)
        # Confirm that database has not been modified
        lot = Lot.objects.all()
        lot.refresh_from_db()
        self.assertIsNone(lot.modified_at)
        url_kwargs = {"id": "not-a-uuid"}
        response = self.client.put(
            self.get_absolute_url(url_kwargs),
            data=json.dumps({"id": non_existing_uuid}),
            content_type="application/json"
        )
        self.assertEquals(response.status_code, status.HTTP_400_BAD_REQUEST)
```



```python
class TestAuctionRetrieveUpdateAPIEndpoint(BaseAPIEndpointTestCase):

    url = 'path:to:url'

    def setUp(self):
        self.user = AuthUserFactory()

    def get_absolute_url(self, url_kwargs: dict) -> str:
        return reverse(self.url, kwargs=url_kwargs)

    def test_unauthenticated_request_returns_403(self):
        valid_lot = LotFactory()
        url_kwargs = {"id": str(valid_lot.id)}
        response = self.client.get(self.get_absolute_url(url_kwargs))
        self.assertEquals(response.status_code, status.HTTP_403_FORBIDDEN)

    def test_valid_id_retrieves_lot_data(self):
        self.client.force_login(self.user)
        valid_lot = LotFactory()
        url_kwargs = {"id": str(valid_lot.id)}
        response = self.client.get(self.get_absolute_url(url_kwargs))
        self.assertEquals(response.status_code, status.HTTP_200_OK)

    def test_invalid_id_returns_not_found(self):
        non_existing_uuid = "048bee0f-659e-496f-85c4-7683f67b4525"
        url_kwargs = {"id": non_existing_uuid}
        response = self.client.put(
            self.get_absolute_url(url_kwargs),
            data=json.dumps({"id": non_existing_uuid}),
            content_type="application/json"
        )
        self.assertEquals(response.status_code, status.HTTP_404_NOT_FOUND)
        # Confirm that database has not been modified
        lot = Lot.objects.all()
        lot.refresh_from_db()
        self.assertIsNone(lot.modified_at)
```

## Planning ahead of implementation

## Why should I plan for tests *before* the implementation?

- ➔ **Validate requirements and design**
- ➔ Verify **acceptance criteria**
  - ◆ Ensure it covers edge cases and spot inconsistencies
  - ◆ Map behaviour to existing components
  - ◆ **Anticipate replicating current system behaviour**
- ➔ Facilitate development iterations
  - ◆ Reduces **uncertainty** and makes it intuitive to develop modularly

Planning ahead for tests **services usability** as a result of **designing the interface** before the **architecture**

## Why would I test simple scenarios?

## Simple issues reflect into big problems

- ➔ Whilst Python looks easy and intuitive, **what you are doing is not trivial**
- ➔ Check for exposed vulnerabilities
  - ◆ Authentication tests: reduce potential for **data breaches**
  - ◆ Migration tests: prevent **data corruption**
  - ◆ Validation tests: avoid **server failure** from unexpected inputs
- ➔ Build **confidence** in your system

## How to get started?

## Method

1. Define testing scope (e.g. an API endpoint, a database migration...)
2. Determine test strategy and level of detail (unit test, system test...)
3. List all the things that can go wrong
    a. Create at least one test for each
4. Describe all details of component successful behaviour
5. Write and run tests until all cases planned for are
6. Measure line coverage > check if there any tests missing

## Guidelines

- Agree on a naming convention and stick to it
  - Make it easy to find which test corresponds to a component
  - Be concise and functionality-specific
- Remember to check for logging messages
- Abstract common logic into reusable components sensibly
  - e.g. API endpoint test classes are often useful
  - Common data set up (e.g. creating users and companies, setting permissions)

“Testing is an **exploration exercise**.

It requires domain knowledge,

focus and willingness to learn.”

– Amir Ghahrai (DevQA)

## Test-Driven Development

## Test-Driven Development: concept & goal

- Tests are developed first to specify and validate what the code will do
    - When a **test fails**, we have made progress: **start implementation**
- Avoid code duplication
- Make the code **clear**, **simple** and "**bug-free**"
- Guarantee coverage of all components
- Ensure your system **meets requirements**
    - Emphasise **production code** rather than **test case design**

## Benefits of reliable automated tests

## Development process & codebase maintenance

- ➔ **Instant feedback** easier to interpret errors
- ➔ Expand vocabulary of **libraries** and behaviour of **data types**
- ➔ Increase domain of the code implemented
- ➔ Powerful tool for **refactoring**: confirm code works as before
- ➔ **Safety checks** for new changes can quickly spot flaws
- ➔ Increased confidence in the system allow us to **experiment more**

## What should I be wary of?

## Be wary of

- ➔ **Inconsistencies between test and other environments**
    - ◆ Ensure dependencies are the same
    - ◆ Reduce configuration mismatch
- ➔ Conditional logic
    - ◆ A test should *always* have the same input and output
    - ◆ Confirm behaviour of different feature flags and environment settings
- ➔ **Keeping factories and fixtures up-to-date** with your database models

## Be wary of

- Testing third-party behaviour or code already tested somewhere else
- **Unmaintainable tests**
  - Testing too many fine details
  - Testing low-level outputs
- **Dependencies on other tests** (never do this)
  - Tests **should always present the same behaviour** whether they **run in parallel or one at a time**

## When should I write automated tests?

## Manual testing x Automated testing

- ➔ Successful delivery: both
- ➔ **Automated tests:** balance time x cost x effectiveness
  - ◆ **How many times** will we want to run this test?
  - ◆ Are there **impediments** to implement automated testing?
  - ◆ What is the cost of **maintenance** for this strategy?
- ➔ **Manual tests:** when human intelligence is required
  - ◆ Outline **test cases**, perform **exploratory testing** and prevent **false positives**

Test coverage should be enough to ensure **no error introduced is silenced**

## Be pragmatic

- **Code duplication is fine** if it makes tests easy to change or delete
- Set up the minimum data needed for testing
  - But **ensure the data is accurate** and production-like if suitable
- Emphasize encapsulation
  - **Avoid dependencies on other tests** and perform the setup from scratch
- Favour system tests in detriment of unit tests
- Be **simple** and descriptive: *document the setup needed for a determined state*

## Summary

## Summary

- ➔ **Plan for testing** at the time of feature design
- ➔ Add tests along with implementation, not at the end
- ➔ Ensure that both implementation and test cases are **modular** and follow **encapsulation** principles
- ➔ Aim for extensive test coverage over perfection

Special cases aren’t special

enough to break the rules,

*although practicality beats purity.*

- Zen of Python

## Sources

- ➔ https://devqa.io/test-automation-advantages-and-disadvantages/
- ➔ https://www.guru99.com/test-driven-development.html
- ➔ http://agiledata.org/essays/tdd.html
- ➔ https://www.softwaretestinghelp.com/why-do-you-like-testing/
- ➔ Team mates and mentors through my career

## Questions