

CI report tracking solution



Thesis

Häme university of applied sciences, Information and Communication Technology

Spring, 2019

Veli-Ville Elis Hietanen

Information and Communication Technology
Riihimäki

Author Veli-Ville Hietanen **Year** 2019

Subject CI report tracking solution

Supervisors Pekka Ahonen, Petri Kuittinen

ABSTRACT

In this thesis I will go through a method of implementation for enabling continuous integration tracking, follow-up, management and how to make best use of the data generated in the process. When I go through each type of software used, I will first explain how they are used and give some advanced practical examples.

The thesis project was based on a radio network software test report dashboard that is used to oversee the continuous integration process and the changes made to it during its introduction. Because of a confidentiality agreement I will keep the practical examples brief and to the point.

A Web application was implemented using the Django web-framework and a PostgreSQL database. In the Django chapter of this paper I will explain how to setup a Django web application relying on practical examples from literature in the field, internet and Nokia Corporation. Please keep in mind that some of the examples are done in the Python version 2.7 and the Django version 1.10 though I will mostly focus on newer Python 3 and Django 2 versions. I will also introduce some competitive solutions.

Lastly, I explain how to manipulate and use the test data reported to the management web application in the PowerBI. This enables the creation of a live big picture using the PowerBI in combination with Django application API.

Keywords Continuous integration, Django, Microsoft PowerBI, Python.

Pages 27

Tieto- ja viestintäteknikka
Hämeen ammattikorkeakoulu

Tekijä Veli-Ville Hietanen

Vuosi 2019

Työn nimi CI report tracking solution

Työn ohjaaja/t Pekka Ahonen, Petri Kuittinen

TIIVISTELMÄ

Tässä opinnäytetyössä käyn läpi yhdenlaisen toteutuksen jatkuvan integraation mahdollistamiseen, seurantaan ja siitä saadun datan hyödyntämiseen hallinnossa. Käyn ensin läpi käytetyn tekniikan, jota seuraa esimerkkejä käytännön toteutuksista.

Opinnäytetyö perustuu Nokia Networks liiketoimintayksikön radioverkko-ohjainten ohjelmistotestauksen jatkuvan integraation valvontaohjelmistoon, sekä siihen tehtyihin muutoksiin. Salassapitosopimuksen takia joltain osin käytännön esimerkit ovat pintaraapaisuja joista selviää vain haluttu käyttötarkoitus.

Valvontaohjelmisto on tehty käyttäen Django web-kehystä, sekä PostgreSQL tietokantaa. Django osiossa selitetään Django verkko-sovelluksen luonti tukeutuen käytännön esimerkkeihin internetistä sekä Nokialta. Huomioitavaa on, että jotkin esimerkit ovat Python 2.7 versiota ja osa 3.0 tai sitä uudempia, mutta pyrin keskittymään työssä Python 3.0 ja Django 2.0 versioihin. Esittelen myös kilpailevia ratkaisuja sekä perustelen miksi juuri Django/Postgresql on käytössä.

Viimeiseksi syvennyn valvontaohjelmistoon raportoidun datan hyväksikäyttämiseen isomman reaaliaikaisen kuvan luomisessa hyödyntäen valvontaohjelmiston ohjelmointirajapintaa, sekä Microsoft PowerBI ohjelmiston raportteja ja ”kojelauta” näkymää.

Avainsanat Continuous integration, Django, Microsoft PowerBI, Python.

Sivut 27

CONTENTS

1	INTRODUCTION	1
2	COMISSIONER	1
3	CONTINUOUS INTEGRATION	2
3.1	Unit testing	3
3.2	Component testing.....	4
3.3	Integration testing.....	5
3.4	System testing	6
3.5	Acceptance testing	7
4	DJANGO	7
4.1	Other options	7
4.2	Editor and integrated development environment.....	7
4.3	PyPi (Python Package Index)	8
4.4	Virtual environments	8
4.5	Underlying structure of the Django web framework.....	9
4.6	Database structure models	13
4.7	Django REST framework.....	13
4.8	Testing of Django functionalities	16
5	MICROSOFT POWERBI	17
5.1	Using testing data to enhance management level decision making	17
5.2	Using power BI in conjunction with Django REST API.....	18
5.3	Report making and publishing using the PowerBI desktop	21
5.4	Keeping data up to date	22
6	CONCLUSION	23
	REFERENCES.....	26

1 INTRODUCTION

This thesis was commissioned by Nokia corporation. The target was to introduce a tool that would illustrate the product testing progress. It is used to visualize the results reported by radio network controller software testers. The application was originally developed for LTE network testing so upgrading and modifying the application were necessary.

After the initial changes there was a migration from the Eucalyptus cloud services to Openstack. When the migration was completed I continued the integration adding new features that were requested by my line manager. These features and integration objectives included, though were not limited, to the following:

- Migrating from Eucalyptus to Openstack cloud platform and updating related documentation in the process
- Planning and creating new features requested by the CI-manager.
- Modifying Admin and API views to comply with the release changes and PowerBI.
- Working with test automation teams helping them to setup test automation reporting.
- Changing unit tests to comply with the new features.
- Different bug fixes

2 COMISSIONER

Nokia corporation started as a paper mill in 1865 and steadily grew until it faced its first bankruptcy in 1910, Nokia was saved when neighbouring Suomen gummitehdas bought the company to secure its energy supply. The name Nokia originally comes from the name of the municipality, where the factory of Suomen Gummitehdas was situated.

Nokia took its current form when three companies Suomen kumitehdas, Suomen kaapelitehdas and Nokia Ltd. were fused into one company called OY Nokia Ab. Nokia unveiled its first VHF-phone in 1964 and its first NMT-phone in 1982. As a Joint venture with Finnish government they started to develop mobile networks from the year 1977 in a company named Televa.

After Microsoft bought Nokia telephone operations in 2013 Nokia re-focused on Mobile networks and acquired Alcatel-Lucent to boost its know-how in this sector, timeframe for this and other acquisitions can be found from Figure 1. (Nokia, 2018)

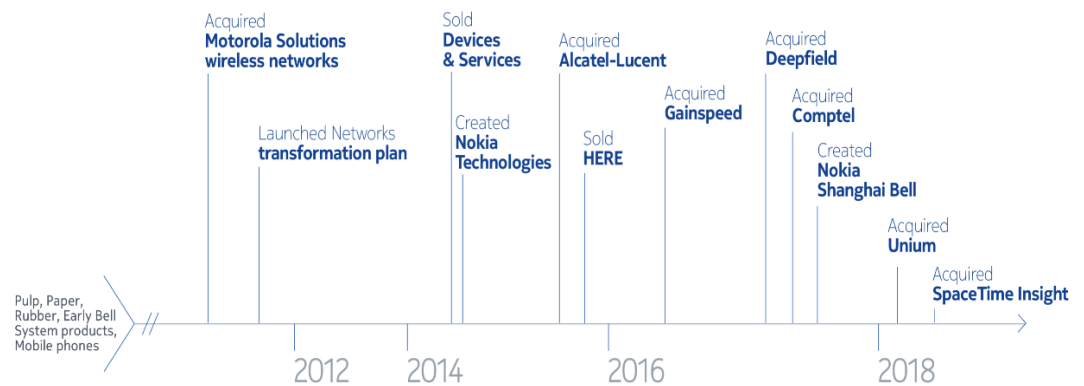


Figure 1. History of Nokia Corporation (Nokia, 2018).

3 CONTINUOUS INTEGRATION

Continuous integration is a software development technique where a team of developers share one repository that they integrate code into. These integrations are always verified by automation allowing bugs to be caught earlier, this in turn allows developers to handle faults earlier decreasing development cycle time and increasing product quality.

In practise basic workflow follows these steps. At first a developer checks out a working copy from the repository and makes the changes needed. After making the changes its important to take into consideration unit tests and update them if needed. When the developer is ready to push the code back to the repository an automated build is run on the developer's computer, if this succeeds the build is ok to be pushed back into the repository where it is automatically built and tested one more time.

If another developer has pushed changes to the repository while the other developer is still working on the code causing a conflict in the push it is the responsibility of the developer, whose build fails, to fix the build and resolve merge conflicts. All this aims to make integrating new changes to code mundane as everyone has the newest codebase and the integration is done daily (Fowler, 2006).

Testing levels are used to clarify and structure major software testing environments to counter testing repetition and overlap. The structure also helps to identify untested or poorly tested areas. Every testing level represents a phase in a basic software development life cycle (Tryqa, what are software software testing levels, 2018). Figure 2 illustrates different software testing methods and their related software delivery steps.

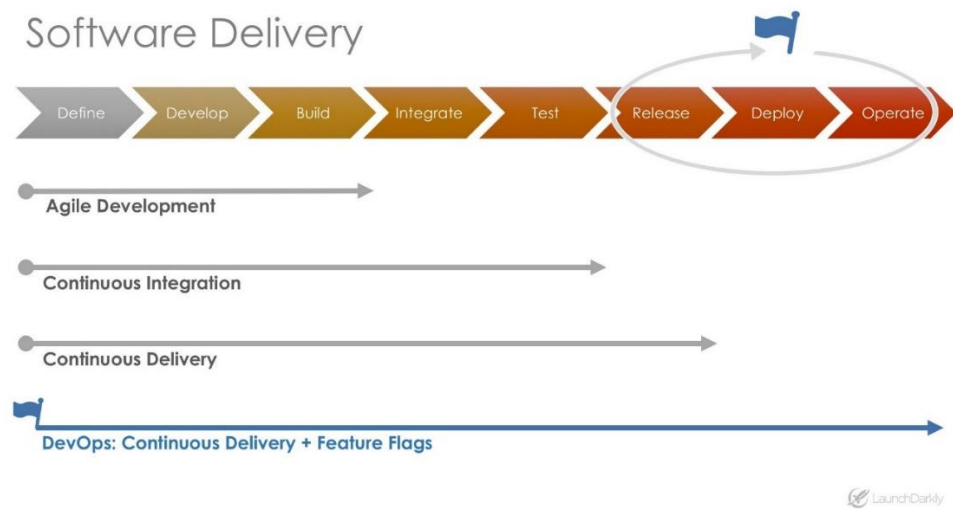


Figure 2. Difference in CI/CD/DevOps visualized (Baker, 2016).

3.1 Unit testing

Unit testing includes the smallest batch of code that can be executed on its own. These test cases are usually created and executed by the developer of that block of code. This in turn streamlines the debugging process by identifying the function where the problem occurs.

To make unit testing work the developers need to make code functions that minimize interdependency which means new code does not break legacy codes that often. In short unit testing is used to test against program specifications (Tryqa, Unit testing, 2017). Figure 3 shows the steps needed for running successful unit tests.

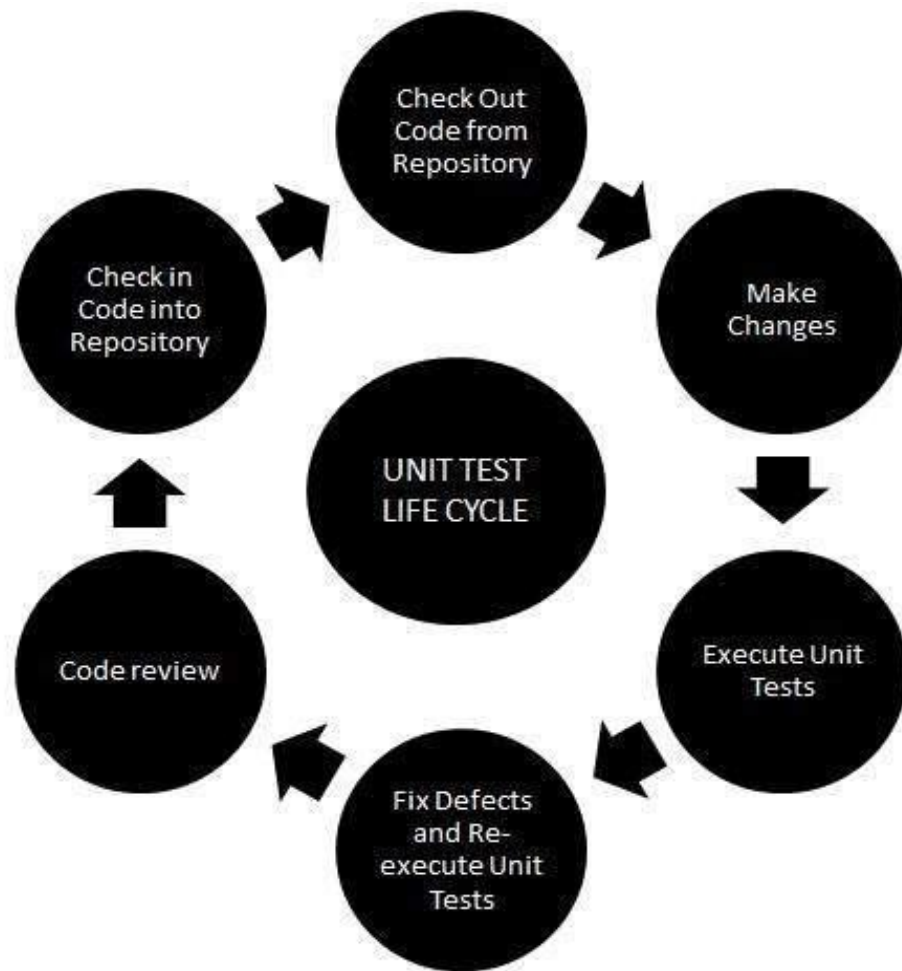


Figure 3. Unit testing and component testing life cycle diagram (Tutorial Point, n.d.).

However excessive unit testing is not recommended as it increases the load for developers and can easily result in a situation where the developer uses same or even more time writing the unit tests than the actual code. There have not been any large-scale studies that finds this type of development (Test Driven Development) to lower the amount of errors or the development time in any given software (Coplien, n.d.).

3.2 Component testing

When you test a bigger functions or modules using code blocks that are individually developed and unit tested it is called component testing. In a software which has all its basic functions you can do component testing without stubs, but when there still are some unfinished dependencies. Component testing can be isolated by simulating the input or the function needed (Tryqa, Component testing, 2017).

As an example, a function that sends an email notification, when a software build is promoted, is tested solo by simulating a trigger without function i.e. instead of sending an email with some very basic content you replace it with a log file input or indicate test successful in some other way if the email function runs successfully.

3.3 Integration testing

Integration testing includes testing interaction between two or more components. This is a vital testing area which every tester is recommended to have good knowledge about. Integration testing is done by team of testers or an individual tester.

As an example, you need integration testing to test functionality between different modules in a webstore (user login, cart, billing etc.) developed by different people. Integration testing is used for testing against technical specifications. International software testing qualifications board states three integration testing methods: Big Bang, top down integration and bottom up integration.

When all the different modules are integrated and tested together at the same time, as illustrated in figure 4, it is called the Big bang testing. This is not generally used, especially in bigger projects, because it is time consuming and it is hard to determine which of the modules are failing and why they are failing. For smaller projects this is the fastest way to verify that everything works as intended.

Big-Bang Approach

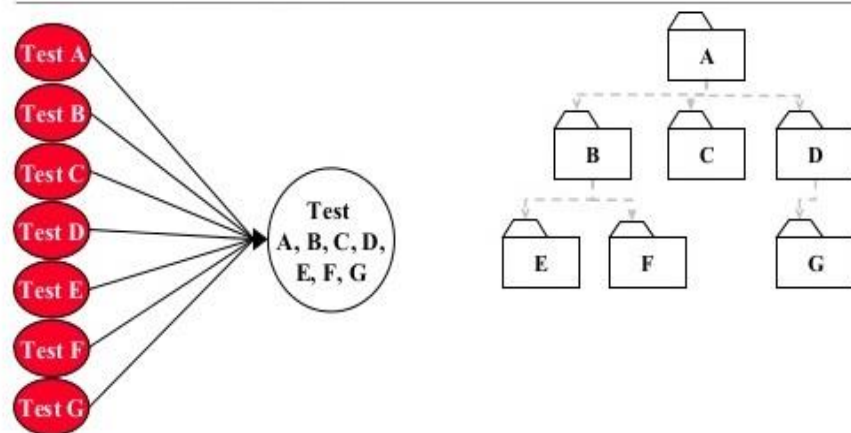


Figure 4. Diagram of Big bang testing (Patil, 2017).

Top down integration follows real life functionality where you are testing one module at a time. Modules that are not being tested but are needed for functionality are replaced with stubs. The biggest drawback of this method is the verification of full basic functionality later in the development and testing cycle. Examples of testing layers are shown in figure 5.

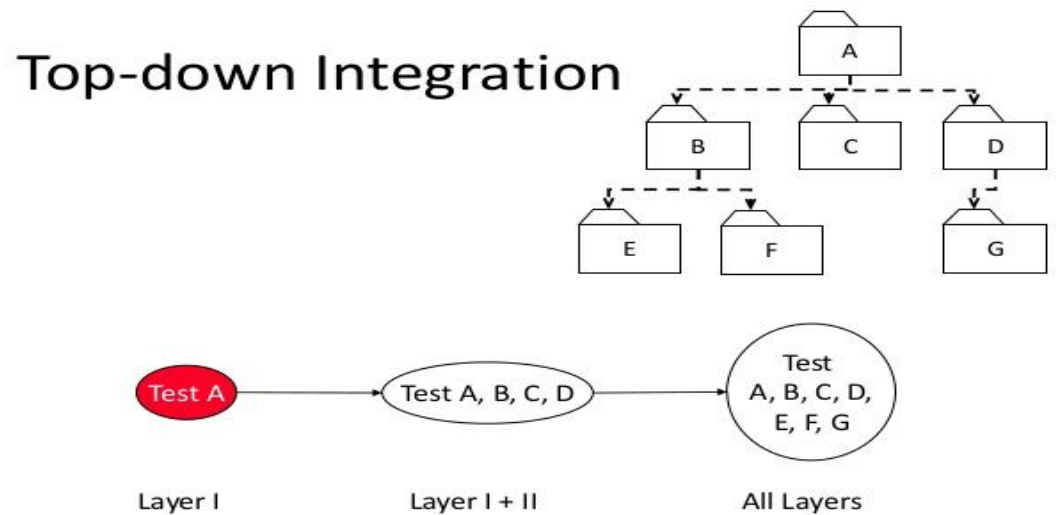


Figure 5. Top-down integration (Slideplayer, n.d.).

Bottom up integration is similar to top down testing but follows opposite architectural structure and uses drivers to simulate the code which calls lower modules. This allows development and testing to be done parallel to one another and modifying modules to a client's needs without compromising higher level modules. The biggest drawback is seeing interface problems later in development. (Tryqa, Integration testing, 2017)

3.4 System testing

System testing is done when you verify the whole product or the system functionality. The scope of the testing depends on what the product is being tested for, such as risks, use cases, requirement specifications or system behaviour which are determined in functional specification (Tryqa, System testing, 2017).

System testing uses real life production environments and test teams separated from developers to minimize any biases. The code base needs to be passed through all the phases above before entering the system testing. A successful system testing or a functional testing phase needs to be planned out before it is started by specifying the testing times, the setups and the

areas where testing needs to be done more specifically than in other areas i.e. risk areas (ISTQB, 2015).

3.5 Acceptance testing

The acceptance testing includes Alpha and Beta testing, of which the Alpha testing is done by the developers and the Beta testing is done by the customer who is acquiring the product.

By integrating acceptance testing into CI loop you have a DevOps type environment setup which loops customer reported bugs back into development cycle lowering the time it takes to fix them. (Tryqa, Acceptance testing, 2017)

4 DJANGO

Django is a free web framework that enables developers to get rid of most of the web development related hurdles by streamlining the development and allowing you to focus on writing the actual application (Django project, 2018). For the last two decades most of the professional web projects have been done with some web framework (Makai, 2019).

4.1 Other options

Django is one of many Python-based web frameworks, other notable web frameworks include Flask and Pyramid which I will briefly introduce.

Flask differs from Django by being a smaller microframework which is fast and simple to setup. Microframework means it does not use components in which pre-existing external 3rd party libraries are providing basic functions. The most commonly used database is MongoDB (Python software foundation, 2018).

Pyramid is a Ruby-on-rails influenced web framework which was created when BFG was included in the pylons project (Chrism, 2010). It is integrated with wide range of SQL and NoSQL databases and makes wide use of 3rd party tools (Pylons project, 2019).

4.2 Editor and integrated development environment

Nano was my main editor during the project. I added the syntax highlighting to it by linking the files from `/usr/share/nano/` to `/.nanorc`. An example output for `cat ~/.nanorc` on my environment:

```
include /usr/share/nano/python.nanorc
```

```
include /usr/share/nano/html.nanorc  
include /usr/share/nano/css.nanorc
```

With these enabled I got syntax highlighting for my user for CSS, HTML and Python files.

For more complex functions I imported the files to JetBrains PyCharm. It was also discussed that this would be used to set up a more streamline development environment with integrated GIT however, because of time constraints this idea was ultimately dropped.

When I worked on the in-house testing software UI I ran the software on my Windows 10 laptop and worked on the code with Visual studio.

4.3 PyPi (Python Package Index)

The official third-party repository for Python is the Python package index. It allows Python users access to over 110 000 third party packages. PyPi is used as a default source by many package managers such as PIP.

PyPi was born out of the need to make installing third-party packages easier and more centralized. It was first introduced in the Python release 1.6.1 in September 2000. (PyPi, n.d.)

4.4 Virtual environments

Virtual environments are used to differentiate global site packages from site packages. This enables running multiple Django projects on the same hosts without version dependencies causing compatibility errors this is illustrated in figure 6.

When a virtual environment is made directories are created to store all the packages. After activating the virtual environment, PIP or some other Python package manager can be used to install the packages in to the virtual environment. Install and active the virtual environment using the following commands.

```
pip install virtualenv
```

```
mkdir venv  
cd /venv
```

```
virtualenv your_venv_name
```

To activate the created environment, you need to run the activate script with the following bash line.

```
source /venv/yourvenv/bin/activate
```

After this your bash line should show the name of your virtual environment indicating where the PIP is installing python packages.

```
(your_venv_name) [user@yourhost ~]#
```

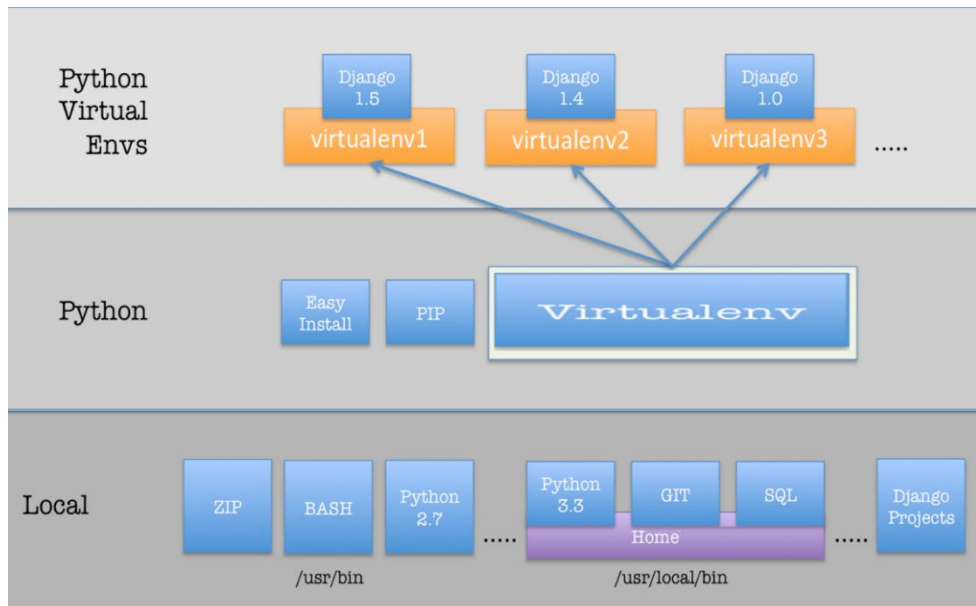


Figure 6. Virtual environments function visualized. (Batta, 2015)

4.5 Underlying structure of the Django web framework

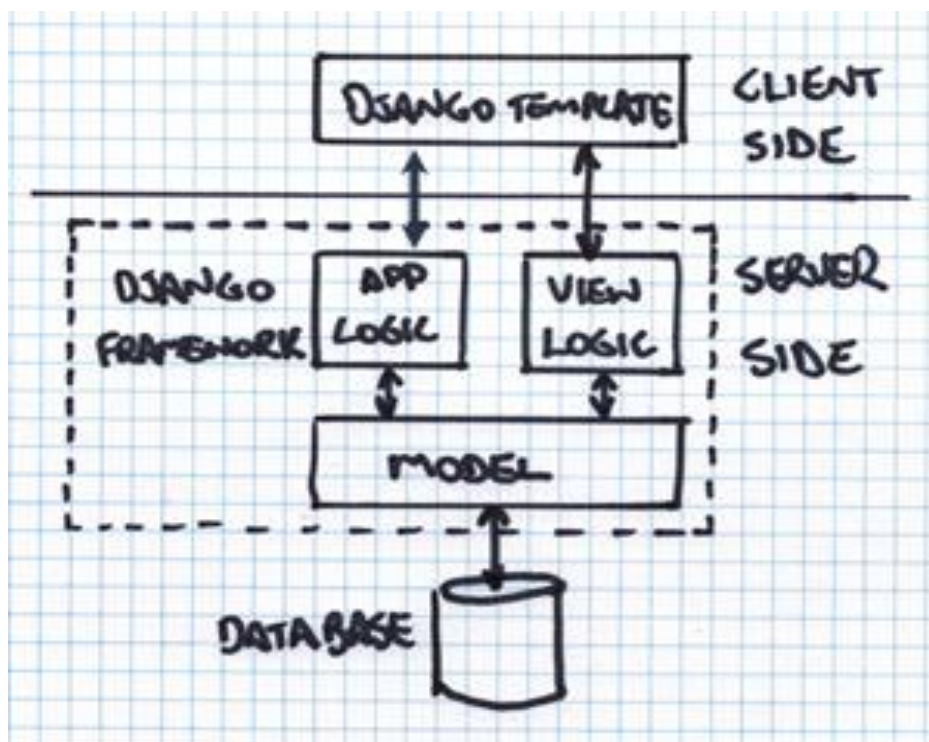


Figure 7. Basic structure of Django (Django book, 2018).

Django is built around managing these three sections, model, template, and view independently which is why this framework uses loose coupling and it is called MVC (Model, view, controller) framework as seen in figure 7. Loose coupling means that the application components have little or no info about the definitions of other components. (Django book, 2018)

The model is used to provide an interface with the database that houses all the data used by the application using ORM (object-relational mapping). The Django ORM tool is used for getting around using SQL that can hard to write and it can take your mind of the language you are actually writing your application in. When using ORM the database is mapped with objects as seen the figure 8 (Django-tutorials, n.d.).

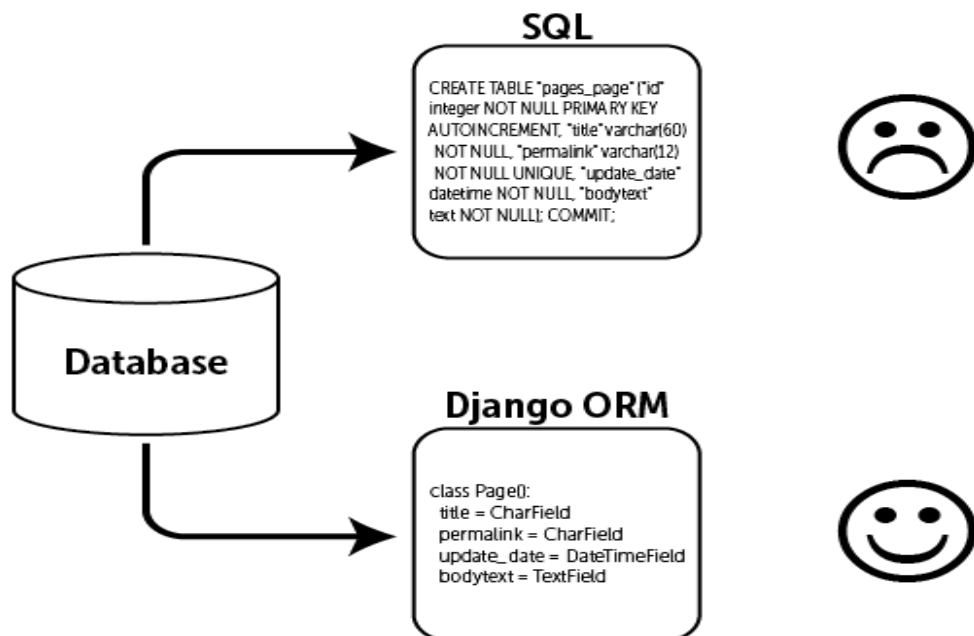


Figure 8. Django ORM diagram (Django-tutorials, n.d.).

Here is an example of a more complex model, the table generated can be found from figure 9:

```
class Test(models.Model):
    date = models.DateTimeField(null=True, default=timezone.now)
    name = models.CharField(max_length=420, null=True)
    result = models.CharField(max_length=300, default=None, null=True)
    execution = models.IntegerField(default=0, null=True)
    more_info = models.CharField(max_length=2000, default='{}', null=True)
    user_info = models.CharField(max_length=2000, default="", blank=True)
    network_elements = models.CharField(max_length=2000, default='{}', null=True)
    build = models.ForeignKey(Build, related_name='tests')
    testline = models.ForeignKey(Testline, default="", null=True, blank=True)
```

Column	Type	Not Null	Default	Constraints
id	integer	NOT NULL	nextval('collector_test_id_seq'::regclass)	
date	timestamp with time zone			
name	character varying(420)			
result	character varying(300)			
execution	integer			
more_info	character varying(2000)			
build_id	integer	NOT NULL		
testline_id	integer			
network_elements	character varying(2000)			
user_info	character varying(2000)	NOT NULL		

Figure 9. The table generated from the ORM above viewed in phpPgAdmin page.

The Django templates allow the separation of design from logic. This is especially important in bigger projects where programming and designing can be done by a different group of people. Usually they are written in HTML, but they support a multitude of text formats.

Using the Django templates allows web designers to create the frontend with placeholders for data, so web developers can add right Django variable tags after the backend is ready. This also makes it possible to use HTML pages that other people have designed allowing you to keep your focus on coding instead of designing. (Django-tutorials, n.d.)

An example template for a popup window:

```
{% block title %}
{{build.rlabel}} <span class='glyphicon glyphicon-time'></span>
{{build.date}} <br>
{{release}}
{% endblock title %}
{% block content %}
    {% if build|build_has_nomination:"_release" %}
        {% if product1 in product or product2 in product %}
            <a href='http://linktoreleasenotes/release_notes?product={{build.product}}&branch=trunk&release_name={{build.rlabel}}' target='blank' ><i class='glyphicon glyphicon-book'></i> ReleaseInfo</a><br>
            {% endif %}
        {% endif %}
        {%#monkeyfix!#}
        {%if "product1" == build.product.name or "product2" == build.product.name or "product3" == build.product.name %}
            <a href='{{build.url}}' target='blank' ><i class='glyphicon glyphicon-book'></i> Build location</a><br>
            {% endif %}
            <a href='{{build.url}}' target='blank' ><i class='glyphicon glyphicon-share-alt'></i> Build URL</a><br>
            {{build.id|build_popout_info}}
        {% endblock content %}
    {% endspaceless %}
```

The template from the projects testing environment above populates a popup window, as seen in figure 10, with important release links and other information. The Django version used is older, so you must go through each variable instead of just declaring them and doing the comparison in one go i.e. if “foo”, “bar” in “foobar”. Because advanced logic is a security issue it is disabled in Django templates along with executing any Python code and assigning a value to a variable.

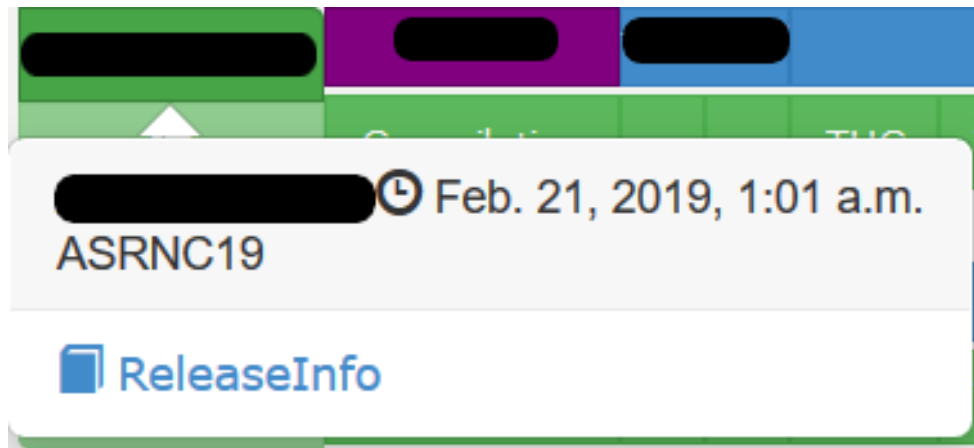


Figure 10. Popup window generated from the example template above.

The Django templates also employ parent child relations, so you do not have to repeat the HTML code. For example, if you use the same top navigation bar for all your subpages, they can inherit the same top navigation bar from one HTML file, an example of this can be found from figure 11, making maintaining and modifying it more convenient (Django-tutorials, n.d.).

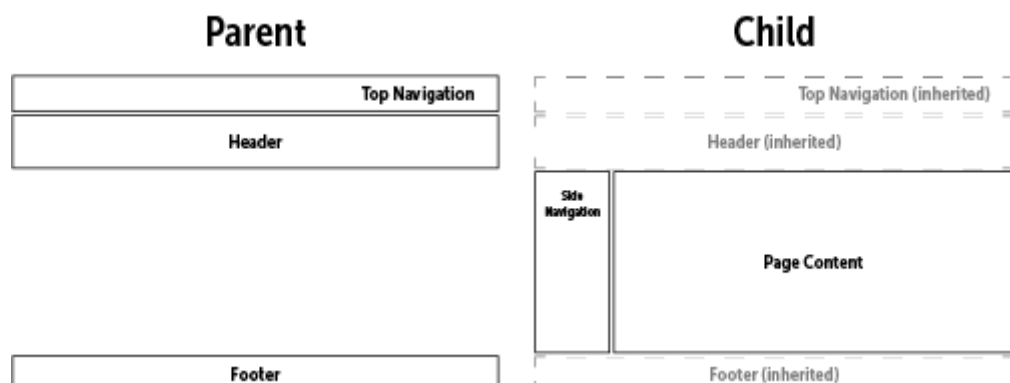


Figure 11. With a parent child relation, you can inherit parts of the website from the parent HTML file (Django-tutorials, n.d.).

Views are used to get data for your templates from a database or some other external data sources. Decisions are made in views about what data is or is not displayed in templates based on input or some other logic. Django built in views include 404, 500, 403 and 400 error display pages. Views as over simplified define what your application URLs file is pointing to (Django-tutorials, n.d.).

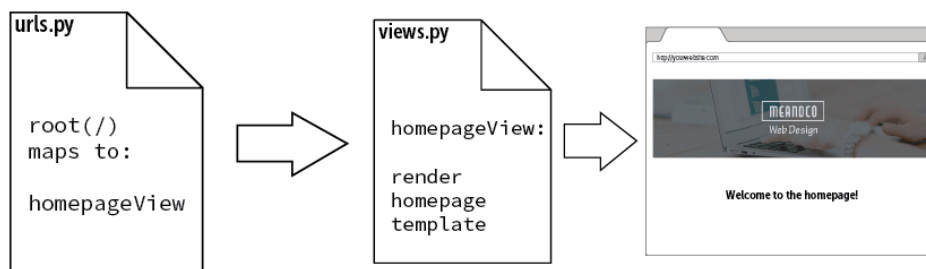


Figure 12. The views function simplified (Django-tutorials, n.d.).

4.6 Database structure models

Databases officially supported by Django are PostgreSQL, MySQL, SQLite and Oracle. Other databases can be connected to it with some third-party applications, it is however recommended to use the officially supported databases if possible (Big-nige, n.d.).

The Postgres database was inherited to my project from the original application that was used to visualize 4G testing status. It was probably originally chosen because it has the best compatibility with Django out of all the officially supported databases (David, 2012).

4.7 Django REST framework

REST (Representational State Transfer) is a design architecture used in API's originally defined by Roy Fielding in 2000. The advantage for using REST API is that it supports any content-type, though JSON or XML are the most popular choices. REST API provides unparallel flexibility compared to other SOAP or RPC (Remote Procedure Calls) APIs (MuleSoft, n.d.). Differences between these API's are more closely specified in table 1. Pure REST API's are built around six constraints:

Table 1. Comparison of the strengths and weaknesses of the three most popular API, REST being the most popular one followed by SOAP and RPC (Stowe, 2015).

SOAP	RPC	REST
Requires a SOAP library on the end of the client	Tightly coupled	No library support needed, typically used over HTTP
Not strongly supported by all languages	Can return back any format, although usually tightly coupled to the RPC type (ie JSON-RPC)	Returns data without exposing methods
Exposes operations/ method calls	Requires user to know procedure names	Supports any content-type (XML and JSON used primarily)
Larger packets of data, XML format required	Specific parameters and order	Single resource for multiple actions
All calls sent through POST	Requires a separate URI/ resource for each action/ method.	Typically uses explicit HTTP Action Verbs (CRUD)
Can be stateless or stateful	Typically utilizes just GET/ POST	Documentation can be supplemented with hypermedia
WSDL - Web Service Definitions	Requires extensive documentation	Stateless
Most difficult for developers to use.	Stateless	More difficult for developers to use.
	Easy for developers to get started	

Stateless means no client session state is ever kept in the server but instead it is always stored in the client side and is transferred with it hence the ST (State Transfer) in the acronym **REST**. This allows the service to be viably scaled to millions of users because resources are not spent on managing user sessions.

A Client-server is a concept of separating the client side from the server side allowing them to be developed independently. This means having the ability to make changes to the database or data structure without it effecting client-side templates and vice versa (MuleSoft, n.d.).

Caching is used on all cacheable data on client side. Responses are used to inform the client whether the data can be cached for later use or not. This allows you to lower the load on your server.

Layered System means you can use the layers to encapsulate older parts of the service from newer more used parts which in turn allows effective use of load balancing. When all the layers are properly secured a systemwide compromise is not that plausible.

Uniform interface means you don't change the communication interface between the client and the server instead of committing to using something like HTTP requests and JSON.

Code on demand is the only optional constraint out of the six for REST as defined by Roy Fielding in his doctoral dissertation. The idea here is to allow user to move code snippets via API. This is the least applied constraint

because it is optional and introduces some security issues with it. (Stowe, 2015)

The Django rest framework allowed me to make use of a flexible and powerful API based on the models of the application. It was used to exchange data between our in-house testing software and with PowerBI.

The REST framework was used to fetch correct software build and test related names when reporting testing status, as seen in figure 13, to the management application. This negated any change for case sensitive data to be typed wrongly by the tester.

The screenshot shows a web form titled "Reporting Portal". It contains several input fields and a table. The fields are: "Build Name" (dropdown menu with "ASRNC_REL_TEST-37000"), "Test Team Name" (dropdown menu with a redacted value), "Test Line" (dropdown menu with a redacted value), "Competence Area" (dropdown menu), "Status" (dropdown menu with "started"), and "More Info" (text input field). Below these fields are two checkboxes: "Submit Case Status to [redacted]" (unchecked) and "Submit Build Status to [redacted]" (checked). Below the checkboxes is a table with two columns: "Network Element" and "Version". The table is mostly redacted with a large black box. At the bottom of the table is a button labeled "+ Add Network Element". At the bottom of the form are two buttons: "Cancel" and "Submit".

Figure 13. A window for reporting the test case status to the Management application.

4.8 Testing of Django functionalities

Test cases for this project were done before I started working with it, so I only did modification to existing tests. Because they were unit tests a model mommy library was used to create missing data before running the tests, such as adding a new build to the application via rest framework API.

```

class NewTestInstanceCreationTest(APITestCase):
    """
    Ensure we can create a new Integration object.
    """

    def setUp(self):
        p = mommy.make(Product, id=1, name='product_name')
        r = mommy.make(Release, id=1, name='release_name')
        t = mommy.make(Testline, id=1, name='test_Line', product=p)
        b = mommy.make(Build, id=1, rlabel='build_Name',
                       product=p, release=r, result='unstable')

    def test_create_test_run(self):
        url = reverse('api:test-list')
        data = {
            "name": "test_1",
            "result": "passed",
            "testline": "test_Line",
            "path": "/path/0/",
            "more_info": {},
            "network_elements": {},
            "execution": 0,
            "rlabel": "build_Name",
            "date": "2016-02-15 12:10:58.168992",
        }
        response = self.client.post(url, data, format='json')
        self.assertEqual(response.status_code,
            status.HTTP_201_CREATED, response.data)
        self.assertEqual(Test.objects.count(), 1)
        self.assertEqual(Test.objects.get().name, 'test_1')

```

Above there is a code snippet from a test used to make sure that you can add a new test via the API. The setUp function creates a build completed with Product and Release and to which the test is reported to.

The test_create_test_run function determines what data is sent to the API and what is the desired response from it. Fetching the created object is also tested with Test.objects.get().

5 MICROSOFT POWERBI

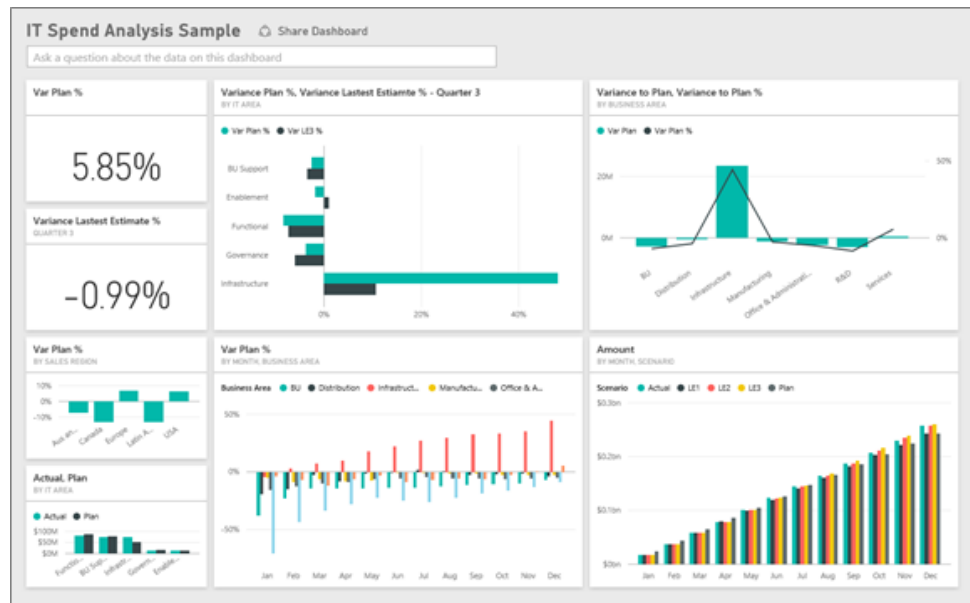


Figure 14. An example of a financial chart generated with Power BI (Microsoft, 2018).

Power BI is a Microsoft business intelligence tool used for visualizing data derived from big datasets. Even though its most common use is financial services as seen in figure 15, it also has some great use cases in CI management as well. You can either connect it straight to the database for a live connection or make use of the API to fetch the data you need.

5.1 Using testing data to enhance management level decision making

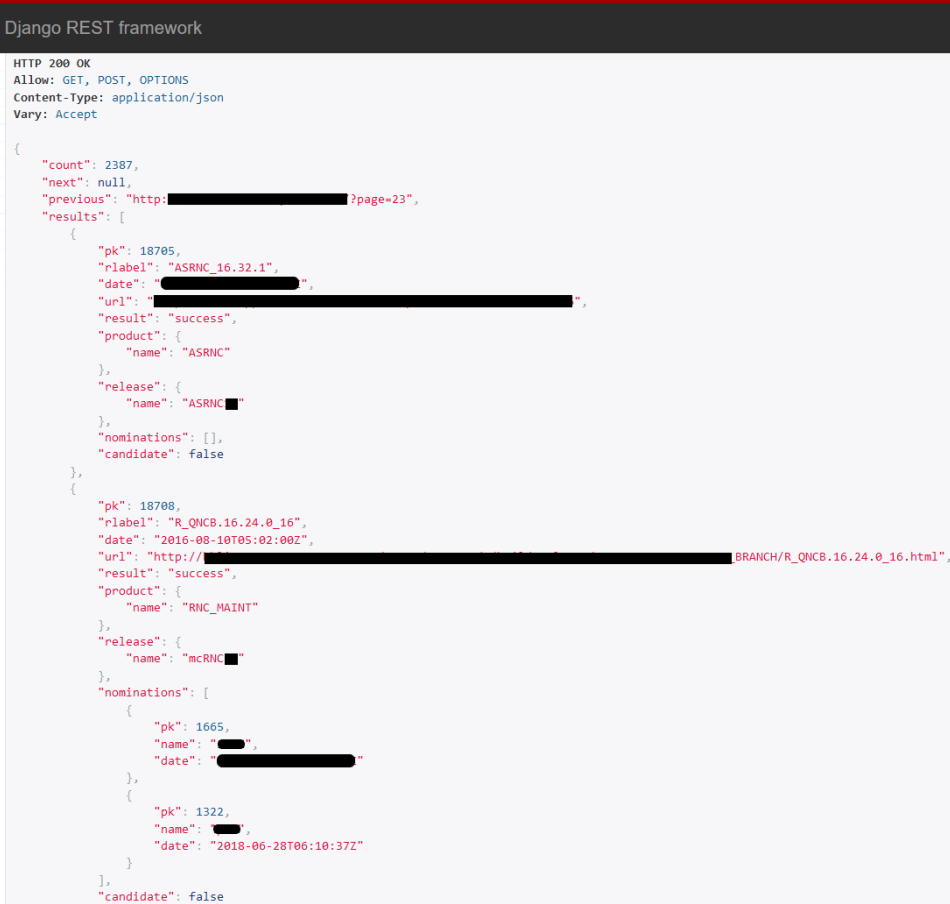
There are a few choices, either using libraries such as PyChart to generate data visualisation on the web application itself or by importing the reported data to a trusted 3rd party program that is already widely used and supported in your organization, in our case it is PowerBI.

The reason we leaned towards using Power BI was the fact that it would save a lot of development time while keeping the report quality and usability high. PowerBI is also integrated with the other Office 365 applications, so it was the obvious choice for us. We could create teams for applications and publish the reports to the right people easily.

5.2 Using power BI in conjunction with Django REST API

Because of network restrictions and problems related to PostgreSQL database access I opted to do the MVP (minimum viable product) with The Django rest API and the get data from web feature of PowerBI. This combination worked out great but, as the dataset grows it will probably be changed to a straight database connection to deal with the refresh speed issues and allowing live data review.

Before being able to create any reports, we need data. Getting data from a web page using PowerBI is straight forward. Some basic knowledge of DAX and PowerBI power query editor is all you need. All the reports were done using Power BI desktop, so I didn't need an on-premises-data-gateway until they were published to the PowerBI web application (Zhang, 2016).



```

Django REST framework
HTTP 200 OK
Allow: GET, POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
  "count": 2387,
  "next": null,
  "previous": "http://[REDACTED]?page=23",
  "results": [
    {
      "pk": 18705,
      "rlabel": "ASRNC.16.32.1",
      "date": "[REDACTED]",
      "url": "[REDACTED]",
      "result": "success",
      "product": {
        "name": "ASRNC"
      },
      "release": {
        "name": "ASRNC [REDACTED]"
      },
      "nominations": [],
      "candidate": false
    },
    {
      "pk": 18708,
      "rlabel": "R_QNCB.16.24.0_16",
      "date": "2016-08-10T05:02:00Z",
      "url": "http://[REDACTED]BRANCH/R_QNCB.16.24.0_16.html",
      "result": "success",
      "product": {
        "name": "RNC_MAINT"
      },
      "release": {
        "name": "mcRNC [REDACTED]"
      },
      "nominations": [
        {
          "pk": 1665,
          "name": "[REDACTED]",
          "date": "[REDACTED]"
        }
      ],
      "product": {
        "pk": 1322,
        "name": "[REDACTED]",
        "date": "2018-06-20T06:10:37Z"
      }
    },
    {
      "candidate": false
    }
  ]
}

```

Figure 15. An example output from the Django REST framework API.

Having a well-planned Django REST framework to work with made this process even easier to complete. Our CI management tool API displays data in JSON format across multiple pages. I had to do a function that would go through all the pages and stop when the page request returns null and a PowerBI query to invoke it.

The GetData function (Masson, 2014) :

```
(page as number) as table =>
let
    Source =
    Json.Document(Web.Contents("webapp/api/builds/?",
    [Query=[page=Number.ToText(page)]])),
    results = Source[results],
    #"Converted to Table" =
    Table.FromList(results,
    Splitter.SplitByNothing(),
    null,
    null, ExtraValues.Error)
in
    #"Converted to Table"
```

The GetData function works by giving it a page number that it fetches and transforms to a database table, allowing me to manipulate and filter the data that I need. The Json.document source is used because the web applications REST API serves the data in JSON format as seen in figure 16. There was a problem with using dynamic URIs (Uniform Resource Identifiers) because the PowerBI needs to validate the URI when committing it to the data sources refresh. This is why the query part is separated from the URI on line 3 (Varga, 2018).

The query that calls GetData function (Youtube Video, 2018) :

```
let
    Iterate_pages = List.Generate( ()=>
        [Result= try GetData(1) otherwise null,
        Page=1],
        each [Result] <> null,
        each [Result = try GetData([Page]+1) otherwise null, Page =[Page]+1],
        each [Result]),
in
    Iterate_pages
```

The Query function calls the GetData function with do-while-loop that iterates over the pages until it gets a null response. This means all the pages are fetched as a list of all the JSON data found inside the web page.

Calling the function with the query returns a list of all the pages. Expanding the pages allows me to pick the data I want to use and connect it with other data from the API using Power Query Editor and the manage relationships tool.

	ABC 123	Column1
1	Record	
2	Record	
3	Record	
4	Record	
5	Record	
6	Record	
7	Record	
8	Record	
9	Record	
10	Record	
11	Record	
12	Record	
13	Record	
14	Record	
15	Record	

Figure 16. A sample output from the GetData query.

Remodelling is the biggest problem when using a web data source. If this data would have been fetched using straight connection to the web applications Postgresql database, it would have copied the relationships used in the database structure. But this method allows us to modify the dataset based on exactly what we need, starting point for the modifications is shown in figure 16 and the end result is shown in figure 17.

Build_id	rlabel	Date	Result	Product	Release
21017	R_QNCB.16.24.2_39.10		success		
20860	R_QNCB.16.24.2_39.9		success		
20750	R_QNCB.16.24.2_39.8		success		
20549	R_QNCB.16.24.2_39.6		success		
19800	R_QNCB.16.24.2_21.6		success		
19566	R_QNCB.16.24.2_36		success		

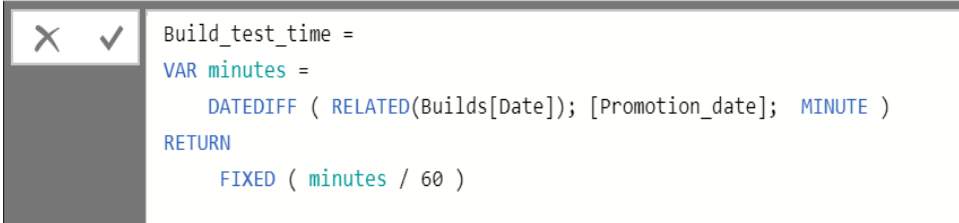
Figure 17. A list of extracted software build information after modifying the dataset.

Modifying and connecting the collected data is the most important part of using PowerBI. This will dictate how well your reports function, load and how fast they refresh.

5.3 Report making and publishing using the PowerBI desktop

Things that you want to monitor closely in a CI environment are the total time it takes to go through one CI loop, the number of uncovered errors and success/failure rates. The first software build status overview report I made had the general test statistics on one page. These statistics included how the builds performed and how long in the average it took for the CI loop to complete.

The Biggest hurdle when modifying the data to comply with my needs was calculating how long it took for any one of the three CI loop phases to pass.



```
Build_test_time =  
VAR minutes =  
    DATEDIFF ( RELATED(Builds[Date]); [Promotion_date]; MINUTE )  
RETURN  
    FIXED ( minutes / 60 )
```

Figure 18. The function used to calculate total CI loop time.

This DAX function, shown in figure 18, returns the amount of time passed from adding the build to giving the build a promotion indicating that the CI loop is completed. The function returns a fixed decimal indicating how many hours have passed. If for example the output is 32.50 it means that it took 32 hours and 30 minutes to complete the loop. This in turn can now be used in different visualisations and comparisons that weren't earlier possible with only the dates. The visualization of this data is shown in figure 19.

With this info we can already see where exactly the problems in the CI loop are if there are any. We can compare the test teams against each other and figure out if there is a need to cut down on the teams automated test-cases or room to add more. You can add more test cases as long as the average test time stays inside constraints. Also take into consideration how much time constraints effect error finding. It can be quite hard to find the middle road between catching all the software errors and staying inside the set timeframe.

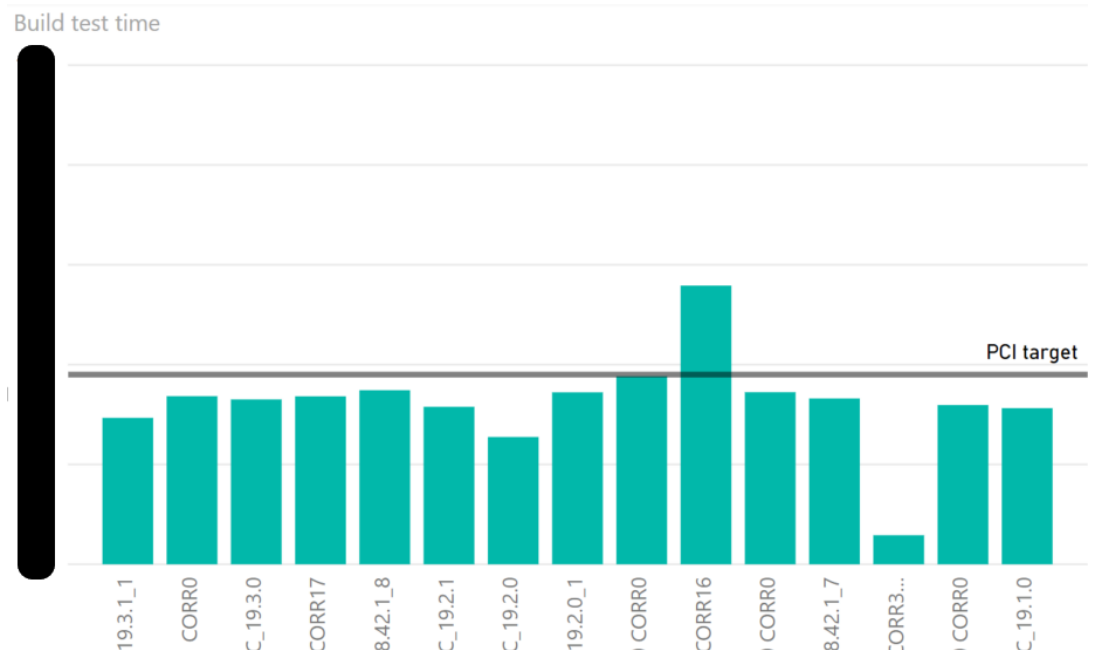


Figure 19. A visualization of the data fetched by Build_test_time function.

After the report is ready with the desired data and visuals it needs to be published from the PowerBI desktop to the PowerBI app space, so that the reports can be shared and viewed by others. I first made an empty application workspace and added a few colleagues to the group. It is important to review everyone's editing rights before adding anything to the application workspace to minimize changes for a mistake (Microsoft, 2018). After the application workspace was all set up and everyone had sufficient editing rights, I published the report from the PowerBI desktop to the app workspace (Microsoft, 2018).

5.4 Keeping data up to date

For this data to be valid it needs to be frequently updated. Because this dataset is inside a protected network a on-premises data gateway is needed when connecting to the API. An On-premises data gateway uses the computer it runs on as a port to connect to a dataset it has been assigned to implementing transport encryption and data compression on all levels. Simplified basic function diagram for the data gateway is shown in figure 20 (Microsoft, 2018)

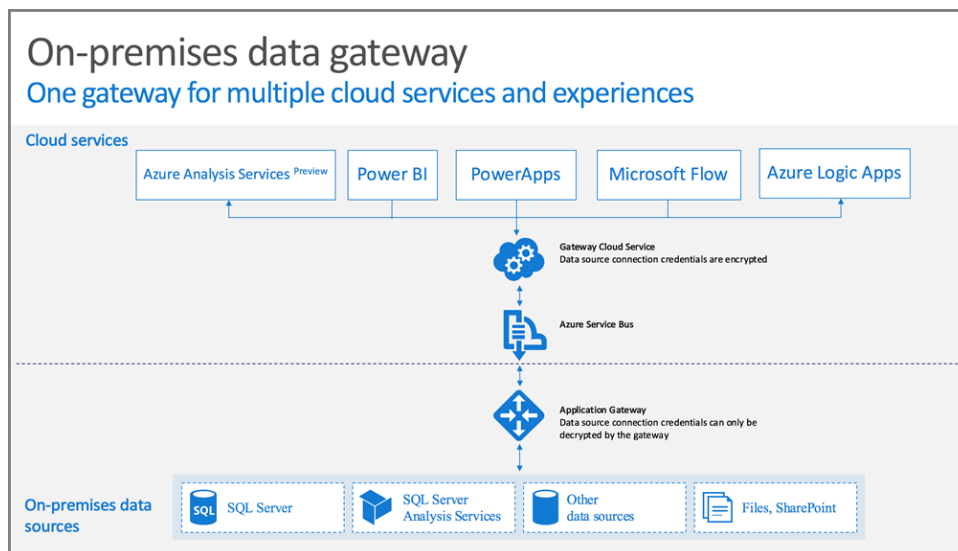


Figure 20. Data gateways basic function visualised by Microsoft docs (Microsoft, 2018).

There are some recommendations about installing a data gateway. It shouldn't be installed on a laptop because when the laptop is turned off or in sleep mode the gateway does not function causing scheduled refreshes to fail. Wi-Fi network can slow down refresh time and both personal and standard gateways require a 64-bit windows OS to work.

After I had a on-premises data gateway setup I could add scheduled refresh to my datasets after linking them to the gateway. If credentials are needed to access some of the data sources they are always sent encrypted. If a straight connection to Postgresql or any other database is used it is possible to have live data that is refreshed each time it is requested by the user. (Microsoft, 2018)

6 CONCLUSION

To save costs Nokia moved from the Eucalyptus to the Openstack environment and that meant migrating all the processes running in the old cloud to a new cloud environment including our CI management application. This was not a major change since there already was some documentation on setting up the application, the biggest difference being moving from the RHEL to Centos.

The migration took two weeks to complete because of my limited experience with the Django, Openstack cloud environment and because of some setup problems. For the end user the migration did not show in any way. I really got familiar with Openstack UI and its features by using a sandbox

version before doing the migration of our application. Also doing the setup for the Django application refreshed my bash skills.

The new features were planned by my CI-manager and most of these features were first raised by product managers and testers. My workflow regarding the new features started at a meeting with the CI-manager. He outlined what he wanted the feature to do and I implemented this to the sandbox version and tested it. If everyone was happy with the way it worked, it was to be implemented to the official version and the changes were committed to GIT.

Each new feature nearly always required changes in the application's own unit tests. These tests came with the software and were augmented by me to comply with the changes.

Supporting test automation teams with the application was a constant effort during the whole project. It mostly involved fixing curl syntaxes and helping with Jenkins job configurations. I also maintained a confluence page with all the essential documentation.

I also worked with the in-house test application front-end development team to enhance its use with the management application. The problem was with the test reporting UI that allowed testers to modify the names of the tested software builds and other case sensitive fields resulting in false test statuses and errors with the API. This was solved by fetching all the case sensitive data (software build names, test names) from the API and restricting the ability to change the fields.

My limited knowledge of JavaScript did create one major fault in the reporting process. Because JavaScript executes asynchronously it caused a situation where data is sent in a wrong sequence resulting in faulty test status emails. This was fixed by the team working on the software by instating a restriction that forced the data to be sent in a right order every time.

This project boosted my teamwork skills and got me familiar with a major software release related workflow. Modifications were made in JavaScript which I had not used for a long time.

Another goal was to be able to use this data at management level decision making. The tool chosen for this was PowerBI which is part of the Microsoft 365 software family. It has great visualization tools that can be used in the reports to display how different software products perform and how long it takes to test them.

Working with PowerBI also required some changes to how and in what way Django REST API displayed data. This part of the project got me familiar

with the REST principles and I got to know the Django REST framework more closely.

Working on this thesis allowed me to get more familiar with the radio network controller software testing and web frameworks, both of which I found very interesting. It also grew my software development and independent working skills.

The project and everything related to it was a big jump into the unknown for me. I did not have very much experience in the Django development before starting at Nokia. Challenges especially at the beginning were great but I rose to the required level with the help of great colleagues and my own interest. The In-house learning tool was also helpful allowing me to go through courses on my own.

The work I did for Nokia and the work I did during the software development courses at school were quite different. There was no unit testing done or even planned at any level during my courses also, neither was there any version control software in use. This was surprising to me because its integration to the courses would not have been too complicated. For me this project with Nokia was a great opportunity see what I lacked in skills and I really focused on getting better at these skills.

The IT industry is so complex and broad that these kind of trainee periods that challenge the student and give him enough responsibilities on real life solutions while providing a learning network are important.

References

- Baker, J. (2016, June 2). *Powering Continuous Delivery With Feature Flags*. Retrieved from <https://blog.launchdarkly.com/powering-continuous-delivery-with-feature-flags/>
- Batta, A. (2015, august 5). *Python virtual environment setup in ubuntu*. Retrieved from <https://2.bp.blogspot.com/-dDJh1jd8afc/WMbjQWZz3UI/AAAAAAAAACB8/KBUg5Gngr38l843Cg2nnnGrf-ReHg1oMACLcB/s1600/python-virtual-env.png>
- Big-nige. (n.d.). *Django Overview*. Retrieved from <https://djangobook.com/django-tutorials/django-overview/>
- Chris. (2010, November 9). *BFG-becomes-pyramid*. Retrieved from plope: <https://web.archive.org/web/20101113024153/http://plope.com/bfg-becomes-pyramid>
- Coplien, j. O. (n.d.). <https://rbc-us.com/>. Retrieved from Why-Most-Unit-Testing-is-Waste: <https://rbc-us.com/documents/Why-Most-Unit-Testing-is-Waste.pdf>
- David, S. (2012, March 2). *which-database-engine-to-choose-for-django-app*. Retrieved from <https://stackoverflow.com/questions/9540154/which-database-engine-to-choose-for-django-app>
- Django book. (2018). *django-structure*. Retrieved from <https://djangobook.com/mdj2-django-structure/>
- Django project. (2018). *Django project*. Retrieved from <https://www.djangoproject.com/>
- Django-tutorials. (n.d., May 12). *Django models*. Retrieved from <https://djangobook.com/django-tutorials/django-overview/>
- Fowler, M. (2006, may 1). *Continuous integration*. Retrieved from <https://www.martinfowler.com/articles/continuousIntegration.html>
- ISTQB. (2015, June 30). *system/functional testing*. Retrieved from <https://www.getsoftwareservice.com/integration-test-strategies/>
- Makai, M. (2019). *fullstackpython*. Retrieved from web-frameworks: <https://www.fullstackpython.com/web-frameworks.html>
- Masson, M. (2014, November 10). *Mat Masson PowerQuery blog*. Retrieved from <https://www.mattmasson.com/2014/11/iterating-over-an-unknown-number-of-pages-in-power-query/>
- Microsoft. (2018, August 6). *Microsoft powerBI documents create workspaces*. Retrieved from <https://docs.microsoft.com/en-us/power-bi/service-create-workspaces>
- Microsoft. (2018, May 6). *Microsoft PowerBI documents on-premises data gateway*. Retrieved from <https://docs.microsoft.com/en-us/power-bi/service-gateway-onprem>
- Microsoft. (2018, June 23). *Microsoft powerBI documents sample chart*. Retrieved from <https://docs.microsoft.com/en-us/power-bi/sample-it-spend>
- Microsoft. (2018, November 28). *Microsoft PowerBI documents what is powerBI*. Retrieved from <https://docs.microsoft.com/en-us/power-bi/desktop-what-is-desktop>
- MuleSoft. (n.d.). *rest-api-design*. Retrieved from <https://www.mulesoft.com/resources/api/what-is-rest-api-design>
- Nokia. (2018). *Nokia history flowchart*. Retrieved from https://www.nokia.com/sites/default/files/inline-images/history_nokia_no_title_3.png

- Nokia. (2018). *Our History*. Retrieved from <https://www.nokia.com/about-us/who-we-are/our-history/>
- Patil, K. (2017, February 23). *Integration-testing*. Retrieved from <https://bitwaretechnologies.com/wp-content/uploads/2017/02/bigbandimage.jpg>
- Pylons project. (2019). *Pyramid Introduction*.
- PyPi. (n.d.). *PyPi*. Retrieved from <https://pypi.org/>
- Python software foundation. (2018, May 2). Retrieved from Flask: <https://pypi.org/project/Flask/>
- Slideplayer. (n.d.). Retrieved from https://images.slideplayer.com/34/8360467/slides/slide_8.jpg
- Stowe, M. (2015). *Undisturbed REST*. San Francisco: MuleSoft.
- Tryqa. (2017). *Acceptance testing*. Retrieved from Acceptance testing: <http://tryqa.com/what-is-acceptance-testing/>
- Tryqa. (2017). *Component testing*. Retrieved from <http://tryqa.com/what-is-component-testing/>
- Tryqa. (2017). *Integration testing*. Retrieved from <http://tryqa.com/what-is-integration-testing/>
- Tryqa. (2017). *System testing*. Retrieved from System testing: <http://tryqa.com/what-is-system-testing/>
- Tryqa. (2017). *Unit testing*. Retrieved from <http://tryqa.com/what-is-unit-testing/>
- Tryqa. (2018). *what are software software testing levels*. Retrieved from <http://tryqa.com/what-are-software-testing-levels/>
- Tutorial Point. (n.d.). *Unit testing*.
- Varga, S. (2018, February 17). *Data inspirations blog*. Retrieved from <http://blog.datainspirations.com/2018/02/17/dynamic-web-contents-and-power-bi-refresh-errors/>
- Youtube Video. (2018, August 5). Retrieved from <https://youtu.be/vhr4w5G8bRA>
- Zhang, E. (2016, March 6). *PowerBI community forum*. Retrieved from <https://community.powerbi.com/t5/Integrations-with-Files-and/Using-a-REST-API-as-a-data-source/td-p/50400>