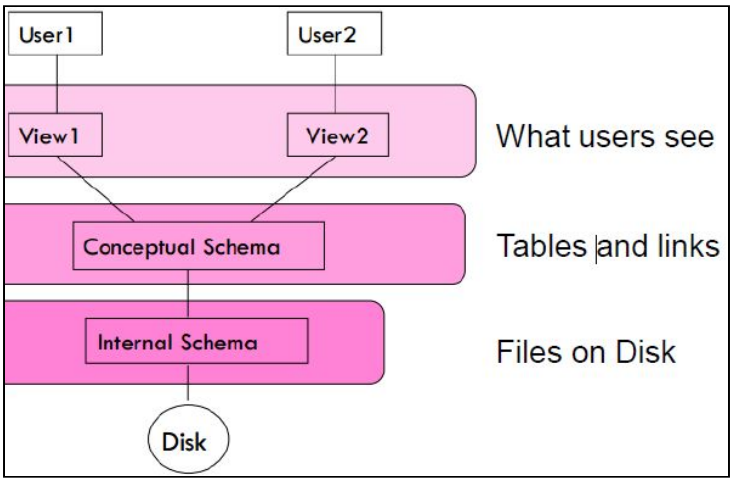
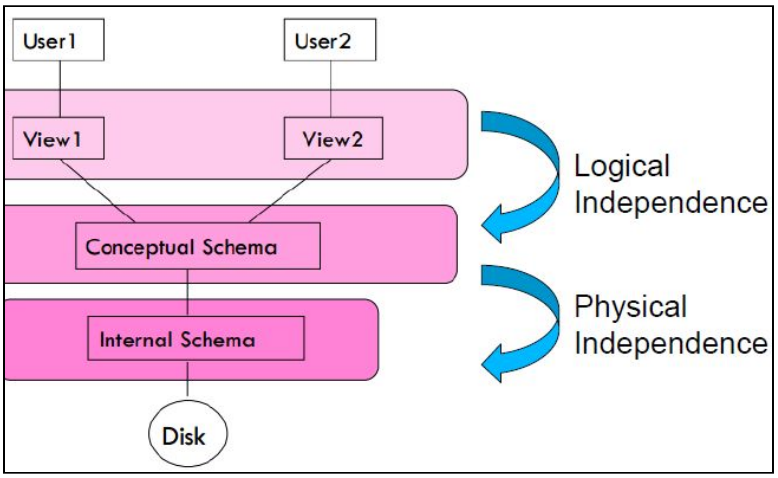
Resumos TPW - AT1

1. Web Application Architectures
1.1 - Data Independence in Databases





- Each level is independent of the levels below

- **Logical Independence**

Change the logical schema without changing the external schema or application programs

- Can add new fields, new tables without changing views
- Can change structure of tables without changing view

- **Physical Independence**

Change the physical schema without changing the logical schema

- Storage space can change
- Type of some data can change for reasons of optimization

What to learn:

- **Keep the VIEW:** (what the user sees)
- **Independent of the MODEL:** (domain knowledge)

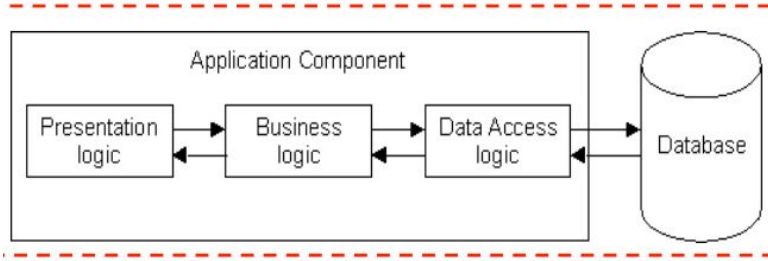
1.2 - N-Tier Architectures

N-Tier have the same components:

- **Presentation, Business/Logic, Data.**

N-Tier try to separate the components into different tiers/layers:

- **Tier:** physical separation
- **Layer:** logical separation

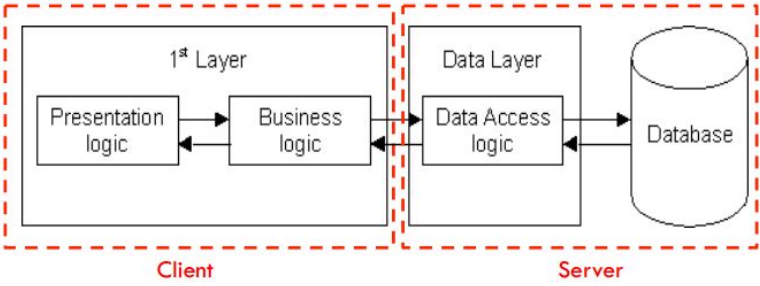


1 - Tier Architecture

All 3 layers are on the **same machine**: all code and processing.

Presentation, Logic, Data layers are **tightly connected**:

- **Scalability:** Single processor means hard to increase volume of processing
- **Portability:** Moving to a new machine may mean rewriting everything
- **Maintenance:** Changing one layer requires changing other layer



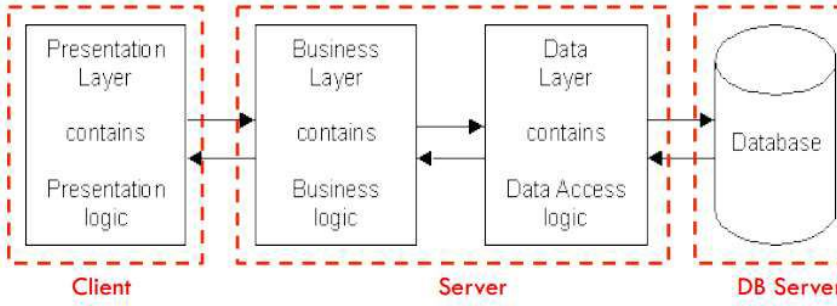
2 - Tier Architecture

Database runs on Server.

- **Separated** from client
- **Easy to switch** to a different database

Presentation and logic layers still **tightly connected:**

- Heavy **load** on server
- Potential **congestion** on network
- Presentation still **tied** to business logic



3 - Tier Architecture

Each layer can potentially run on a **different machine**.

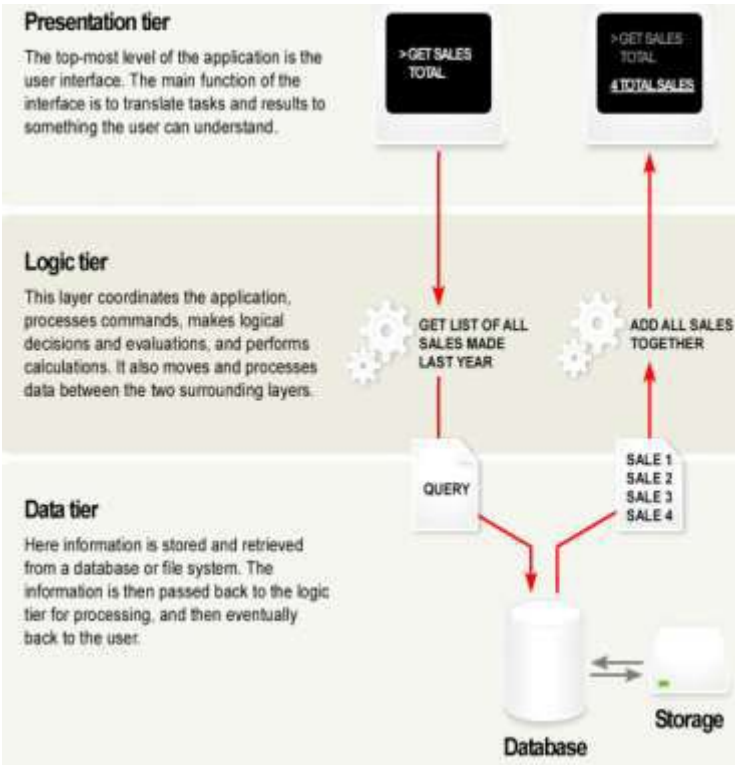
Presentation, logic, and data layers are **disconnected**.

- **Architecture Principles**

- Client - Server Architecture
- Each layer: **independent**, **not expose dependencies** of the implementation
- Unconnected layers do **not connect**
- Changes only affects the layer running on that particular platform

- **Presentation Layer** (Front-end):

- User **Interface:** Handles Interaction
- Should not contain **business logic** or **data access code**



- **Logic Layer** (middleware/back-end)

- Set of rules for **processing information**
- Can accommodate many users
- Should not contain **presentation** or d**ata access code**

- **Data Layer** (back-end):

- The physical storage layer for data persistence
- Manages access to DB or file system
- Should not contain **presentation** or **business logic code**

- ## 3-Tier Architecture Web Apps

Presentation Layer

- Static or dynamically generated content rendered by the browser (front-end)

Logic Layer

- A dynamic content processing and generation level application server, e.g., Python Django Framework (middleware)

Data Layer

- A database, comprising both data sets and the database management system or RDBMS software that manages and provides access to the data (back-end)

- ## 3-Tier Architecture – Advantages

Independence of Layers

- **Easier** to maintain
- Components are **reusable**
- Faster development (division of work):
  - Web designer does **presentation**
  - Software engineer does **logic**
  - DB admin does **data model**

1.2 - Design Patterns

Construction and testing
- how do we build a web application?
- what technology should we choose?

Re-use
- can we use standard components?

Scalability
- how will our web application cope with large numbers of requests?

Security
- how do we protect against attack, viruses, malicious data access, denial of service?

Different data views
- user types, individual accounts, data protection

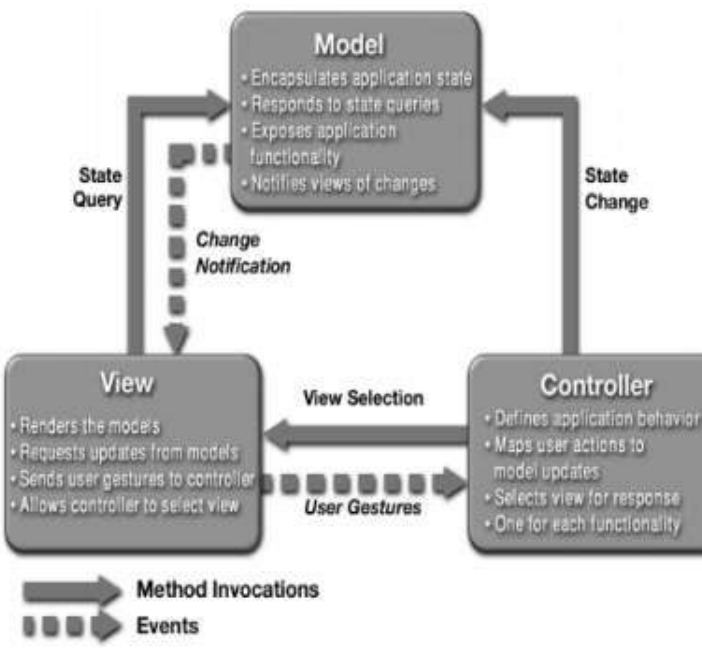
Design Pattern:
- general and reusable solution to a commonly occurring problem in the design of software
- template for how to solve a problem that has been used in many different situations
- NOT a finished design
    - The pattern must be adapted to the application
    - Cannot simply translate into code

1.2.1 - The MVC Design Pattern

Design Problems:
- Need to change the look-and-feel without changing the core/logic
- Need to present data under different contexts, etc…

Design Solution:
- **Separate core functionality** from the **presentation** and **control logic** that uses this functionality
- Allow **multiple views** to share the **same data model**
- Make **supporting multiple clients** easier to implement, test, and maintain

The Model-View-Controller Pattern



- Design pattern for graphical systems that promotes **separation between model and view**
- The **logic** required for **data maintenance** (database) is **separated** from **how** the **data** is **viewed** (graph, numerical) and **how** the data can be **interacted** with (GUI, command line, touch)

Model

- **manages** the **behavior** and **data** of the application domain
- **responds** to **requests** for information about its **state** (usually from the **view**) follows instructions to **change state** (usually from the **controller**)

- Contains domain-specific knowledge
- linked to a database
- Independent of view

View

- **renders** the **model** into a form suitable for interaction, typically a user interface (multiple views can exist for a single model for different purposes)

- Presents data to the user
- Allows user interaction

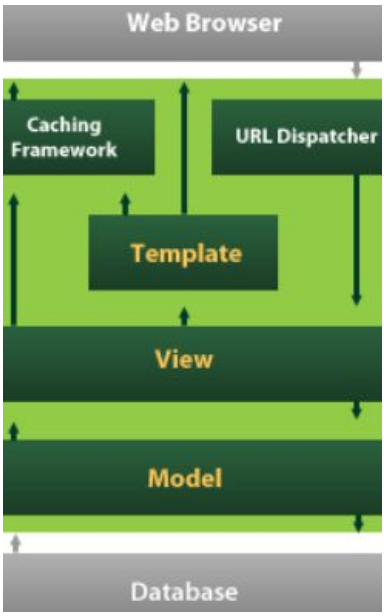
Controller

- receives **user input** and initiates a **response** by making calls on model objects
- accepts input from the user and instructs the model and viewport to perform actions based on that input

- defines how user interface reacts to user input (events)
- receives messages from view (where events come from)
- sends messages to model (tells what data to display)

2. Django I

Segue arquitetura MVC
Possui ORM (Object Relational Mapper)



Arquitetura

- **Models:** Descreve os dados
- **Views:** Controla o que os users vêm
- **Templates:** Controla como o vêm
- **URL Dispatcher:** Expedidor de URLs

2.1 - Views

- Create views in **views.py** as:

```
def numerot(request, num):
    tparams = {
        'num_arg': num,
    }
    return render(request, 'numerot.html', tparams)
```

- No **urls.py** adicionar route para a view

```
path('hello/', views.hello, name='hello'),
```

2.2 - Templates

Argumento/Variável: *{ { num_arg } }*
Template Tags: *{% ifequal num_arg 1000 %} / {% else %} / {% endifequal %}*

2.3 - Static Files

Os static files são ficheiros que se pretende simplesmente referenciar e servir ao cliente, sem qualquer processamento prévio.

- Imagens, CSS, Scripts

3. Django II

3.1 - Models

MTV - Model, Template, View
Model: Camada de acesso aos dados
Permite definir acesso, validação, comportamento, relações entre os dados.

No file models.py definir classes do modelo de dados.

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    date = models.DateField()
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher,
                on_delete=models.CASCADE)
    def __str__(self):
        return self.title
```

Relações entre as classes:

1:M e M:1

- Atributo da classe **models.ForeignKey**
    - Publisher (1) : (M) Book
    - O atributo é colocado na classe Book (M), aquela que representa muitos objetos para um.

1:1 (Único)

- Atributo da classe **models.OneToOne**
    - Exemplo: Book (1) : (1) Author
    - O atributo "deve" ser colocado na classe que mais "necessita" da outra

M:N

- Atributo da classe **models.ManyToManyField**
    - Book (M) : (N) Author
    - O atributo "deve" ser colocado na classe que mais "necessita" da outra
        - (No presente exemplo, o autor pode existir, por si, sem necessidade de ter livros, mas o livro necessita de um autor)

3.2 - Django's Database Layer

Django Admin Site: Gestão dos dados pertencentes ao modelo de dados.
Na admin.py registar as classes a gerir:

admin.site.register(Author)

- **Inserir um objeto**
  - a = Author(name='Antonio Pedro', email='apedro@email.com')
  - a.save()
- **Modificar um objeto**
  - a.email = 'antonio.pedro@email.com'
  - a.save()
- **Selecionar todos os objetos**
  - Autor.objects.all()
- **Filtrar objetos (por nome)**
  - Autor.objects.filter(name='Author1')
- **Filtrar por nome e por email**
  - Author.objects.filter(name='Autor1', email='...')
- **Filtrar por nome parecido**
  - Author.objects.filter(name_contains='Autor')
- **Aceder a um único objeto**
  - Author.objects.get(email='autor1@email.com')
- **Ordenação**
  - Publisher.objects.order_by("city", "country")
- **Filtragem e Ordenação**
  - Publisher.objects.filter(country='Portugal').order_by("-city")
- **Selecionar os primeiros resultados**
  - Publisher.objects.order_by("city", "country")[0]
  - Publisher.objects.order_by("city", "country")[0:4]
  - **Não são permitidos índices negativos**
- **Remover um objeto**
  - Author.objects.get(email='autor1@email.com').delete()

4. Django III

4.1 - Envio/Receção de Dados - Forms

Objeto "request" do tipo "HttpRequest" permite aceder a dados recebidos pelo web server
Podem ser acedidos através de atributos e métodos dedicados (request.path(), ex.)

View para aceder a dados no header HTTP

```
def info(request):
    values = request.META.items()
    html = []
    for k, v in values:
        html.append('<tr><td>%s</td><td>%s</td></tr>' % (k, v))
    return HttpResponse('<table>%s</table>' % '\n'.join(html))
```

FORMS

Forms servem para envio/receção de dados cliente <-> servidor

- **Lado do cliente:** forms podem ser usados para Get ou Post
- **Lado do servidor:** request.GET e request.POST para aceder aos dados

Exemplo:

```
def booksearch(request):
    if 'query' in request.POST:
        query = request.POST['query']
        if query:
            books = Book.objects.filter(title__icontains=query)
            return render(request, 'booklist.html',{'boks':books, 'query':query})
        else:
            return render(request, 'booksearch.html', {'error':True})
    else:
        return render(request, 'booksearch.html', {'error':False})
```

Pesquisa:

```
{% extends "layout.html" %}

{% block content %}

{% if error %}
    <p style="...">ERROR: Insert a query term.</p>
{% endif %}

<form action="." method="post">
    {% csrf_token %}
    <input type="text" name="query">
    <input type="submit" value="Search">
</form>

{% endblock %}
```

Resultados pesquisa:

```
{% extends "layout.html" %}
{% block content %}
<p>Search by: <strong>{{ query }}</strong></p>
{% if boks %}
<p>Found {{ boks|length }} book{{ boks|pluralize }}.</p>
<ul>
    {% for bok in boks %}
        <li>{{ bok.title }}</li>
        <ul>
            <li>{{ bok.publisher }}</li>
            <li>{{ bok.date }}</li>
            <ul>
            {% for aut in bok.authors.all %}
                <li>{{ aut }}</li>
            {% endfor %}
            </ul>
        </ul>
    {% endfor %}
</ul>
{% else %}
    <p>Not found any result.</p>
{% endif %}
{% endblock %}
```

5. Django Framework

5.1 - Django Forms

Form class: describes a form and determines how it works and appears in the browser.
Form class instantiation we can populate it:

- data from a saved model instance (as in the case of admin forms for editing);
- data that we have collated from other sources;
- data received from a previous HTML form submission.

Creating a Django Form

forms.py:

```
from django import forms


# Create your forms here.
class BookQueryForm(forms.Form):
    query = forms.CharField(label='Search:', max_length=100)
```

bookquery.html:

```
<form action="." method="post">
    {% csrf_token %}
    {{ form }}
    <input type="submit" value="Search">
</form>
```

views.py

```
def bookquery(request):
    # if POST request, process form data
    if request.method == 'POST':
        # create form instance and pass data to it
        form = BookQueryForm(request.POST)
        if form.is_valid(): # is it valid?
            query = form.cleaned_data['query']
            books = Book.objects.filter(title__icontains=query)
            return render(request, 'booklist.html', {'boks': books, 'query': query})
    # if GET (or any other method), create blank form
    else:
        form = BookQueryForm()
    return render(request, 'bookquery.html', {'form': form})
```

Access to form data, after its validation.

Data Field

- Data submitted with a form, using Form Fields, can be validated through is_valid() function.
- After validation, data can be accessed in form.cleaned_data dictionary.
- The data in this dictionary is already converted into Python types, for immediate use.

- BooleanField – as Check Box Input
- CharField – as Text Input
- IntegerField and FloatField – as Number Input
- DateField, TimeField – as Text Input
- ChoiceField – as Select
- MultipleChoiceField – as Select Multiple
- FileField – File Input

Rendering Form Fields Individually:

```
<form action="." method="post" class="form-horizontal">
    {% csrf_token %}
    <h4>Insert query to search book titles.</h4>
    <hr />
    <div class="form-group">
        {{ form.query.label_tag }}
        <div class="col-md-10">
            {{ form.query }}
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <input type="submit" value="Search" class="btn btn-default" />
        </div>
    </div>
</form>
```

5.2 - Django Authentication

It handles both **authentication** and **authorization**.

- **Authentication** verifies if users are who they claim to be.
- **Authorization** determines what authenticated users are allowed to do.

Implements: Users, groups, permissions, forms, view tools, password hashing.

Create Login and Logout (urls.py)

```
urlpatterns = [
    path('login/', auth_views.LoginView.as_view(template_name='login.html'), name='login'),
    path('logout', auth_views.LogoutView.as_view(next_page='/'), name='logout'),
```

login.html

```
<form action="." method="post" class="form-horizontal">
    {% csrf_token %}
    <h4>Use a local account to log in.</h4>
    <hr />
    <div class="form-group">
        {{ form.username.label_tag }}
        <div class="col-md-10">
            {{ form.username }}
        </div>
    </div>
    <div class="form-group">
        {{ form.password.label_tag }}
        <div class="col-md-10">
            {{ form.password }}
        </div>
    </div>
```

Authorization

```
def authorins(request):
    if not request.user.is_authenticated or request.user.username != 'admin':
        return redirect('/login')
    # if POST request, process form data
    if request.method == 'POST':
        # create form instance and pass data to it
        form = AuthorInsForm(request.POST)
        if form.is_valid(): # is it valid?
            name = form.cleaned_data['name']
            email = form.cleaned_data['email']
            a = Author(name=name, email=email)
            a.save()
            return HttpResponse("<h1>Author Inserted!!!</h1>")
    # if GET (or any other method), create blank form
    else:
        form = AuthorInsForm()
    return render(request, 'authorins.html', {'form': form})
```

Through object "request.user", it's possible to verify if a given user is authenticated and have authorization to do operations.

5.3 - Django Sessions

State

- HTTP protocol is a stateless protocol, which means that it doesn't have any mechanism to save the connection state and as so it doesn't allow sessions creation.
- To do this, some exterior mechanisms were developed in web clients and servers, which allow to save state data over multiple HTTP connections, producing artificial sessions.

Like: Cookies

Cookies

- A cookie is a little piece of information sent by a web server to its client, a browser, to save it while they are in communication.
- It's possible to save some kind of information in this cookie, like the user's username, for example.
- This cookies mechanism is in wide use by almost web sites, but it has some disadvantages:
  - Saving cookies in the browser is not compulsory, which doesn't allow to offer warranty of a good service;
  - They can't be used to save important information – they aren't secure;
  - The server can be inhibited, at some time, to access crucial information to continue the interaction with the client.

Django Sessions:

- Django manage automatically the sessions with its clients in a simple and clean way, through "request.session" object, which is a dictionary.

```
def bookquery(request):
    # if POST request, process form data
    if request.method == 'POST':
        # create form instance and pass data to it
        form = BookQueryForm(request.POST)
        if form.is_valid(): # is it valid?
            query = form.cleaned_data['query']
            if 'searched' in request.session and request.session['searched'] == query:
                return HttpResponse('Query already made!!!')
            request.session['searched'] = query
            books = Book.objects.filter(title__icontains=query)
            return render(request, 'booklist.html', {'boks': books, 'query': query})
    # if GET (or any other method), create blank form
    else:
        form = BookQueryForm()
    return render(request, 'bookquery.html', {'form': form})
```

6. Production

6.1 Web Applications Deployment

Production Environment:

- Environment provided by the server computer where you will run your web application for external consumption.

Includes:

- Computer Hardware
- OS
- Programming Language
- Web server
- Application server
- Databases

- The server computer could be located on your own facility and connected to the Internet by a fast link

OR

- Use a server computer that is hosted "in the cloud".

IaaS

Infrastructure as a Service – is a remotely accessible computing/networking hardware, provided as a hosting service.

- Many IaaS providers offer options to preinstall a particular operating system, onto which you must install the other components of your production environment.
- Other vendors allow you to select more fully-featured environments, including a complete particular web framework, e.g Django, and a web-server setup

PaaS

Platform as a Service – is another kind of hosting service where the host platform takes care of:

- most of the production environment – web server, application server, load balancers;
- and most of what you need to scale the application.

- PaaS makes deployment quite easy, because you just need to concentrate on your web application and not all the other server infrastructure.

Choosing a Hosting Service

Issues to take in account:

- How busy your site is likely to be and the cost of data and computing resources required to meet that demand.
- Level of support for scaling horizontally (adding more machines) and vertically (upgrading to more powerful machines) and the costs of doing so.
- Where the supplier has data centers, and hence where access is likely to be fastest.
- The host's historical uptime and downtime performance. • Tools provided for managing the site — are they easy to use and are they secure (e.g. SFTP vs FTP).
- Inbuilt frameworks for monitoring your server. Known limitations. Some hosts will deliberately block certain services (e.g. email). Others offer only a certain number of hours of "live time" in some price tiers, or only offer a small amount of storage.
- Additional benefits. Some providers will offer free domain names and support for SSL certificates that you would otherwise have to pay for.
- Whether the "free" tier you're relying on expires over time, and whether the cost of migrating to a more expensive tier means you would have been better off using some other service in the first place!