Advanced **Django**

Building complex, secure and scalable web application with Django

Vikash Patel

What we will learn?

1. Database Models and Forms
2. Generic class-based views (CBVs)
3. REST APIs with Django REST Framework
4. Django Channels and real-time applications
5. Writing custom management commands
6. Customizing Django's admin interface
7. Extending Django with third-party apps
8. Caching and performance optimization
9. Django security best practices
10. Deployment and scalability

Why we use these features?

- build more complex and powerful web applications (realtime apps)
- improve the performance and scalability of your web applications
- make your code more reusable and maintainable (modules or extensions)
- improve the security of your web applications
- customize Django to fit your specific needs

1. Models and Forms

- Django Model Forms provides a way to link the **data submitted** by the client through the **form** to a **Data Model** created to store the entry.
- Less code
- Easy to maintain, extend
- We can render the django form as html with *form.as_p* attribute
- Easily change the default style with crispy-forms package, includes error highlighting.

1.1 Create a model ‘BioLink’ in models.py

```
class BioLink(models.Model):
    name = models.CharField(max_length=100)
    link = models.URLField()
    owner = models.ForeignKey('auth.User', related_name='bio_links', on_delete=models.CASCADE)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    def __str__(self):
        return f'{self.name}({self.link})'
```

1.2 Create the Form in forms.py

```
from django import forms
from biolinks.models import BioLink

class BioLinkForm(forms.ModelForm):
    class Meta:
        model = BioLink
        fields = ('name', 'link')
```

1.3 Create a form view to render the form

```
def FormView(request):
    if request.method == 'POST':
        form = BioLinkForm(request.POST)
        if form.is_valid():
            form.instance.owner = request.user
            form.instance.save()
            form.save()
            return HttpResponse('New Link Saved')
    else:
        form = BioLinkForm()
        context = { 'form': form, }
    return render(request, 'biolinks/form.html', context)
```

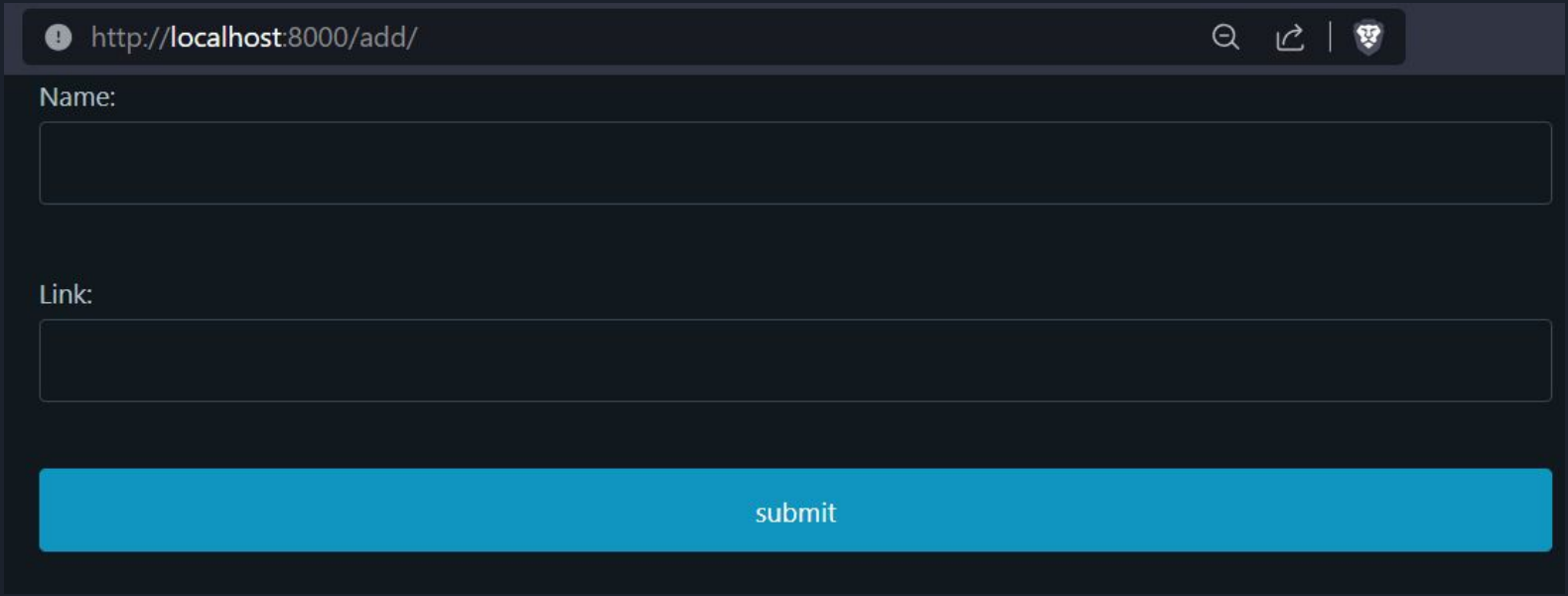
1.4 The template ‘biolinks/form.html’

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

1.5 Add a new url path

```
urlpatterns = [
    path('', BioLinksListView.as_view(), name='bio_links'),
    path('my/', BioLinksOwnedByUserListView.as_view(), name='bio_links_by_user'),
    # path('', bio_links_view, name='Bio Links home'),
    path('add/', FormView, name='FormView'),
]
```

1.6 This the rendered page



2. Generic class-based views (CBVs)

Here are some of the main points about Django generic CBVs:

- CBVs are classes that inherit from the View class.
- CBVs provide common functionality for handling HTTP requests, such as dispatching requests to the appropriate method, handling errors, and rendering templates.
- CBVs are reusable, meaning that you can use them to implement different types of views without having to write a lot of boilerplate code.
- CBVs are DRY, meaning that you can avoid duplicating code by using the same CBVs to implement different views.
- CBVs are maintainable, meaning that your code will be easier to understand and maintain if you use CBVs.

2.1 CBVs: Examples

Here are some examples of Django generic CBVs:

- **ListView**: Displays a list of objects.
- **DetailView**: Displays a single object.
- **CreateView**: Creates a new object.
- **UpdateView**: Updates an existing object.
- **DeleteView**: Deletes an existing object.

2.2 CBVs: Benefits

Here are some of the benefits of using Django generic CBVs:

- **Code reuse:** CBVs can be reused to implement different types of views without having to write a lot of boilerplate code.
- **DRY:** CBVs can help you to avoid duplicating code by using the same CBVs to implement different views.
- **Maintainability:** Your code will be easier to understand and maintain if you use CBVs.
- **Performance:** CBVs can help to improve the performance of your web application by caching data and reducing the number of database queries that are required to render a page.

2.3 CBVs: Code Example

```python
from django.views.generic.list import ListView

class BioLinksOwnedByUserListView(ListView):
    model = BioLink
    template_name = 'biolinks/index.html'
    def get_queryset(self):
        return BioLink.objects.filter(owner=self.request.user)

    def get_context_data(self, **kwargs):
        # Call the base implementation first to get the context
        context = super(BioLinksOwnedByUserListView, self).get_context_data(**kwargs)
        # Create any data and add it to the context
        context['bio_links'] = self.get_queryset()
        return context
```

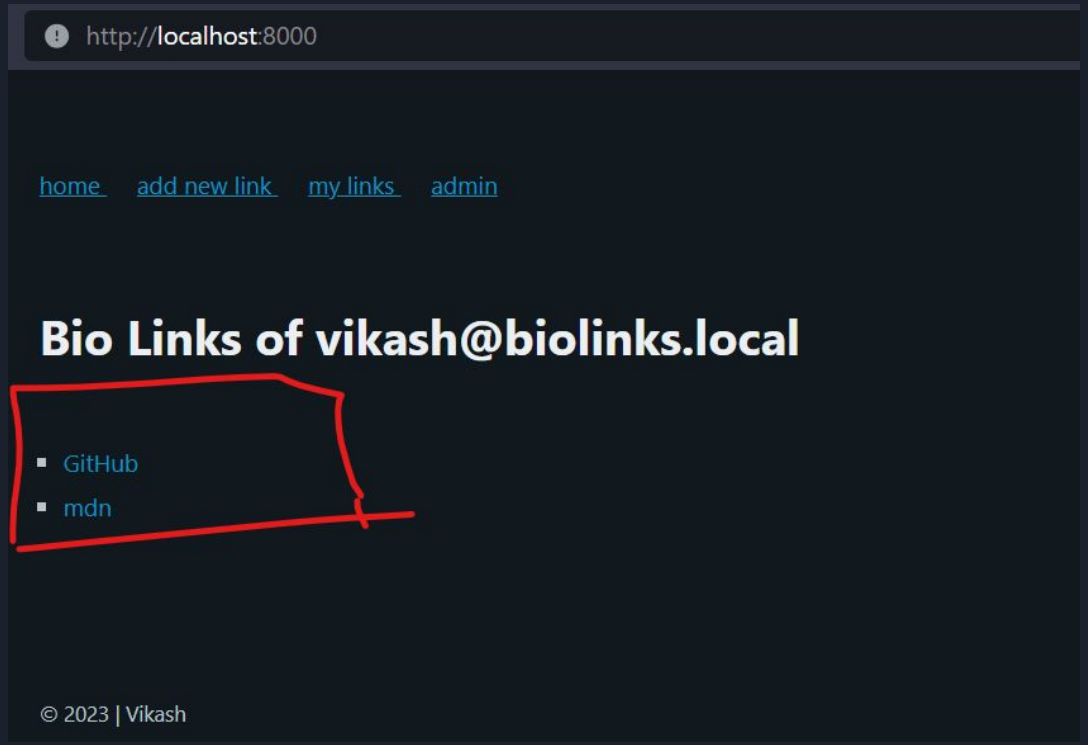
The code of template ‘biolinks/index.html’

```
….

<ul>
    {% for bio_link in bio_links %}
    <li>
        <a href="{{ bio_link.link }}" target="_blank">{{ bio_link.name }}</a>
    </li>
    {% endfor %}
</ul>

….
```

The page rendered looks like this (a list)



3. REST APIs with Django REST Framework

Read more: https://www.django-rest-framework.org/

- REST APIs are a way to expose data and functionality to other applications through a standardized interface.
- DRF is used for building REST APIs with Django.
- DRF provides a number of features that make it easy to build REST APIs, including:
  a. **Serialization:** Django REST Framework supports a variety of serialization formats, including JSON, XML, and YAML.
  b. **Authentication:** Django REST Framework provides a number of authentication mechanisms, including basic authentication, token authentication, and OAuth2 authentication.
  c. **Throttling:** Django REST Framework provides a number of throttling mechanisms to limit the number of requests that a client can make.
  d. **Versioning:** Django REST Framework provides a number of versioning mechanisms to allow different versions of the API to be served simultaneously.
- Django REST Framework is a popular choice for building REST APIs with Django, and it is used by many large companies, including Mozilla, Pinterest, and Instagram.

3.1 DRF Features

Here are some of the benefits of using DRF:

- **Ease of use:** Django REST Framework is easy to use, even if you are new to building REST APIs.
- **Flexibility:** Django REST Framework is very flexible and can be used to build a wide variety of REST APIs.
- **Performance:** Django REST Framework is performant and can handle a large number of requests.
- **Scalability:** Django REST Framework is scalable and can be used to build REST APIs that can handle large amounts of traffic.
- **Community support:** Django REST Framework has a large and active community, which means that there are many resources available to help you learn how to use it and troubleshoot problems.

4. Django Channels and real-time apps

Read more: https://channels.readthedocs.io/

- Django Channels is a library that extends Django to support WebSockets and other asynchronous protocols.
- WebSockets are a communication protocol that allows for full-duplex communication between a client and a server over a single TCP connection.
- Real-time applications are applications that allow for continuous, two-way communication between the client and the server. (like, chatbots)
- One important thing: Django channels require a asynchronous server (like daphane)

4.1 Benefits of Django Channels

Here are some of the benefits of using Django Channels to build real-time applications:

- **Easy to use:** Django Channels is easy to use, even if you are new to building real-time applications.
- **Well-integrated with Django:** Django Channels is well-integrated with Django, which makes it easy to use Django features such as authentication and authorization in your real-time applications.
- **Scalable:** Django Channels is scalable and can handle a large number of concurrent connections.
- **Community support:** Django Channels has a large and active community, which means that there are many resources available to help you learn how to use it and troubleshoot problems.

5. Custom management commands

This is the directory structure for creating a command (here ***list-biolinks*** is a command).

The ***_private*** module will not be available as a command. You can use it to write your logic and manage the code.

```
biolinks
├──management
│    │    __init__.py
│    │
│    └──commands
│          list-biolinks.py
│          _private.py
│          __init__.py
```

5.1 Add the logic in the ‘list-biolinks.py’

```python
from django.core.management.base import BaseCommand, CommandError
from biolinks.models import BioLink
from django.contrib.auth import get_user_model

class Command(BaseCommand):
    help = "Prints all the available biolinks"
    def add_arguments(self, parser):
        # optional argument (remove -- to make positional argument)
        parser.add_argument("--user-emails", nargs="+", type=str)

    def handle(self, *args, **options):
        # write you logic here
        self.stdout.write(self.style.SUCCESS("Hello World"))
```

5.2 override ‘handle’ method

```
def handle(self, *args, **options):
    biolinks = []
    if 'user_emails' in options and options["user_emails"] is not None:
        for user_email in options["user_emails"]:
            try:
                user = get_user_model().objects.get(email=user_email)
                biolinks = BioLink.objects.filter(owner_id=user.id)
            except BioLink.DoesNotExist:
                raise CommandError('Poll "%s" does not exist' % user_email)
    else:
        biolinks = BioLink.objects.all()
    for biolink in biolinks:
        self.stdout.write(self.style.SUCCESS(biolink.link))
```

5.2 The output of the command

```
pwsh  advanced-django-features-demo  main ≡  .venv 3.11.5  default@ap-south-1
1:42 AM >>I python .\manage.py list-biolinks --user-emails vikash@biolinks.local
https://github.com
http://localhost:8000/add/
```

```
pwsh  advanced-django-features-demo  main ≡
1:47 AM >>I python .\manage.py list-biolinks
https://github.com
https://developer.mozilla.org/
http://localhost:8000/add/
```

6. Customize Admin Dashboard

Read more: https://docs.djangoproject.com/en/4.2/ref/contrib/admin/

Two options:

1. Install extensions and themes
2. Write code and add functionality

6.1 Admin Panel Customization

- Modifying a Change List Using list_display
- Providing Links to Other Object Pages
- Adding Filters to the List Screen
- Adding Search to the List Screen
- Changing How Models Are Edited

6.1.1 Modifying a Change List Using list_display

```
class BioLinkAdmin(admin.ModelAdmin):
    list_display = ('id', 'name', 'link', 'owner', 'created_at',
'updated_at',)
```

	ID	NAME	LINK	OWNER	CREATED AT	UPDATED AT
	3	http://localhost:8000/add/	http://localhost:8000/add/	vikash@biolinks.local	Oct. 15, 2023, 7:32 p.m.	Oct. 15, 2023, 7:32 p.m.
	2	mdn	https://developer.mozilla.org/	v@example.org	Oct. 15, 2023, 6:20 p.m.	Oct. 15, 2023, 6:20 p.m.
	1	GitHub	https://github.com	vikash@biolinks.local	Oct. 15, 2023, 5:39 p.m.	Oct. 15, 2023, 5:39 p.m.

6.1.2 Providing Links to Other Object Pages

Create method inside admin class

```
def view_owner(self, obj):
    url = (
        reverse("admin:users_customuser_change", args=(obj.owner.id,))
        + "?"
        + urlencode({"owner__id": f"{obj.id}"})
    )
    return format_html('<a href="{}"> {}</a>', url, obj.owner.email)
view_owner.short_description = "Owner"
```

Use it in the list display:

```
class BioLinkAdmin(admin.ModelAdmin):
    list_display = ('id', 'name', 'link', 'view_owner', 'created_at', 'updated_at',)
```

6.1.3 Adding Filters to the List Screen

```
class BioLinkAdmin(admin.ModelAdmin):
    list_filter = ('created_at', 'updated_at', 'owner')
```

FILTER

By created at

Any date
Today
Past 7 days
This month
This year

By updated at

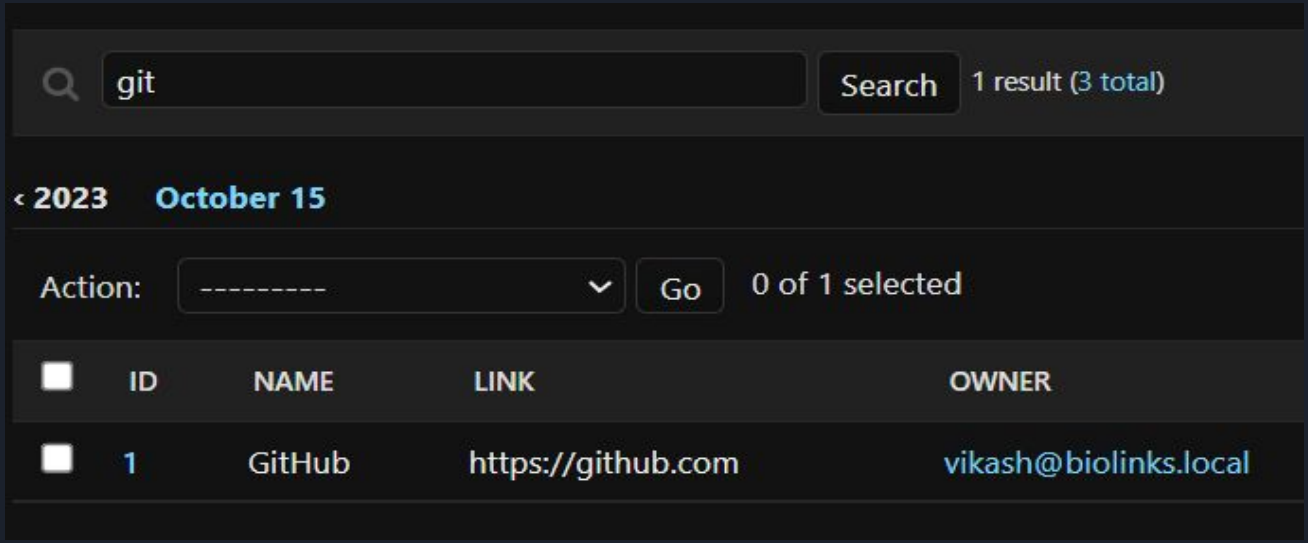
Any date
Today
Past 7 days
This month
This year

By owner

All
v@example.org
vikash@biolinks.local

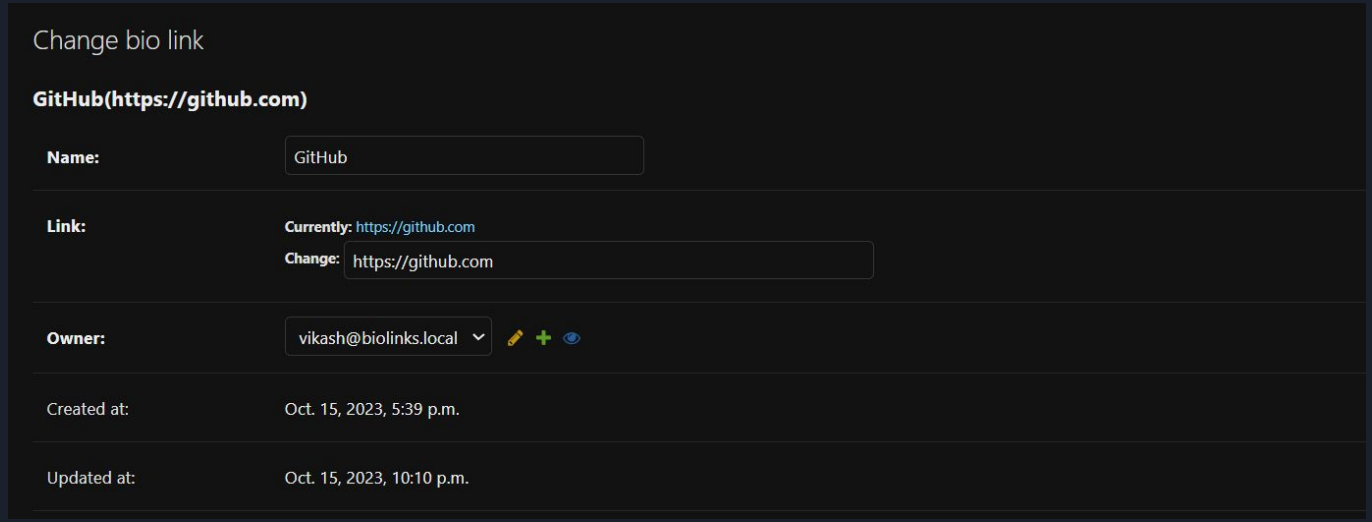
6.1.4 Adding Search to the List Screen

```
class BioLinkAdmin(admin.ModelAdmin):
    search_fields = ('name',)
```



6.1.5 Changing How Models Are Edited

```
class BioLinkAdmin(admin.ModelAdmin):
    fields = ('name', 'link', 'owner', 'created_at', 'updated_at',)
    readonly_fields = ('created_at', 'updated_at',)
```



7. Extending Django with third-party apps

- Extensions provide additional functionality.
- There different types of extensions
  - Extending management command functionality
  - Extending admin dashboard functionality
  - Support extensions (caching, database, security, performance etc)

7.1 Some of the useful extensions

Read more: https://djangopackages.org/

- Dj-tracker - Query Performance Tracker
- Django Debug Toolbar - Debugging in development
- GitHub - djblets/djblets: A collection of useful extensions for Django.
- Django Extensions - Adds many useful management commands
- django-admin-generator · PyPI - Generate admin panel config for models
- Django-allauth - Authentication provider (Social accounts, Multifactor)
- Django-two-factor-auth - Dedicated 2 factor application (need to integrate with our project)
- django-storages - Django Storage, store media files and static assets in cloud (s3, google bucket, b2 etc)

8. Performance and Caching

Django Caching and performance optimization write points:

- **Use caching:** Add redis cache to handle the repetitive requests.
- **Optimize database queries:** Database queries are one of the most common performance bottlenecks in Django applications. You can optimize database queries by using the following tips:
  - Use indexes on the columns that you are querying.
  - Avoid using `SELECT *` queries. (Always filter on the server side)
  - Use `LIMIT` and `OFFSET` clauses to paginate your results. (fetch only what you need)
  - Use **prefetch** and **select_related** to reduce the number of database queries that are required to render a page.
- **Optimize static files**: Host the css, js, images, media files on a dedicated file server (or S3). Implement efficient cache mechanism. Use a CDN (cloudfront, cloudflare etc)
- **Use a profiler:** Run a profiler to understand the slow parts of the application.
- **Send small packets:** Don't send a lot of data in a single request.

(When app is slow, implement cache)

8.1 Caching with Django

- Django provides a built-in caching framework that can be used to cache database queries, template fragments, and other data.
- The Django caching framework supports a variety of cache backends, such as Memcached, Redis, and the database cache.
- To use caching in Django, you first need to choose a cache backend and configure it in your Django settings file.
- Once you have configured a cache backend, you can start caching data using the **cache** API.
- The cache API provides a number of methods for storing and retrieving cached data.
- You can also use caching middleware to automatically cache database queries and template fragments.
- Caching can improve the performance of your Django application by reducing the number of database queries that are required to render a page.
- **Implement the policy to invalidate and update the cache.**

9. Security Best Practices

- **Keep Django up to date:** Make sure to regularly update Django and its dependencies to the latest versions to patch any known security vulnerabilities.
- **Enable Debug Mode carefully:** Debug Mode should only be enabled on development and staging environments. Debug Mode can expose sensitive information to attackers in production environments.
- **Secure your Django Admin Panel: T**he Django Admin Panel is a powerful tool, but it can also be a security risk if it is not properly secured. Make sure to use a strong password for the Admin Panel and to enable two-factor authentication.
- **Implement Strong Authentication:** Users should be required to use strong passwords and two-factor authentication. User accounts should also be locked out after a certain number of failed login attempts.
- **Protect Against Cross-Site Request Forgery (CSRF):** Django provides built-in protection against CSRF attacks, but it is important to make sure that your application is properly configured.
- **Sanitize User Input:** All user input should be sanitized before it is used to prevent XSS attacks.
- **Use Prepared Statements:** Prepared statements can help to prevent SQL injection attacks.
- **Implement Content Security Policy (CSP):** A CSP can help to protect your application from XSS attacks and other types of attacks.
- **Regularly Backup and Monitor Your Application:** It is important to regularly backup your application data and to monitor your application for signs of attack.

`

10. Deployment and scalability

- A Django application can be deployed in either PaaS environment or IaaS environment.
- Popular choices are: Heroku, Digital Ocean and AWS cloud.
- If you have highly active application then, you must go with multi tier deployment.
- You can consider using, horizontal scaling and load balancer.
- Deploy your database in a managed service (reduce maintenance overhead)
- Before deployment, read the deployment checklist page from django docs

10. 1 Deployment checklist

Run **python manage.py check --deploy**

```
pwsh  advanced-django-features-demo  main ≡ ?1 ~3 -21                .venv 3.11.5
3:47 AM » python manage.py check --deploy
System check identified some issues:

WARNINGS:
?: (security.W004) You have not set a value for the SECURE_HSTS_SECONDS setting. If your entire
site is served only over SSL, you may want to consider setting a value and enabling HTTP Strict
Transport Security. Be sure to read the documentation first; enabling HSTS carelessly can cause
serious, irreversible problems.
?: (security.W008) Your SECURE_SSL_REDIRECT setting is not set to True. Unless your site should
be available over both SSL and non-SSL connections, you may want to either set this setting True
 or configure a load balancer or reverse-proxy server to redirect all connections to HTTPS.
?: (security.W012) SESSION_COOKIE_SECURE is not set to True. Using a secure-only session cookie
makes it more difficult for network traffic sniffers to hijack user sessions.
?: (security.W016) You have 'django.middleware.csrf.CsrfViewMiddleware' in your MIDDLEWARE, but
you have not set CSRF_COOKIE_SECURE to True. Using a secure-only CSRF cookie makes it more diffi
cult for network traffic sniffers to steal the CSRF token.
?: (security.W018) You should not have DEBUG set to True in deployment.
```

Get the code from here (GitHub repo)

https://to.lorbic.com/GDAv6a

https://github.com/vk4s/advanced-django-features-demo

Questions

Thank you

Contact:

Vikash Patel

Linkedin: https://www.linkedin.com/in/vk4s/