ChatGPT

Full Stack React/Django Interview Q&A

- **What are React Hooks and why were they introduced?** React Hooks are functions (introduced in React 16.8) that let functional components "hook into" React state and lifecycle features without using class components [1]. Hooks like `useState`, `useEffect`, etc., enable components to manage state and side-effects more concisely. They solve problems of sharing stateful logic across components, making code more modular and reusable than class-based lifecycles [2]. For example:

```
import { useState } from 'react';
function Counter() {
  const [count, setCount] = useState(0);
  return <button onClick={() => setCount(count+1)}>Count: {count}</button>;
}
```

Here `useState` adds a `count` state to the functional component [1].

- **What is `useState` in React and how does it work?** The `useState` Hook lets functional components have local state. It returns a state variable and a setter function:

```
const [value, setValue] = useState(initialValue);
```

On first render, `value` is set to `initialValue`. Calling `setValue(newVal)` schedules a re-render with the state updated. Internally React preserves the state between renders; each update enqueues a state change so React can compute the new UI [3].

- **What is `useEffect` and how is it used for side effects?** `useEffect` lets you perform side effects in functional components (data fetching, subscriptions, DOM updates). It runs after render by default. For example, to fetch data from an API when a component mounts or when certain dependencies change:

```
useEffect(() => {
  fetch('/api/data').then(res => res.json()).then(data => setData(data));
}, [/* dependency array */]);
```

If you pass an empty array `[]`, the effect runs once (like `componentDidMount` in class components) [4]. The dependency array controls when the effect re-runs. In each case, React ensures the returned effect runs at the appropriate times to mimic lifecycle methods [4] [5].

- **Explain the difference between props and state in React.** *Props* (short for properties) are inputs to a component, passed from a parent; they are immutable within the child. *State* is internal, mutable data managed by the component itself. For instance, a parent can pass a message prop to a child, but the child cannot change that prop's value. The child can hold its own state and update it (e.g. via `useState`). In summary, props flow *into* a component (read-only), whereas state lives inside a component (read/write) [6].

- **What is the React component lifecycle and how do Hooks relate to it?** In class components, lifecycle methods like `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` controlled side-effects. In functional components, Hooks like `useEffect` and `useLayoutEffect` cover these phases. For example, `useEffect(() => { … }, [])` runs once after initial render (like `componentDidMount`); if you include dependencies, the effect re-runs on updates (like `componentDidUpdate`) [4]. Unmount cleanup can be done by returning a function from the effect. React's newer concurrency (React Fiber) model also affects when effects commit, but `useEffect` abstracts most of that for us [4].

- **How do you handle forms in React?** React commonly uses *controlled components*: form input values are tied to component state. For example:

```
const [name, setName] = useState("");
return (
  <form onSubmit={...}>
    <input value={name} onChange={e => setName(e.target.value)} />
    {/* ... */}
  </form>
);
```

On each keystroke, the state is updated via `onChange`, so React always has the current input value in state. This allows validation, conditional UI, and handling submit easily. Alternatively, libraries like *React Hook Form* or *Formik* abstract form handling, but under the hood they rely on React state and events. Uncontrolled components (using refs) are another approach, but controlled is most common for interview Qs.

- **What is React Router and how do you use it?** React Router is the standard library for client-side routing in React. It lets you map URL paths to components. Typically you wrap your app in `<BrowserRouter>` and use `<Route>` components or hooks. For example:

```
import { BrowserRouter, Route, Link, Switch } from 'react-router-dom';
// ...
<BrowserRouter>
  <nav>
    <Link to="/">Home</Link>
    <Link to="/about">About</Link>
  </nav>
  <Switch>
```

```
    <Route exact path="/" component={Home} />
    <Route path="/about" component={About} />
  </Switch>
</BrowserRouter>
```

This sets up two routes. React Router renders the appropriate component based on `window.location`. It also supports nested routes, parameters, redirect, and history navigation. Use `<Link>` or `useHistory` to programmatically navigate.

- **When should you use Redux vs. React Context API for state management?** Context API is built into React for passing data down the component tree (avoiding prop drilling). It's great for simple global state (theme, language, current user) and small apps. Redux, by contrast, creates a centralized store with actions and reducers, providing predictable state updates and devtools for debugging. Redux has more boilerplate, but it scales better for large apps with complex state and many interactions. In short, for a small or medium app, Context (with `useContext`) is simpler; for a large app needing fine-grained control, undo/redo, or advanced middleware, Redux is preferred [7] [8].

- **What is React Context and how do you use it?** React Context lets you provide a value at a top level and consume it in any descendent without manual prop passing. To use it:

```
const MyContext = React.createContext(defaultValue);
function App() {
  const [value, setValue] = useState("hello");
  return (
    <MyContext.Provider value={{value, setValue}}>
      <Child />
    </MyContext.Provider>
  );
}
function Child() {
  const { value, setValue } = useContext(MyContext);
  return <div>{value}</div>;
}
```

The child calls `useContext(MyContext)` to access the provided data. This avoids prop drilling by making data available to any nested component.

- **What is React.lazy and how does it help performance?** Lazy loading components delays loading non-essential code until needed, reducing initial bundle size. In React, use `React.lazy()` with `<Suspense>` to achieve this [9] [10]. For example:

```
const Heavy = React.lazy(() => import('./HeavyComponent'));
// in render:
<Suspense fallback={<div>Loading...</div>}>
```

```
  {showHeavy && <Heavy />}
</Suspense>
```

Here `HeavyComponent` is only loaded when `showHeavy` is true, and while it loads, the fallback UI is shown. This significantly improves initial load time by splitting code into chunks [9] [10]. Similarly, you can lazy-load route-based components with React Router.

- **What is React.memo and when would you use it?** `React.memo` is a higher-order component that memoizes a functional component: it shallowly compares props and re-renders the component only if props have changed [11]. Use it to optimize performance when a component renders the same output for the same props, preventing unnecessary re-renders. For example:

```
const HeavyList = React.memo(function HeavyList({ items }) {
  /* ... renders list ... */
});
```

Now if parent re-renders with identical `items`, `HeavyList` will skip updating. Combined with `useCallback` / `useMemo`, it helps avoid expensive recomputations [11].

- **What are Django models and how are they defined?** In Django, a *model* is a Python class that maps to a database table. Models live in `models.py`. Fields (e.g. `CharField`, `IntegerField`) define columns. For example:

```
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=100)
    birthdate = models.DateField(null=True)
```

This defines an `Author` table with `id`, `name`, and `birthdate` columns. Django automatically generates the schema and lets you interact with rows as Python objects. You can add methods or metadata via an inner `Meta` class for ordering or uniqueness. Changes to models are migrated to the DB using `makemigrations` / `migrate`.

- **What are Django views and what's the difference between function-based and class-based views?** A Django *view* is a Python callable that takes a web request and returns a response (often rendering a template or JSON). Function-based views are simple Python functions, while class-based views (CBVs) are classes (subclassing `View` or `APIView`) that bundle handling methods (`get()`, `post()`, etc.). CBVs and especially generic class-based views (like `ListView`, `DetailView`, DRF's `ListAPIView`, etc.) provide reusable behavior (e.g. CRUD for a model) and can reduce boilerplate. For API development, DRF's `APIView` and `ViewSet` are class-based. A function view example:

```
def hello(request):
    return HttpResponse("Hello, world")
```

A simple class-based view example (in DRF):

```
from rest_framework.views import APIView
class HelloView(APIView):
    def get(self, request):
        return Response({"msg": "Hello"})
```

- **What is a serializer in Django REST Framework (DRF) and why is it used?** A serializer converts Django model instances (or other data) to JSON (or other formats) and vice versa (deserialization). It also handles validation. For example:

```
from rest_framework import serializers
class AuthorSerializer(serializers.ModelSerializer):
    class Meta:
        model = Author
        fields = ['id', 'name', 'birthdate']
```

This defines how `Author` objects are represented in API responses and how incoming JSON is validated and saved. Serializers thus bridge between Django models and JSON for REST APIs. They replace forms in API contexts.

- **What is a ViewSet in DRF, and why use it?** A **ViewSet** is a DRF class that groups multiple related views (actions) into a single class. Rather than writing separate views for list, retrieve, create, update, delete, you define all actions in one `ViewSet`. DRF's routers then generate URL routes for each action automatically. For example, a `ModelViewSet` with `queryset` and `serializer_class` defined gives you full CRUD at once. It **keeps code organized and reduces duplication** [12]. Example:

```
from rest_framework import viewsets
class AuthorViewSet(viewsets.ModelViewSet):
    queryset = Author.objects.all()
    serializer_class = AuthorSerializer
```

Using a router, this creates endpoints for listing authors (`GET /authors/`), detail (`GET /authors/{id}/`), and so on.

- **How does Django REST Framework handle authentication and permissions?** DRF supports multiple **authentication** schemes: SessionAuthentication (Django's default sessions/cookies), TokenAuthentication, and JWT (often via third-party packages). In your settings you can enable them,

and DRF automatically checks credentials. For example, with JWT, clients include an `Authorization: Bearer <token>` header on API calls; DRF validates it to identify the user.

Permissions control access once authenticated. DRF provides classes like `IsAuthenticated`, `IsAdminUser`, and `DjangoModelPermissions`. For instance, `IsAuthenticated` allows any logged-in user (403 otherwise), and `IsAdminUser` only allows users with `is_staff=True` [13]. You can attach permissions globally or per-view. For custom rules, subclass `BasePermission` and override `has_permission()` and/or `has_object_permission()` [14].

- **How do you handle forms in Django?** Django has Form and ModelForm classes. A `Form` defines fields and validation explicitly; a `ModelForm` is tied to a model and auto-generates fields. For example:

```
from django import forms
from .models import Author

class AuthorForm(forms.ModelForm):
    class Meta:
        model = Author
        fields = ['name', 'birthdate']
```

In a view, you instantiate this form with `request.POST` data. `form.is_valid()` runs validation and, if valid, `form.save()` can create/update the model instance. Django forms automatically handle CSRF tokens in templates if you include `{{ form.csrf_token }}`. They simplify input sanitization, repopulating form errors, and rendering HTML form fields.

- **What are Django signals and how do you use them?** Signals let you run code in response to certain events (like model save/delete). Common signals include `pre_save`, `post_save`, `pre_delete`, etc. For example, to automatically create a profile when a new user is saved:

```
from django.db.models.signals import post_save
from django.dispatch import receiver
@receiver(post_save, sender=User)
def create_profile(sender, instance, created, **kwargs):
    if created:
        Profile.objects.create(user=instance)
```

Here, `create_profile` runs after an `User` is created (because `created` is True) and makes a `Profile`. Signals are a decoupled way to trigger side effects on model events [15].

- **How do you use the Django ORM to query the database?** Django's ORM abstracts SQL. You use model classes and manager/queryset methods. For example, `Author.objects.get(id=1)` fetches one author. `Author.objects.filter(name__icontains="john")` returns a queryset of authors with "john" in their name. You can chain filters, order by, and more. Changes: `author =`

`Author(name="X"); author.save()` or bulk create. The ORM handles escaping and security. It also supports raw SQL if needed. In summary, the ORM lets you "add, delete, change, and query objects" as Python, without writing SQL [16].

- **What are `select_related` and `prefetch_related` in Django, and why use them?** These are ORM tools to optimize queries and avoid the "N+1" problem. `select_related('fk_field')` performs a SQL join to include related models in one query (best for ForeignKey/OneToOne) [17]. For example:

```
books = Book.objects.select_related('author').all()
for book in books:
    print(book.author.name)
```

This fetches each book and its author in a single query [17]. `prefetch_related('many_field')` fetches related objects (ManyToMany or reverse FK) in a separate query and joins them in Python [18]. Using these greatly reduces database hits, as the ORM can batch-load related data instead of querying per object [17] [18].

- **What is Django middleware and what are common use cases?** Middleware are hooks processing requests/responses globally. Each middleware is a Python class with `__call__` or specific methods. Django ships with common middleware for CSRF protection, authentication, sessions, security headers, etc. For example, the `CsrfViewMiddleware` automatically checks for CSRF tokens on unsafe requests. You might write custom middleware to log requests, enforce SSL redirects, or handle exceptions. Middleware wrap around view processing: before the view runs (request phase) and after (response phase) [19] [20].

- **How does Django handle CSRF protection?** Django uses a token-based CSRF protection. In HTML forms, you include `{% csrf_token %}`, which inserts a hidden input with a secret token. On form submission, Django verifies this token matches the user's session. The token is unique per user session and per form. If the token is missing or invalid, Django returns 403 Forbidden. For AJAX/React: the CSRF token can be sent as an `X-CSRFToken` header (Django looks for this header in non-GET requests). In practice, a React frontend fetches the CSRF cookie (`csrftoken`) and includes it in requests. This mechanism prevents cross-site request forgery by ensuring the request originated from your site [21].

- **What is REST API versioning and how do you implement it?** Versioning ensures old clients aren't broken by API changes. Common strategies: **URL versioning** (e.g. `/api/v1/users/`), **query parameter** (`/api/users?version=1`), or **header versioning** (custom header like `Accept: application/vnd.myapp.v1+json`). URL versioning is explicit and widely used (most RESTful). Query or header versioning can be cleaner for clients. In DRF, you can configure `DEFAULT_VERSIONING_CLASS` (e.g. `NamespaceVersioning`, `URLPathVersioning`) to handle versions. Each method has trade-offs, but typically URL versioning is most straightforward while header versioning hides the versioning detail from URLs [22].

- **How do you paginate results in Django REST Framework?** DRF offers built-in pagination classes. Examples: `PageNumberPagination` (adds `?page=` parameter), `LimitOffsetPagination` (`?limit=` and `?offset=`), and `CursorPagination` (opaque cursor for large datasets). In settings you set `DEFAULT_PAGINATION_CLASS` (e.g. `PageNumberPagination`) and `PAGE_SIZE`. In views, you can also set `pagination_class`. For example, `PageNumberPagination` will return paginated JSON with `next`/`previous` links and results for the current page. `LimitOffsetPagination` allows clients to control page size and offset. `CursorPagination` is efficient for deep pagination by avoiding counting and offset (it encodes position in a cursor). The choice depends on performance needs: simple cases use page numbers; very large/query-sensitive sets use cursor for performance [23].

- **How are errors handled in DRF?** DRF catches common exceptions (like `Http404`, `PermissionDenied`, and DRF's own `APIException` subclasses) in its views and returns appropriate HTTP responses. By default, an error response is JSON with a `"detail"` field. For example, if a `DELETE` is not allowed, DRF returns HTTP 405 with body `{"detail": "Method 'DELETE' not allowed."}` [19]. Validation errors (from serializers) return HTTP 400 with a JSON object mapping field names to lists of error messages, e.g. `{"name": ["This field may not be blank."]}` [20]. You can customize the error format by writing a custom exception handler: define a function that modifies the `response` (for example, adding the status code to the JSON) and set `REST_FRAMEWORK['EXCEPTION_HANDLER']` to point to it [24] [25].

- **What are the differences between REST and GraphQL?** REST and GraphQL are two API approaches. **REST** uses multiple endpoints (e.g. `/users`, `/posts/`) and fixed data structures per endpoint. Clients might over-fetch (receiving more data than needed) or under-fetch (needing multiple requests for nested data). **GraphQL** provides a single endpoint and a query language: the client specifies exactly which fields it needs, and the server returns only that data. GraphQL solves the over-fetching problem by letting clients request just the needed data in one query. In contrast, RESTful design is resource/URL-based, while GraphQL is query-based [26] [27]. GraphQL typically uses `query`, `mutation`, and `subscription` operations over HTTP (usually POST), whereas REST uses standard HTTP verbs (GET/POST/PUT/DELETE) on different URLs. Security, caching, and monitoring differ: REST can leverage HTTP caching on endpoints; GraphQL often needs different caching strategies. In essence, GraphQL is more flexible and performant for clients needing varied data, at the cost of a more complex schema definition [26] [27].

- **How do you interact with a Django API from a React frontend?** In React you can fetch data from the Django REST API using `fetch` or libraries like `axios`. Typically you call the API inside `useEffect` (to run on mount) and store the result in state. Example with `fetch`:

```
useEffect(() => {
  fetch('/api/authors/')
    .then(res => res.json())
    .then(data => setAuthors(data));
}, []);
```

The code above runs once after render, requests JSON from the `/api/authors/` endpoint, and sets it in component state. You then render `authors` from state. You must handle loading/error states as needed. Alternatively, use `async/await` in `useEffect`. The key idea is React calls the API (ensuring any needed auth token or CSRF header) and updates the UI with the response [28].

- **How is security handled in a React/Django app (CSRF, JWT, OAuth)?** For CSRF, Django's CSRF protection requires a valid token on unsafe requests. With React as a separate client, you typically read the CSRF cookie (`csrftoken`) that Django sets and include it in an `X-CSRFToken` header on POST/PUT/DELETE requests. This satisfies Django's check [21].

For token-based auth, JSON Web Tokens (JWT) are common: upon login, Django returns a signed JWT, and React stores it (e.g. in `localStorage` or an `HttpOnly` cookie). Subsequent API calls include `Authorization: Bearer <token>`. On the backend, libraries (like [django-rest-framework-simplejwt]) verify the token's signature and expiry. JWT is stateless and useful for SPAs, but you must handle token refresh securely (short-lived access token + refresh token mechanism).

For OAuth, typical pattern is using an OAuth provider (Google, Facebook, etc.) with Django-allauth or a similar library. You redirect the user to the provider, get back an auth code, exchange it on the server for a user identity, and then issue your own session or token. In React, you often use a popup or redirect flow to the OAuth URL. In interviews, just note that OAuth is for delegated identity, JWT is for stateless auth, and Django sessions (default auth) relies on cookies + CSRF.

- **How can you optimize performance in React and Django?** In React, performance tips include:
- **Code-splitting & lazy loading**: use `React.lazy` / `Suspense` or dynamic `import()` to load parts of the app on-demand (shown above) [9] [10].
- **Memoization**: use `React.memo`, `useCallback`, and `useMemo` to avoid unnecessary re-renders when props/state haven't changed [11].
- **Avoid anonymous functions/objects in render** so child components can skip re-render via shallow prop check.
- **Optimize images and assets**: serve compressed images and use lazy-loading for images.

In Django, key optimizations include: - **Database query optimization**: use `select_related` and `prefetch_related` to fetch related data in fewer queries [17] [18]. Add indexes on frequently filtered fields. - **Caching**: Django provides caching (per-site, per-view, template fragment, low-level) to store expensive computations or pages [29]. For example, use `cache_page` decorator or the low-level cache API to store querysets or rendered templates. - **Profiling**: identify slow queries (use `django-debug-toolbar` or database logs). - **Serving static/media efficiently**: use a CDN or `whitenoise` in production to serve static React builds and media files. - **Query optimizations**: avoid heavy aggregation in view loops. Use pagination for large querysets.

- **DevOps/Deployment Considerations:** Deploying a React/Django app often involves bundling the React app (e.g. building static files) and serving them via a web server (Nginx) while running Django (with Gunicorn or uWSGI) behind it. CI/CD pipelines (GitHub Actions, Jenkins, etc.) can automate testing and deployment. Use Docker to containerize both front and back ends or use services like Heroku/AWS. For Django, collect static files (`collectstatic`) and run migrations as part of deployment. Environment variables (for secrets, DB config) should be used instead of hardcoding. Monitor performance (e.g. use Sentry/Datadog for errors, New Relic for performance). Ensure secure

settings (HTTPS, secure cookies). While not typically quizzed in detail, understanding the end-to-end deployment pipeline (containerization, continuous integration, webserver config, etc.) shows readiness for production environments.

Sources: Concepts and examples are drawn from official and community resources, including GeeksforGeeks, DRF documentation, and React best practices [1] [7] [12] [21], among others. Each answer includes relevant citations for reference.

---

[1] [2] [3] [4] [5] [11] [28] React Hooks Interview Questions & Answers - 2025 - GeeksforGeeks
https://www.geeksforgeeks.org/reactjs/top-react-hooks-interview-questions-answers/

[6] React Interview Questions and Answers - GeeksforGeeks
https://www.geeksforgeeks.org/reactjs/react-interview-questions/

[7] [8] Context API vs. Redux : Which One For your Next Project - GeeksforGeeks
https://www.geeksforgeeks.org/blogs/context-api-vs-redux-api/

[9] [10] What is lazy loading in React?
https://www.designgurus.io/answers/detail/what-is-lazy-loading-in-react

[12] [13] [14] [22] [23] Top 30 Most Common django rest framework interview questions You Should Prepare For
https://www.vervecopilot.com/interview-questions/top-30-most-common-django-rest-framework-interview-questions-you-should-prepare-for

[15] [16] [21] [29] Top 50 Django Interview Questions and Answers - GeeksforGeeks
https://www.geeksforgeeks.org/python/django-interview-questions/

[17] [18] Optimizing Django Queries with select_related and prefetch_related | by Mehedi Khan | Django Unleashed | Medium
https://medium.com/django-unleashed/optimizing-django-queries-with-select-related-and-prefetch-related-e404af72e0eb

[19] [20] [24] [25] Exceptions - Django REST framework
https://www.django-rest-framework.org/api-guide/exceptions/

[26] [27] GraphQL vs REST API - Difference Between API Design Architectures - AWS
https://aws.amazon.com/compare/the-difference-between-graphql-and-rest/