## 1. HOW HTTP APPLICATIONS PRESERVE THE STATE OF AN APPLICATION ACROSS MULTIPLE REQUEST-RESPONSE CYCLES.

HTTP is a stateless protocol, meaning it does not retain any memory of previous interactions between a client and a server. To preserve state across multiple request–response cycles (especially for authentication and session management), web applications rely on mechanisms such as cookies, server-side sessions, and token-based authentication. Cookies typically store a unique session ID in the user's browser, allowing the server to associate incoming requests with a stored session object that maintains user-specific data, such as login status (Django uses this approach through features like *"request. COOKIES*" and built-in session middleware).

This aligns with the explanation in the Django course material, which describes how incoming requests include data such as cookies and other metadata that the application can read to determine user context. Frameworks like Django then store session data securely on the server while the browser only keeps the session identifier. In modern systems, token-based methods such as JSON Web Tokens (JWTs) can also be used, where the token itself contains signed user information and is sent with each request. Through these mechanisms, applications are able to maintain continuity and authenticated state even though the underlying HTTP protocol remains stateless.

## 2. THE PROCEDURES FOR PERFORMING DJANGO DATABASE MIGRATIONS TO A SERVER-BASED RELATIONAL DATABASE LIKE MARIADB

- To perform Django database migrations to a server-based relational database (e.g., MariaDB) you follow a *"development -> deploy -> run*" pattern:
- develop your models locally and run python manage.py makemigrations to produce migration files and python manage.py migrate to apply them to your local DB (this workflow is described in the project guide).

- Prepare the server DB: install and run the database service (MariaDB), create the target database and a dedicated DB user with appropriate privileges, and optionally set up networking/SSL for remote access.
- Install the Python DB adapter on the server (commonly mysqlclient or PyMySQL for MariaDB/MySQL) and configure Django's DATABASES setting (in settings.py) to use the MySQL backend (django.db.backends.mysql) with the server host, port, name, user, and password; include production options like CONN_MAX_AGE and any OPTIONS required by your adapter.
- Commit your migration files to version control and deploy your code to the server (or run migrations as part of your CI/CD pipeline).
- On the server (or via your deployment process), run python manage.py migrate to apply the committed migrations against the server database; if you need to run against a non-default DB, use --database <alias>.
- For complex schema/data changes, consider a staging environment first, include data migrations (RunPython) within your migration files, and—for production—take a DB backup, put the application into maintenance mode (or scale down traffic), then run migrations; if something goes wrong you can migrate to a previous migration or restore the backup.
- Additionally, ensure the DB user has ALTER/CREATE/DROP privileges needed for schema changes, avoid long blocking migrations during peak traffic (split large changes), and test migrations in staging to verify both schema and data migrations behave correctly before running in production.