SECTION 7 — Utility Layer Deep Dive

JWT, Password, Response, AppError, AsyncHandler

1. What Is the Utility Layer?

The utility layer contains reusable helper functions that:

- Are not tied to one specific module
- Support multiple parts of the system
- Improve code cleanliness
- Reduce duplication

Utilities follow this principle:

Write once, reuse everywhere.

In Monsta, the utility folder contains:

```
utils/
 ├── jwt.util.js
 ├── password.util.js
 ├── response.util.js
 ├── appError.util.js
 ├── asyncHandler.util.js
```

Each utility solves a specific cross-cutting concern.

2. JWT Utility (jwt.util.js)

Purpose

Handles:

- Generating access tokens
- Generating refresh tokens
- Verifying tokens

JWT stands for JSON Web Token.

Why We Use JWT

JWT enables:

- Stateless authentication
- Secure user identity verification
- Scalable session management

Instead of storing sessions in server memory, JWT stores user identity inside the token.

Example Implementation

```
import jwt from "jsonwebtoken";

export const generateAccessToken = (userId) => {
  return jwt.sign(
    { id: userId },
    process.env.JWT_SECRET,
    { expiresIn: "15m" }
  );
};

export const verifyAccessToken = (token) => {
  return jwt.verify(token, process.env.JWT_SECRET);
};
```

How It Works

1. User logs in.
2. Backend generates token.
3. Token sent to client.
4. Client sends token in Authorization header.
5. Backend verifies token before allowing access.

JWT utility centralizes this logic.

3. Password Utility (password.util.js)

Purpose

Handles:

- Password hashing
- Password comparison

We never store plain passwords.

Why Hash Passwords?

If database is compromised:

- Plain passwords expose user accounts.
- Hashed passwords remain protected.

Hashing ensures security.

Example Implementation

```
import bcrypt from "bcryptjs";

export const hashPassword = async (password) => {
  const salt = await bcrypt.genSalt(10);
  return bcrypt.hash(password, salt);
};

export const comparePassword = async (password, hashedPassword) => {
  return bcrypt.compare(password, hashedPassword);
};
```

Workflow

Register:

- Hash password
- Store hashed password

Login:

- Compare input password with stored hash

This utility prevents repeating bcrypt logic everywhere.

4. Response Utility (response.util.js)

Purpose

Ensures consistent success response format.

Enterprise systems never return random response formats.

Example Implementation

```
export const successResponse = (
  res,
  data = null,
  message = "Success",
  statusCode = 200
) => {
  return res.status(statusCode).json({
    success: true,
    message,
    data
  });
};
```

Why This Is Important

Instead of writing:

res.status(200).json({ user });

We write:

successResponse(res, user, "User fetched");

Benefits:

- Consistency
- Standardized structure
- Easier frontend integration
- Cleaner controllers

5. AppError Utility (appError.util.js)

Purpose

Creates standardized error objects.

Instead of:

throw new Error("Invalid credentials");

We use:

throw new AppError("Invalid credentials", 401);

Example Implementation

```
class AppError extends Error {
  constructor(message, statusCode, errorCode = null) {
    super(message);
    this.statusCode = statusCode;
    this.errorCode = errorCode;
  }
}

export default AppError;
```

Why Use Custom Error Class?

Because it allows:

- Custom status codes
- Custom error codes
- Structured error handling
- Cleaner error middleware

This is enterprise-level error design.

6. Async Handler Utility (asyncHandler.util.js)

Problem It Solves

Normally, async route handlers require try/catch:

```
try {
  ...
} catch (error) {
  next(error);
}
```

Writing this repeatedly is inefficient.

Solution

Wrap async functions in a utility.

Example Implementation

```
const asyncHandler = (fn) => {
  return (req, res, next) => {
    Promise.resolve(fn(req, res, next)).catch(next);
  };
};
```

export default asyncHandler;

Usage Example

Instead of:

```
export const login = async (req, res, next) => {
  try {
    ...
  } catch (error) {
    next(error);
  }
};
```

We write:

```
export const login = asyncHandler(async (req, res) => {
  ...
});
```

Cleaner and safer.

7. How All Utilities Work Together

Example: Login Process

```
Controller
  ↓
Password Utility (compare)
  ↓
JWT Utility (generate token)
  ↓
Response Utility (send response)
  ↓
If error → AppError
  ↓
AsyncHandler forwards to error middleware
```

Utilities cooperate to keep controllers minimal.

8. Why Utilities Matter in Enterprise Applications

Utilities:

- Reduce duplication
- Improve maintainability
- Standardize behavior
- Improve readability
- Simplify debugging
- Support scalability

Enterprise systems always separate reusable logic.

9. What Students Must Understand

1. Utilities are not business logic.
2. Utilities support multiple modules.
3. JWT logic should not be inside controllers.
4. Password hashing should not be repeated.
5. Response format must be consistent.
6. Error objects should be structured.
7. Async error handling should be centralized.

10. Summary

By the end of this section, students should understand:

- What utilities are
- Why they exist
- How JWT works
- Why passwords are hashed
- Why consistent responses matter
- How AppError structures errors
- How asyncHandler removes repetitive try/catch

This prepares students to understand centralized error handling in the next section.