**import 'dotenv/config';** This imports the dotenv package to load environment variables from a .env file into process.env. It allows you to manage configuration settings, like port numbers, without hardcoding them in your application.

**Handling Uncaught Exceptions:** process.on('uncaughtException', (err) => { });This event listener catches exceptions that weren't handled elsewhere in the application.

**Handling Unhandled Promise Rejections:** process.on('unhandledRejection', (err) => {});This event listener handles promise rejections that weren't caught.

**Graceful Shutdown on SIGTERM:** process.on('SIGTERM', () => { }); The listener for the SIGTERM signal (often sent by process managers or when you manually terminate the app) logs a message and closes the server gracefully, ensuring that ongoing requests can finish before the process exits.

process.on is a method in Node.js that allows you to listen for specific events that occur in the process.

 **@babel/polyfill**: Provides support for new JavaScript features in older environments by emulating missing functionality.

 **axios**: A promise-based HTTP client for making requests to servers (e.g., GET, POST) in Node.js and the browser.

 **bcryptjs**: A library for hashing passwords using the bcrypt algorithm for secure storage.

 **body-parser**: Middleware for parsing incoming request bodies in JSON or URL-encoded formats.

 **casual**: A library to generate random data for testing or seeding databases.

 **compression**: Middleware that compresses HTTP responses to reduce payload size and improve performance.

 **cookie-parser**: Middleware that parses cookies from the HTTP request header.

 **cors**: Middleware that enables Cross-Origin Resource Sharing (CORS), allowing servers to handle requests from different domains.

 **dotenv**: Loads environment variables from a .env file into process.env to manage configuration.

 **express**: A web framework for building RESTful APIs and web applications in Node.js.

 **express-rate-limit**: Middleware that limits repeated requests from the same IP to protect against DoS attacks.

 **express-session**: Middleware for managing user sessions and storing session data on the server.

 **helmet**: Middleware that adds security headers to protect Express apps from common vulnerabilities.

 **hpp**: Middleware to protect against HTTP Parameter Pollution (HPP) attacks.

 **html-to-text**: Converts HTML content to plain text, often used for email or other plain text processing.

 **jsonwebtoken**: A library for generating and verifying JSON Web Tokens (JWT) for authentication and authorization.

 **knex**: A SQL query builder for Node.js that supports various database engines, like MySQL and PostgreSQL.

 **morgan**: Middleware that logs HTTP requests in the console for easier debugging and monitoring.

 **multer**: Middleware for handling file uploads in Express applications.

 **mysql2**: A MySQL client for Node.js that provides promises-based and performant database querying.

 **nodemailer**: A library for sending emails from Node.js applications.

 **nodemon**: A development tool that automatically restarts the server when files change, improving the development workflow.

 **passport**: Authentication middleware for Node.js that integrates various authentication strategies.

 **passport-github2**: A GitHub authentication strategy for Passport to allow GitHub OAuth login.

 **pug**: A template engine for Node.js that helps generate HTML with a simplified syntax.

 **sharp**: A high-performance image processing library, commonly used for resizing and converting images.

 **slugify**: Converts strings into URL-friendly slugs, typically used for SEO-friendly URLs.

 **stripe**: A Node.js SDK for integrating Stripe's payment processing services into applications.

 **uniqid**: A library to generate unique IDs, often used for identifying resources like database records.

 **validator**: A library for validating and sanitizing strings (e.g., emails, URLs, etc.).

 **xss-clean**: Middleware to prevent Cross-Site Scripting (XSS) attacks by sanitizing input.

Explanation of these code:

* *🟨const* multerStorage = multer.memoryStorage();
* *🟨const* multerFilter = (*req*, *file*, *cb*) => {
* if (file.mimetype.startsWith('image')) {
* cb(null, true);
* } else {
* cb(new AppError('Not an image! Please upload only images.', 400), false);
* }};
* *🟨const* upload = multer({
* storage: multerStorage,
* fileFilter: multerFilter,
* });
* *🟨const* uploadUserPhoto = upload.single('photo');
* *🟨const* resizeUserPhoto = catchAsync(*async* (*req*, *res*, *next*) => {
* if (!req.file) return next();
* req.file.filename = `user-${req.user.id}-${Date.now()}.jpeg`;
* await sharp(req.file.buffer)
* .resize(500, 500)
* .toFormat('jpeg')
* .jpeg({ quality: 90 })
* .toFile(`public/img/users/${req.file.filename}`);
* next();
* });
* *🟨const filterObj = (obj, ...allowedFields) => {*
* *const newObj = {};*
* *Object.keys(obj).forEach((el) => {*
* *if (allowedFields.includes(el)) newObj[el] = obj[el];*
* *});*
* *return newObj;*
* *};*

**1. const multerStorage = multer.memoryStorage();**

* **Purpose**: This line sets up a storage option for handling file uploads using the multer library.
* **Explanation**:
  + multer.memoryStorage() tells multer to keep the file in memory as a Buffer rather than saving it to disk immediately.
  + This is often useful when you want to process the file in memory before saving it somewhere (e.g., compressing or resizing an image).

**2. const multerFilter = (req, file, cb) => { ... };**

* **Purpose**: This function is used to filter the types of files being uploaded.
* **Explanation**:
  + file.mimetype.startsWith('image'): Checks if the file being uploaded is an image (e.g., image/png, image/jpeg).
  + If the file is an image, it calls cb(null, true) to indicate the file is accepted.
  + If the file is not an image, it calls cb(new AppError('Not an image! Please upload only images.', 400), false); which triggers an error and rejects the file upload.
  + This is a security measure to prevent non-image files from being uploaded.

**3. const upload = multer({ storage: multerStorage, fileFilter: multerFilter });**

* **Purpose**: Configures the multer middleware for handling file uploads with the defined storage and fileFilter options.
* **Explanation**:
  + storage: multerStorage: Sets the storage option to use memoryStorage, so the files will be held in memory.
  + fileFilter: multerFilter: Uses the multerFilter defined above to only allow image files to be uploaded.
  + This creates an upload instance that can be used to handle uploads in different ways (e.g., single, multiple).

**4. const uploadUserPhoto = upload.single('photo');**

* **Purpose**: Sets up a middleware to handle single image uploads for a field named 'photo'.
* **Explanation**:
  + upload.single('photo') creates a middleware that expects a single file upload with the field name photo (as defined in the form or frontend request).
  + This middleware will process the file, apply the multerFilter, and then store it in memory (as configured by multerStorage).

**5. const resizeUserPhoto = catchAsync(async (req, res, next) => { ... });**

* **Purpose**: This function processes and resizes the uploaded user photo before saving it to disk.
* **Explanation**:
  + **Check if a file exists**: If no file was uploaded (if (!req.file) return next();), it skips to the next middleware.
  + **Generate a filename**: Creates a filename based on the user’s ID and the current timestamp (e.g., user-1234-1609459200000.jpeg).
  + **Resize and save the image**:
    - sharp(req.file.buffer): Uses the sharp library to process the image held in memory (req.file.buffer).
    - .resize(500, 500): Resizes the image to be 500x500 pixels.
    - .toFormat('jpeg').jpeg({ quality: 90 }): Converts the image to JPEG format with 90% quality.
    - .toFile(...): Saves the processed image to the specified directory (public/img/users/).
  + **Pass the filename to the next middleware**:
    - The processed image's filename is stored in req.file.filename so it can be accessed in subsequent middlewares.
    - The function then calls next() to continue to the next middleware or route handler.

**Summary**

* These functions work together to handle the entire image upload process for a user profile picture:
  1. **multerStorage**: Stores files in memory.
  2. **multerFilter**: Filters and only allows image uploads.
  3. **upload**: Configures the upload with the above settings.
  4. **uploadUserPhoto**: Middleware to handle single file uploads with the field name photo.
  5. **resizeUserPhoto**: Processes the uploaded image, resizes it, and saves it to disk with a new filename.

**What It Does filterObj:**

1. **Takes an object and a list of allowed fields as input**:
   * obj: The object from which we want to filter properties.
   * ...allowedFields: A list of property names (passed as arguments) that should be retained in the new object.
2. **Creates an empty newObj**:
   * const newObj = {};
   * This is the new object that will contain only the properties that match the allowedFields.
3. **Iterates over the keys of obj**:
   * Object.keys(obj).forEach((el) => { ... });
   * Object.keys(obj) returns an array of the keys (property names) in the obj object.
   * forEach((el) => { ... }) loops through each key (el).
4. **Checks if the current key is in the allowedFields**:
   * if (allowedFields.includes(el)): Checks if the current key (el) is included in the array allowedFields.
   * If it is, it means that this is an allowed property and should be copied to newObj.
5. **Copies the property to newObj**:
   * newObj[el] = obj[el];
   * This copies the value of the current property (el) from obj to newObj.
6. **Returns the new object**:
   * return newObj;
   * After all allowed fields are copied, the function returns the newObj.