This is a Java class named `VoterJwtAuthenticationFilter` that extends `OncePerRequestFilter`. It is designed to handle JWT (JSON Web Token) authentication for requests in a web application. Let's go through each part of the code and explain its purpose:

1. `@Component`: This annotation marks the class as a Spring bean, allowing it to be automatically detected and registered in the Spring context during component scanning.

2. `@RequiredArgsConstructor`: This Lombok annotation automatically generates a constructor with required arguments for the class fields. In this case, it will create a constructor with arguments for `jwtService` and `userDetailsService`.

3. `public class VoterJwtAuthenticationFilter extends OncePerRequestFilter`: The class `VoterJwtAuthenticationFilter` extends `OncePerRequestFilter`, which ensures that the `doFilterInternal` method is executed only once for each request.

4. `private final VoterJwtService jwtService`: This field is of type `VoterJwtService`, which presumably handles JWT-related operations such as token validation and extraction of user details from the token.

5. `private final UserDetailsService userDetailsService`: This field is of type `UserDetailsService`, which is an interface provided by Spring Security for managing user details and authentication.

6. `protected void doFilterInternal(...) throws ServletException, IOException`: This is the main method of the class responsible for filtering incoming requests and handling JWT authentication.

7. `if (request.getServletPath().contains("/auth/p2/**"))`: This condition checks if the request path contains "/auth/p2/". If it does, the request is passed through the filter chain without further processing. It suggests that requests to "/auth/p2/" are not subject to JWT authentication.

8. `final String authHeader = request.getHeader("Authorization");`: This line fetches the value of the "Authorization" header from the incoming request, which is where the JWT token should be present.

9. `final String jwt;`: This variable will hold the extracted JWT token after parsing the "Authorization" header.

10. `final String userEmail;`: This variable will store the user's email address extracted from the JWT token.

11. `if (authHeader == null || !authHeader.startsWith("Bearer "))`: This condition checks if the "Authorization" header is absent or doesn't start with the "Bearer " prefix, indicating that the request does not contain a JWT token. If true, the request is passed through the filter chain without further processing.

12. `jwt = authHeader.substring(7);`: This line extracts the JWT token by removing the "Bearer " prefix from the "Authorization" header. The substring starts at index 7 to skip the "Bearer " part.

13. `userEmail = jwtService.extractUsername(jwt);`: This line calls the `extractUsername` method of `jwtService`, presumably to extract the user's email from the JWT token.

14. `if (userEmail != null && SecurityContextHolder.getContext().getAuthentication() == null)`: This condition checks if the `userEmail` is not null (i.e., a valid email was extracted from the token) and also ensures that there is no existing authentication in the security context.

15. `UserDetails userDetails = this.userDetailsService.loadUserByUsername(userEmail);`: This line retrieves the `UserDetails` object for the given `userEmail` from the `userDetailsService`. The `loadUserByUsername` method is responsible for fetching user details based on the username (in this case, the user's email address).

16. `if (jwtService.isTokenValid(jwt, userDetails))`: This condition checks if the JWT token is valid by calling the `isTokenValid` method of `jwtService` and passing the token and `userDetails`.

17. `UsernamePasswordAuthenticationToken authToken = new UsernamePasswordAuthenticationToken(userDetails, null, userDetails.getAuthorities());`: If the token is valid, this line creates an `Authentication` token called `authToken` using the `userDetails` object and the user's authorities (roles/permissions) obtained from `userDetails.getAuthorities()`.

18. `authToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));`: This line sets additional details for the `authToken` object, such as the IP address and session ID of the current request. These details are captured using `WebAuthenticationDetailsSource` and are useful for auditing and logging purposes.

19. `SecurityContextHolder.getContext().setAuthentication(authToken);`: This sets the `authToken` as the authenticated `Authentication` object in the security context, which effectively logs the user in.

20. `filterChain.doFilter(request, response);`: Finally, the request is passed along the filter chain for further processing by other filters or to the application's endpoint if no other filters are present.

Overall, this class is responsible for checking the incoming request for a valid JWT token, validating it, and authenticating the user if the token is valid. If the request path contains "/auth/p2/", the filter allows the request to pass without authentication. Otherwise, it performs JWT authentication using the provided `jwtService` and `userDetailsService`.