Okay, this is a solid approach for collecting page visit metrics while ensuring data privacy and control, especially given your concern about sensitive data and avoiding Google Analytics. Using a **Spring Interceptor (specifically a HandlerInterceptor)** on the backend is an excellent way to do this.

Here's a detailed breakdown of how to implement an interceptor-based solution to track page visits in your Spring/Hibernate backend, assuming your frontend is JavaScript-based (e.g., a Single Page Application - SPA, or a traditional multi-page app).

Core Idea

The Spring `HandlerInterceptor` allows you to intercept requests *before* they reach your controller, *after* they've been handled by the controller, and *after* the view has been rendered. For page visit tracking, the `preHandle()` method is usually the most suitable.

Implementation Steps

1\. Define the Metric Data Model (Optional but Recommended for Clarity)

While you could just log directly, having a simple entity helps if you ever want to store more complex metrics or query them easily.

```java
// src/main/java/com/yourcompany/yourapp/metrics/models/PageView.java
package com.yourcompany.yourapp.metrics.models;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import java.time.LocalDateTime;

@Entity
@Table(name = "page_views") // Optional: define table name
public class PageView {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String userId; // Or sessionId, or actual username if authenticated
    private String pageUrl;
    private String userAgent; // Optional: Browser and OS info
```

```
    private String ipAddress; // Optional: User's IP address
    private LocalDateTime timestamp;

    // Constructors
    public PageView() {
        this.timestamp = LocalDateTime.now();
    }

    public PageView(String userId, String pageUrl, String userAgent, String ipAddress) {
        this(); // Call default constructor to set timestamp
        this.userId = userId;
        this.pageUrl = pageUrl;
        this.userAgent = userAgent;
        this.ipAddress = ipAddress;
    }

    // Getters and Setters
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
    public String getUserId() { return userId; }
    public void setUserId(String userId) { this.userId = userId; }
    public String getPageUrl() { return pageUrl; }
    public void setPageUrl(String pageUrl) { this.pageUrl = pageUrl; }
    public String getUserAgent() { return userAgent; }
    public void setUserAgent(String userAgent) { this.userAgent = userAgent; }
    public String getIpAddress() { return ipAddress; }
    public void setIpAddress(String ipAddress) { this.ipAddress = ipAddress; }
    public LocalDateTime getTimestamp() { return timestamp; }
    public void setTimestamp(LocalDateTime timestamp) { this.timestamp = timestamp; }

    @Override
    public String toString() {
        return "PageView{" +
                "id=" + id +
                ", userId='" + userId + '\'' +
                ", pageUrl='" + pageUrl + '\'' +
                ", userAgent='" + userAgent + '\'' +
                ", ipAddress='" + ipAddress + '\'' +
                ", timestamp=" + timestamp +
                '}';
    }
}
```

2\. Create a Repository for Persistence

```java
// src/main/java/com/yourcompany/yourapp/metrics/repositories/PageViewRepository.java
package com.yourcompany.yourapp.metrics.repositories;

import com.yourcompany.yourapp.metrics.models.PageView;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface PageViewRepository extends JpaRepository<PageView, Long> {
    // Spring Data JPA provides basic CRUD operations automatically
}
```

3\. Implement the HandlerInterceptor

This is the core logic.

```java
// src/main/java/com/yourcompany/yourapp/metrics/interceptors/PageViewInterceptor.java
package com.yourcompany.yourapp.metrics.interceptors;

import com.yourcompany.yourapp.metrics.models.PageView;
import com.yourcompany.yourapp.metrics.repositories.PageViewRepository;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;

import java.time.LocalDateTime;

@Component
public class PageViewInterceptor implements HandlerInterceptor {

    private final PageViewRepository pageViewRepository;

    // Use constructor injection for dependencies
    @Autowired
    public PageViewInterceptor(PageViewRepository pageViewRepository) {
        this.pageViewRepository = pageViewRepository;
```

```
    }

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler) throws Exception {
        // This method is called before the controller's handler method is executed.
        // It's a good place to capture the start of a "page visit".

        // --- Data Collection Logic ---
        String userId = null; // Default to null

        // 1. Get User ID (if authenticated)
        // If you're using Spring Security:
        if (request.getUserPrincipal() != null) {
            userId = request.getUserPrincipal().getName(); // Get username or user ID
        } else {
            // If not authenticated, use session ID or a cookie-based identifier
            // For true user tracking, consider a persistent cookie ID that maps to a "user"
            userId = request.getSession().getId(); // Example: Using session ID
        }

        // 2. Get Page URL
        String pageUrl = request.getRequestURI(); // Full path of the request
        // If it's an SPA and all requests go to /index.html, you might need
        // a custom header from the frontend (see section below)

        // 3. Get User Agent (Browser/OS Info)
        String userAgent = request.getHeader("User-Agent");

        // 4. Get IP Address
        String ipAddress = request.getRemoteAddr(); // Basic, might need proxy awareness for
production

        // --- Create and Save PageView Object ---
        PageView pageView = new PageView(userId, pageUrl, userAgent, ipAddress);
        pageView.setTimestamp(LocalDateTime.now()); // Ensure timestamp is current

        // Save asynchronously to avoid blocking the request thread
        // For simple apps, direct save is fine. For high traffic, consider:
        // - A separate thread pool for logging
        // - A message queue (Kafka, RabbitMQ)
        // - An event-driven approach
        new Thread(() -> {
            try {
```

```
                pageViewRepository.save(pageView);
                // System.out.println("Logged PageView: " + pageView.getPageUrl() + " by " +
pageView.getUserId());
            } catch (Exception e) {
                System.err.println("Error saving page view: " + e.getMessage());
                // Log this error properly with a logger (e.g., SLF4J/Logback)
            }
        }).start();


        // Return true to allow the request to proceed to the controller.
        // Return false to stop the request (e.g., if you want to block based on some condition).
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object
handler, ModelAndView modelAndView) throws Exception {
        // This method is called after the controller's handler method is executed,
        // but before the view is rendered.
        // Less common for simple page visit tracking unless you need to manipulate the model.
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response,
Object handler, Exception ex) throws Exception {
        // This method is called after the complete request has finished,
        // including view rendering. Useful for cleanup or final logging.
    }
}
```

4\. Register the Interceptor

You need to tell Spring to use your `PageViewInterceptor`.

```java
// src/main/java/com/yourcompany/yourapp/config/WebConfig.java
package com.yourcompany.yourapp.config;

import com.yourcompany.yourapp.metrics.interceptors.PageViewInterceptor;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
```

```
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class WebConfig implements WebMvcConfigurer {

    private final PageViewInterceptor pageViewInterceptor;

    @Autowired
    public WebConfig(PageViewInterceptor pageViewInterceptor) {
        this.pageViewInterceptor = pageViewInterceptor;
    }

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        // Apply the interceptor to all incoming requests
        registry.addInterceptor(pageViewInterceptor);

        // Or, apply to specific paths:
        // registry.addInterceptor(pageViewInterceptor).addPathPatterns("/api/**");
        // registry.addInterceptor(pageViewInterceptor).excludePathPatterns("/static/**", "/css/**");
    }
}
```

5\. Frontend (JavaScript) Considerations (Crucial for SPAs)

If your frontend is a **Single Page Application (SPA)**, the user might navigate between "pages" (views) without making a full new request to the backend. The `request.getRequestURI()` in your interceptor will only reflect the API calls or the initial `/index.html` load.

To track actual page views in an SPA, you need to:

* **Send a dedicated "page view" event from the frontend to the backend** whenever the user navigates to a new "logical page" in the SPA.
* This would typically be a lightweight GET or POST request to a dedicated `/api/trackPageView` (or similar) endpoint.

Example Frontend (React/Vue/Angular - Pseudo-code):

```javascript
// In your SPA router's navigation guard or component mount:

// Example with a router
router.afterEach((to, from) => {
```

```
  // `to.path` would be the logical page URL like /dashboard, /settings, etc.
  sendPageViewToBackend(to.path);
});

function sendPageViewToBackend(pagePath) {
  fetch('/api/trackPageView', {
    method: 'POST', // Or GET if no sensitive data in body
    headers: {
      'Content-Type': 'application/json',
      'X-Page-Path': pagePath // Custom header to send the SPA path
    },
    // body: JSON.stringify({ pagePath: pagePath }) // If using POST body
  })
  .then(response => {
    // Handle response if needed, but often fire-and-forget
    if (!response.ok) {
      console.error('Failed to send page view:', response.statusText);
    }
  })
  .catch(error => {
    console.error('Network error sending page view:', error);
  });
}
```

Modified Backend Interceptor for SPA:

If you go this route, your interceptor needs to be slightly smarter or you'd use a dedicated controller endpoint.

Option A: Interceptor Catches `X-Page-Path` (Still using interceptor for *all* API calls, but specifically for the `trackPageView` one)

```java
// Modified preHandle in PageViewInterceptor
@Override
public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
    String pageUrl = request.getHeader("X-Page-Path"); // Try to get SPA path first
    if (pageUrl == null || pageUrl.isEmpty()) {
        pageUrl = request.getRequestURI(); // Fallback to API URI for direct API calls
    }

    // ... rest of your data collection and saving logic ...
```

```
    return true;
}
```

Option B: Dedicated Tracking Endpoint (More explicit for SPAs)

You'd create a specific Spring `@RestController` endpoint:

```java
// src/main/java/com/yourcompany/yourapp/metrics/controllers/MetricsController.java
package com.yourcompany.yourapp.metrics.controllers;

import com.yourcompany.yourapp.metrics.models.PageView;
import com.yourcompany.yourapp.metrics.repositories.PageViewRepository;
import jakarta.servlet.http.HttpServletRequest;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody; // If sending path in body
import org.springframework.web.bind.annotation.RequestHeader; // If sending path in header
import org.springframework.web.bind.annotation.RestController;

import java.security.Principal;
import java.time.LocalDateTime;

@RestController
public class MetricsController {

    private final PageViewRepository pageViewRepository;

    @Autowired
    public MetricsController(PageViewRepository pageViewRepository) {
        this.pageViewRepository = pageViewRepository;
    }

    @PostMapping("/api/trackPageView")
    public void trackPageView(@RequestHeader(value = "X-Page-Path", required = false) String
pagePath,
                              Principal principal, // For Spring Security user
                              HttpServletRequest request) {

        String userId = (principal != null) ? principal.getName() : request.getSession().getId();
        String actualPageUrl = (pagePath != null && !pagePath.isEmpty()) ? pagePath :
request.getRequestURI();
```

```
        String userAgent = request.getHeader("User-Agent");
        String ipAddress = request.getRemoteAddr();

        PageView pageView = new PageView(userId, actualPageUrl, userAgent, ipAddress);
        pageView.setTimestamp(LocalDateTime.now());

        // Save asynchronously (recommended for metrics endpoints)
        new Thread(() -> {
            try {
                pageViewRepository.save(pageView);
                // Log success
            } catch (Exception e) {
                // Log error
            }
        }).start();
    }
}
```

Advantages of the Interceptor Approach

 * **Centralized Logging:** All page view logic is in one place.
 * **Security & Privacy:** You control *exactly* what data is collected and how it's stored, keeping sensitive information in-house. This is crucial for compliance (GDPR, CCPA, etc.).
 * **Performance:** The `preHandle` method is efficient. Saving data asynchronously ensures it doesn't block the main request thread.
 * **Flexibility:** You can easily add more data points (e.g., screen resolution, browser version) if needed.
 * **No Third-Party Data Sharing:** You're not sending any data to Google or any other third party.

Hibernate Configuration

Ensure your `application.properties` (or `application.yml`) is correctly configured for Hibernate and your database.

```properties
# application.properties
spring.datasource.url=jdbc:mysql://localhost:3306/your_database_name?useSSL=false&serverTimezone=UTC
spring.datasource.username=your_username
spring.datasource.password=your_password
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

```
spring.jpa.hibernate.ddl-auto=update # Use 'update' for development, 'validate' or 'none' for production
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect # Or appropriate dialect for your DB
```

Considerations for "Sensitive Data"

* **What is sensitive?** Define clearly what data should *not* be collected (e.g., PII like names, email addresses, financial details).
* **Data Minimization:** Only collect what's absolutely necessary for your metrics.
* **Anonymization/Pseudonymization:** If you need to track users but can't store actual IDs, consider generating pseudonymous IDs (e.g., a hash of a user's ID or a unique cookie value).
* **User Consent:** If you are tracking users, ensure your application has a clear privacy policy and, if required by law (like GDPR), obtains explicit user consent for tracking.

This robust interceptor-based solution gives you complete control over your user activity metrics while prioritizing data security, which is exactly what you need.
Citations: [[1]](https://github.com/testpress/android-sdk)