# Flow

**Actors in the Example**

1. **Client (Browser)**:
   * URL: https://app.client.com
   * Makes WebAuthn requests to the server and communicates with the authenticator.
2. **Server (Relying Party)**:
   * URL: https://api.server.com
   * Issues the WebAuthn options for registration and authentication and verifies responses.
3. **Authenticator**:
   * Physical or virtual device (e.g., hardware security key, phone, platform authenticator).
   * Stores and manages credentials securely.

**Flow with Example**

**1. Registration**

1. **User Visits Client**:
   * User navigates to https://app.client.com in their browser.
2. **Client Requests Registration Options from Server**:
   * https://app.client.com sends a request to https://api.server.com/register to get registration options.
   * The server generates:
     + A **challenge**.
     + Relying Party ID (rpId) = server.com.
     + Origin (https://app.client.com).
3. **Client Invokes WebAuthn API**:
   * The client (browser) calls:

navigator.credentials.create({ publicKey: registrationOptions });

* + Browser verifies that:
    - The request originates from https://app.client.com.
    - The rpId matches the suffix of app.client.com (it does, because rpId = server.com).

1. **Authenticator Registers**:
   * The authenticator:
     + Stores the credential, binding it to rpId = server.com.
     + Signs the challenge to prove possession of the private key.
2. **Server Verifies Registration**:
   * https://api.server.com/register verifies:
     + Challenge matches.
     + Origin (https://app.client.com) is valid.
     + Credential is bound to server.com.

**2. Authentication**

1. **User Visits Client**:
   * User navigates to https://app.client.com to log in.
2. **Client Requests Authentication Options from Server**:
   * https://app.client.com sends a request to https://api.server.com/authenticate.
   * The server generates:
     + A **challenge**.
     + Relying Party ID (rpId) = server.com.
     + Origin (https://app.client.com).
3. **Client Invokes WebAuthn API**:
   * The client (browser) calls:

navigator.credentials.get({ publicKey: authenticationOptions });

* + Browser verifies:
    - Request is from https://app.client.com.
    - rpId matches the credential's rpId (server.com).

1. **Authenticator Authenticates**:
   * Authenticator verifies:
     + Credential is bound to rpId = server.com.
   * Signs the challenge.
2. **Server Verifies Authentication**:
   * https://api.server.com/authenticate verifies:
     + Challenge matches.
     + Origin (https://app.client.com) is valid.
     + Signature is valid for the credential.

**Origin in Action**

* The **origin** is always the domain and protocol of the client making the WebAuthn API call (e.g., https://app.client.com).
* The **relying party ID (rpId)** is defined by the server and must match or be a suffix of the origin.
* The browser ensures the request is valid by checking that the origin matches the rpId.

**Example Configuration**

**Registration Options:**

Sent by the server to the client:

json

Copy code

{

"rp": {

"id": "server.com",

"name": "My App"

},

"user": {

"id": "dXNlcmlkMTIz", // Base64 encoded user ID

"name": "johndoe",

"displayName": "John Doe"

},

"challenge": "MTIzNDU=", // Base64 encoded challenge

"pubKeyCredParams": [

{ "type": "public-key", "alg": -7 }

],

"timeout": 60000

}

**Verification on Server:**

When verifying responses:

* The server ensures:
  + origin = https://app.client.com.
  + rpId = server.com.
  + Challenge matches.

**Summary of Relationships**

* **Origin**: The browser verifies it matches the source of the WebAuthn call (https://app.client.com).
* **Relying Party ID**: Authenticator binds credentials to this (server.com).
* **Browser Role**: Acts as a gateway, ensuring security and consistency between origin, rpId, and the authenticator.

# What is publicKeyCredentialParameters?

The publicKeyCredentialParameters property is part of the **WebAuthn** specification and is used during the registration process to define which cryptographic algorithms are acceptable for the authenticator to use when generating a credential.

**Definition**

In WebAuthn, publicKeyCredentialParameters is an array of objects, each specifying:

1. **Credential Type (type)**:
   * Specifies the type of credential to be created.
   * Common value: "public-key" (used in WebAuthn for public-key-based authentication).
2. **Algorithm Identifier (alg)**:
   * Indicates the cryptographic algorithm to use for key pair generation and signing.
   * Uses values from the COSE Algorithm registry.

**Structure**

Here’s how publicKeyCredentialParameters is structured in JSON:

json

Copy code

"publicKeyCredentialParameters": [

{ "type": "public-key", "alg": -7 }, // ES256 (ECDSA using SHA-256)

{ "type": "public-key", "alg": -257 } // RS256 (RSA using SHA-256)

]

**Supported Algorithms**

Below are commonly used alg values and their descriptions:

| **Algorithm** | **Value** | **Description** |
| --- | --- | --- |
| ES256 | -7 | ECDSA with SHA-256 |
| RS256 | -257 | RSA with SHA-256 |
| PS256 | -37 | RSASSA-PSS with SHA-256 |
| ES384 | -35 | ECDSA with SHA-384 |

The alg specifies the cryptographic mechanism that the authenticator will use to generate a key pair and sign challenges.

**Usage in Java (WebAuthn4j Example)**

In Java, publicKeyCredentialParameters is defined using the PublicKeyCredentialParameters class. Below is an example of how to construct it in your WebAuthn service:

java

Copy code

private List<PublicKeyCredentialParameters> getPublicKeyCredentialParameters() {

List<PublicKeyCredentialParameters> publicKeyCredentialParameters = new ArrayList<>();

// Add preferred algorithms

publicKeyCredentialParameters.add(new PublicKeyCredentialParameters(

PublicKeyCredentialType.PUBLIC\_KEY, COSEAlgorithmIdentifier.ES256));

publicKeyCredentialParameters.add(new PublicKeyCredentialParameters(

PublicKeyCredentialType.PUBLIC\_KEY, COSEAlgorithmIdentifier.RS256));

return publicKeyCredentialParameters;

}

In this code:

1. The PublicKeyCredentialType.PUBLIC\_KEY constant is used to define the credential type.
2. Algorithms like COSEAlgorithmIdentifier.ES256 and COSEAlgorithmIdentifier.RS256 are used.

**How It's Used**

During **registration**, the publicKeyCredentialParameters array is sent to the authenticator as part of the PublicKeyCredentialCreationOptions. The authenticator evaluates the list and selects the first compatible algorithm it supports to generate a key pair.

**Example Workflow:**

1. The server specifies acceptable algorithms (e.g., ES256 and RS256).
2. The browser sends these options to the authenticator.
3. The authenticator selects one algorithm (e.g., ES256) and generates the key pair.
4. The public key is returned to the server for storage.

**Key Points**

* The order of algorithms in publicKeyCredentialParameters matters; the authenticator picks the first supported algorithm.
* Using multiple algorithms increases compatibility with various authenticators.
* Common practice is to prioritize **ES256** (compact and secure) as the first option.

# What is AttestationConveyancePreference in WebAuthn?

The AttestationConveyancePreference is a property used in **WebAuthn registration** to specify how attestation information (metadata about the authenticator device) should be conveyed from the authenticator to the server. It determines the level of device information provided during the creation of credentials.

**Purpose of Attestation**

Attestation data helps verify:

1. **Device Authenticity**: Confirms the device is genuine and adheres to security standards.
2. **Trust Level**: Ensures the authenticator meets requirements for security-sensitive operations.

However, sharing detailed attestation data might leak information about the user's device or identity, which raises privacy concerns.

**Values of AttestationConveyancePreference**

1. **none**:
   * No attestation data is sent.
   * Focuses on user privacy.
   * Server cannot verify the authenticator’s type or properties.
   * Example use case: Low-security scenarios where device type verification isn’t necessary.
2. **indirect**:
   * Sends attestation data via an intermediary (e.g., a Trusted Attestation Service).
   * The server receives only anonymized data, protecting user privacy.
   * Example use case: Applications that want some assurance about the device type without compromising user privacy.
3. **direct**:
   * Sends the full attestation data directly from the authenticator.
   * Provides the most information about the authenticator, including its manufacturer and model.
   * Example use case: High-security scenarios where detailed authenticator verification is required.
4. **enterprise**:
   * For enterprise use cases where user identity and authenticator type need strong binding.
   * Enables controlled device registration for enterprise systems.
   * Example use case: Corporate environments with managed devices.

# **What is AuthenticatorSelectionCriteria in WebAuthn?**

AuthenticatorSelectionCriteria is a property used during **WebAuthn registration** to define the criteria for selecting which type of authenticator should be used during the creation of a new credential. This helps the server to guide the user in selecting an authenticator that best fits the required security and usability criteria.

**Purpose of AuthenticatorSelectionCriteria**

The AuthenticatorSelectionCriteria helps ensure that the authenticator used in the registration process meets specific security and usability requirements. For example, you might want to ensure that the authenticator:

* Supports a certain level of security (e.g., requires user verification).
* Is cross-platform (usable on multiple devices).
* Supports a specific kind of user interaction (e.g., biometric vs. PIN).

**Fields in AuthenticatorSelectionCriteria**

The main parameters of AuthenticatorSelectionCriteria are:

1. **AuthenticatorAttachment**:
   * Defines the type of authenticator to use based on how it connects to the device (e.g., USB, NFC, built-in).
   * Possible values:
     + **CROSS\_PLATFORM**: The authenticator can be used across multiple platforms (e.g., USB or NFC security keys).
     + **PLATFORM**: The authenticator is built into the platform (e.g., biometric sensor on a laptop or smartphone).
   * **Example**: If you want the user to register using a USB security key, you can specify AuthenticatorAttachment.CROSS\_PLATFORM.
2. **RequireResidentKey**:
   * Specifies whether the authenticator should support storing credentials in a resident key (stored on the authenticator itself).
   * Possible values:
     + **true**: The authenticator must support storing credentials on the device.
     + **false**: The authenticator does not need to support this feature.
     + **preferred**: The authenticator may or may not support resident keys, but it’s preferred.
   * **Example**: If you want the user’s credentials to be stored on the authenticator, set this to true.
3. **UserVerificationRequirement**:
   * Specifies whether the authenticator must require user verification (e.g., fingerprint, PIN, facial recognition).
   * Possible values:
     + **REQUIRED**: The authenticator must require user verification.
     + **PREFERRED**: The authenticator may require user verification, but it’s not mandatory.
     + **DISCOURAGED**: User verification is not required or should be avoided.
   * **Example**: If you want to enforce the use of biometric authentication (fingerprint or face recognition), you would set this to UserVerificationRequirement.REQUIRED.

**Example of Using AuthenticatorSelectionCriteria**

Here’s an example of how you can set AuthenticatorSelectionCriteria in the **Java WebAuthn4j** implementation during **registration**:

java

Copy code

AuthenticatorSelectionCriteria authenticatorSelectionCriteria = new AuthenticatorSelectionCriteria(

AuthenticatorAttachment.CROSS\_PLATFORM, // Ensure it's cross-platform (e.g., USB key)

true, // Require the authenticator to support resident keys (i.e., store credentials on the device)

UserVerificationRequirement.PREFERRED // Prefer but do not require user verification (e.g., fingerprint)

);

// Create the PublicKeyCredentialCreationOptions with the defined authenticator selection criteria

PublicKeyCredentialCreationOptions creationOptions = new PublicKeyCredentialCreationOptions(

relyingParty,

userEntity,

challenge,

publicKeyCredentialParameters,

timeout,

allowedCredentials,

authenticatorSelectionCriteria,

attestationConveyancePreference,

null // Extensions, if any

);

**Explanation of the Example:**

* **AuthenticatorAttachment.CROSS\_PLATFORM**: Specifies that the authenticator should be cross-platform, such as a USB or NFC key.
* **true**: This enforces that the authenticator supports resident keys, meaning that the credentials can be stored directly on the authenticator, which is useful for offline use.
* **UserVerificationRequirement.PREFERRED**: It’s preferred (but not mandatory) that the authenticator requires user verification (e.g., fingerprint, PIN).

**Impact of AuthenticatorSelectionCriteria on Registration**

When you set these criteria, the authenticator will evaluate whether it meets the required conditions and then proceed with credential creation. If the conditions are not met, the authenticator might not allow the user to register or prompt them to use a different authenticator.

**Common Use Cases for AuthenticatorSelectionCriteria**

1. **Cross-Platform Authenticators**:
   * Ensure that the user can use an authenticator (e.g., USB or NFC key) on any device they own.
   * **Use case**: For multi-device security, where the user can authenticate with the same security key on different computers or browsers.
2. **Platform Authenticators**:
   * Restrict authenticator usage to built-in devices like fingerprints or facial recognition on a smartphone or laptop.
   * **Use case**: For mobile-first applications or environments where users only authenticate using the built-in biometric sensors.
3. **High-Security Environments**:
   * Require **user verification** for every login attempt (e.g., enforcing fingerprint or PIN verification).
   * **Use case**: Banking or financial services where security is critical.

**Summary of AuthenticatorSelectionCriteria Options**

| **Field** | **Possible Values** | **Description** |
| --- | --- | --- |
| **AuthenticatorAttachment** | CROSS\_PLATFORM, PLATFORM | Defines the type of authenticator (USB key vs. built-in device). |
| **RequireResidentKey** | true, false, preferred | Enforces or prefers authenticators that support resident keys. |
| **UserVerificationRequirement** | REQUIRED, PREFERRED, DISCOURAGED | Specifies whether user verification is required. |