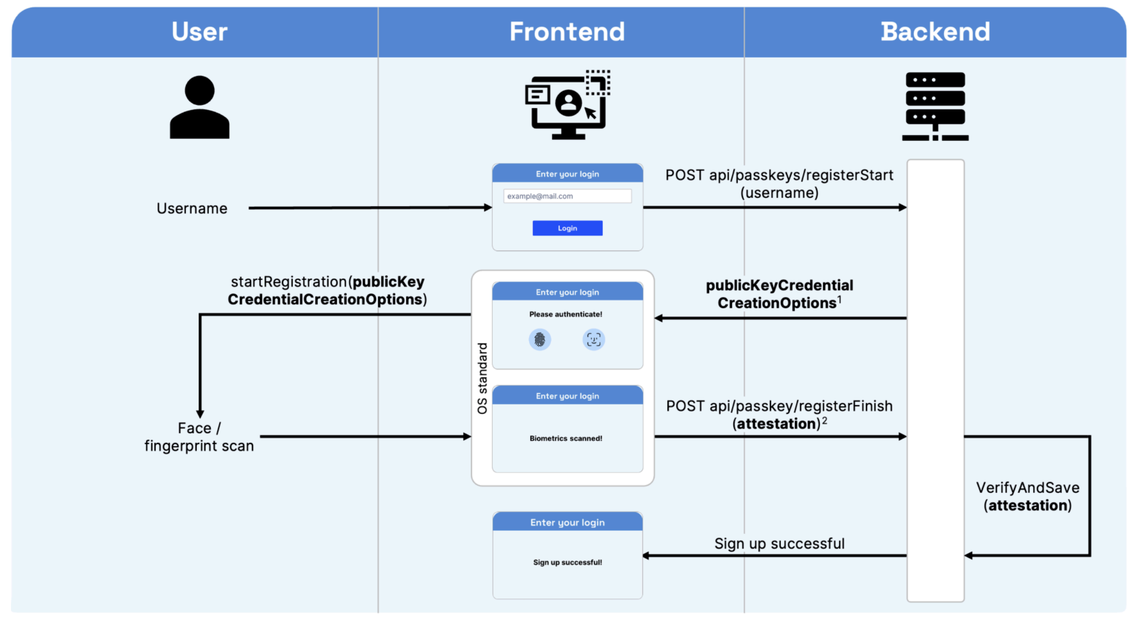
Passkeys Cheat Sheet - Overview

Everything you need to know about WebAuthn and passkeys

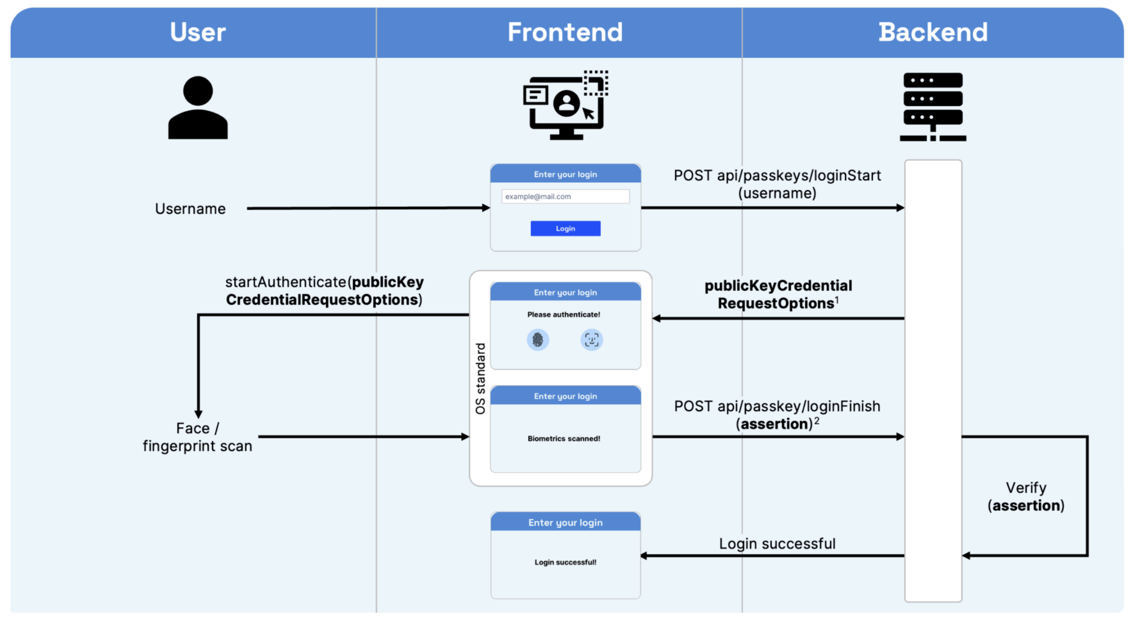
Ceremonies

Authentication with passkeys is based on the two processes, also called ceremonies, **registration** (aka the "attestation phase") and **login** (aka the "assertion phase").
Each phase requires a random challenge generated by the server, which is signed by the authenticator and sent back to the WebAuthn server to verify the user.

Registration (attestation)


User
Frontend
Backend
Username
Enter your login
example@mail.com
Login
POST api/passkeys/registerStart (username)
startRegistration(**publicKey CredentialCreationOptions**)
publicKeyCredential CreationOptions[1]
OS standard
Please authenticate!
Face / fingerprint scan
Biometrics scanned!
POST api/passkey/registerFinish (**attestation**)[2]
VerifyAndSave (**attestation**)
Sign up successful
Sign up successful!

Login (assertion)



1) The cryptographic challenge is created by the WebAuthn server and passed to the client via the Options
2) The challenge is signed by the authenticator and returned to the server via the attestation / assertion objects

PublicKeyCredentialCreationOptions

```
"PublicKeyCredentialCreationOptions": {
  "rp": {
    "id": "passkeys.eu",
    "name": "Corbado Passkeys Demo"
  },
  "user": {
    "displayName": "john.doe",
    "id": "dXNyLZ...DU1OTc",
    "name": "john@doe.com"
  },
  "challenge": "888fix4Bus...pHHr3Y",
  "pubKeyCredParams": [
    {
      "alg": -7,
      "type": "public-key"
    },
    {
      "alg": -257,
      "type": "public-key"
    }
  ],
  "excludeCredentials": [],
  "authenticatorSelection": {
    "authenticatorAttachment": "platform",
    "residentKey": "required",
    "userVerification": "required"
  },
  "attestation": "none",
  "extensions": []
```

PublicKeyCredentialCreationOptions is the central object of the **attestation phase (Registration).**
It is created by and returned from the WebAuthn server, containing these attributes:

- **rp:** Identifies the Relying Party (= the server looking to authenticate the user), usually the ID is the server domain.
- **user:** Contains data about the user account requesting attestation. The ID is a byte sequence chosen by the Relying Party, that must not contain personal information. The username or e-mail address is saved instead in the name or displayName attribute.
- **challenge:** A randomly generated base64URL encoded string that needs to be signed by the authenticator.
- **pubKeyCredParams:** Specified attributes of the credential to be created, usually the supported algorithm(s).
- **timeout:** Optional time in milliseconds for the client to wait for the call to complete.
- **excludeCredentials:** Optional list of credentials to limit the creation of multiple passkeys on one device.
- **authenticatorSelection:** Optional selection of the used authenticator for the method, e.g. whether a residentKey is required. See the last page for more information.
- **attestation:** Can be used to request that the attestation object is passed on to the Relying Party in a specific form. Possible values are "none" (default), "indirect", "direct" and "enterprise".
- **extensions:** Optional request(s) for additional processing, such as specific return values. e.g.
  - *credProbs* requests information on whether the created credential is discoverable
  - *minPinLength* allows to request the authenticator's minimum PIN length
  - *prf* allows the Relying Party to use outputs from a pseudo-random function (PRF) associated with a credential

PublicKeyCredentialRequestOptions

```
"publicKeyCredentialRequestOptions": {
  "challenge": "pT7HMA-...dFPHk",
  "timeout": 500,
  "rpId": "passkeys.eu",
  "userVerification": "preferred",
  "allowCredentials": [],
  "extensions": []
 }
}
```

PublicKeyCredentialRequestOptions is the central object of the **assertion phase (Login).**
It is created by and returned from the server, containing these attributes:

- **challenge, timeout, extensions:** see above
- **rpId:** The identifier of the Relying Party for the assertion request, usually its domain.
- **allowCredentials:** Optional list of credentials that are allowed for authentication, indicating the caller's preference by descending order. This list would be filled with PublicKeyCredentialDescriptors.
- **userVerification:** Optional value to specify requirements for user verification during the operation. Possible values are "preferred" (default), "required" or "discouraged".

Attestation

During the Registration Ceremony, the Authenticator returns this **Registration Response**. (You can try this yourself in WebAuthn Playground)

```
{
  "root": {
    "id": "QFPlQVypLmmx71eOtmS3IfCFky0",
    "rawId": "QFPlQVypLmmx71eOtmS3IfCFky0",
    "response": {
      "attestationObject": {
        "fmt": "none",
        "attStmt": {},
        "authData": {
          "rpIdHash": "t8DGRTBfls-BhOH2QC404lvdhe_t2_NkvM0nQWEEADc",
          "flags": {
            "userPresent": true,
            "userVerified": true,
            "backupEligible": true,
            "backupStatus": true,
            "attestedData": true,
            "extensionData": false
          },
          "counter": "0000",
          "aaguid": "00000000-0000-0000-0000-000000000000",
          "credentialID": "QFPlQVypLmmx71eOtmS3IfCFky0",
          "credentialPublicKey": "pQECAyYgASFYIEa-lpSiQ4P...",
          "parsedCredentialPublicKey": {
            "keyType": "EC2 (2)",
            "algorithm": "ES256 (-7)",
            "curve": 1,
            "x": "Rr6WlKJDg8MlbIq9mmHQzk2p2c_s7QoNKr7yMa7I8pM",
            "y": "tAELYp7h3sYNjZZIZgHPYiaSzFxQVTl8cgZ_7wm13Vw"
          }
        }
      },
      "clientDataJSON": {
        "type": "webauthn.create",
        "challenge": "AAABeB78HrIemh1jTdJICr_3QG_RMOhp",
        "origin": "https://opotonniee.github.io",
        "crossOrigin": false
      },
      "transports": [
        "hybrid",
        "internal"
      ],
      "publicKeyAlgorithm": -7,
      "publicKey": "MFkwEwYHKoZIzj0CAQYIKoZIzj0DA...",
      "authenticatorData": "t8DGRTBfls-BhOH2QC404lvdhe_..."
    },
    "type": "public-key",
    "clientExtensionResults": {},
    "authenticatorAttachment": "cross-platform"
  }
}
```

The **attestationObject** is a CBOR encoded object, containing information about the newly created credentials, the public key and other relevant data

- **fmt** is typically evaluated to "none" for passkeys
- **attStmt** is empty for passkeys and filled for other authenticators, e.g. hardware security keys
- **authData** is a buffer of values containing the following data:

Name	Length (bytes)	Description
RP ID hash	32	This is the SHA-256 hash of the origin, e.g. *passkeys.eu*
FLAGS	1	Determining multiple information, e.g. whether the User is present.
COUNTER	4	For passkeys in enclaves this is usually 0, while it's the actual sign counter for the security keys.
ATTESTED CREDENTIAL DATA	variable	Will contain credential data if it's available in a COSE key format.
EXTENSIONS	variable	These are any optional extensions for authentication, read more here

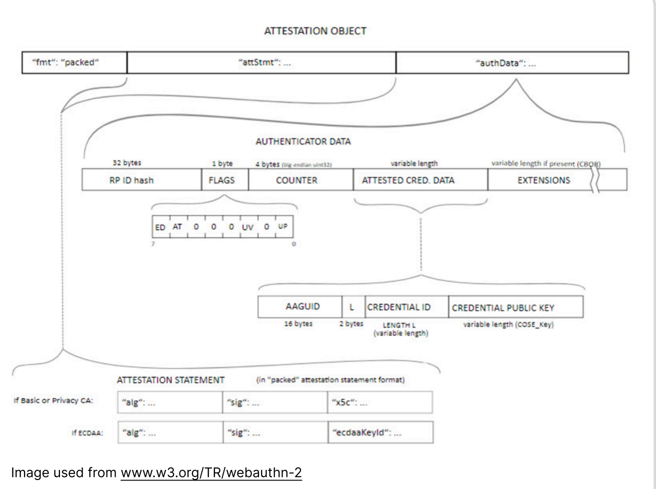


Image used from www.w3.org/TR/webauthn-2

Passkeys are generated with **COSE Algorithms,** indicating the used algorithm in the *algorithm*-attribute of the parsedCredentialPublicKey:

Value	**-36**	**-35**	**-8**	**-7**	**-259**	**-258**	**-257**	**-39**	**-38**	**-37**
Name	ES512	ES384	Ed25519	ES256	RS512	RS384	RS256	PS512	PS384	PS256
Description	ECDSA w/SHA-512	ECDSA w/SHA-384	Ed25519 w/ EdDSA	ECDSA w/SHA-256	RSASSA-PKCS1-$_{v1\_5}$ using SHA-512	RSASSA-PKCS1-$_{v1\_5}$ using SHA-384	RSASSA-PKCS1-$_{v1\_5}$ using SHA-256	RSASSA-PSS w/ SHA-512	RSASSA-PSS w/ SHA-384	RSASSA-PSS w/ SHA-256
Typically used by				🍎 & 🤖			🪟			

The ***transport* - property** indicates which mode of cross-device authentication is possible with the credential.
Possible combinations are:

1. "transports": ["internal", "hybrid"]: Passkeys can be used from the platform authenticator (e.g. Face ID, Touch ID, Windows Hello) or via cross-device authentication
2. "transports": ["internal"]: Passkeys can be used from the platform authenticator (e.g. Face ID, Touch ID, Windows Hello)
3. No "transports" property set: default behavior which gives no restrictions

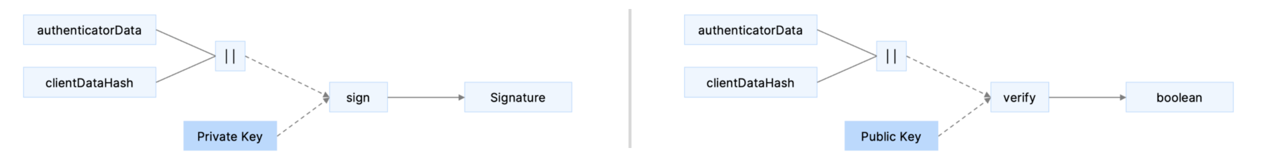
Assertion

During the Login Ceremony, the Authenticator returns this Login Response. (You can try this yourself in the WebAuthn Playground)

```
"root": {
  "id": "QFPlQVypLmmx71eOtmS3IfCFky0",
  "rawId": "QFPlQVypLmmx71eOtmS3IfCFky0",
  "type": "public-key",
  "response": {
    "authenticatorData": {
      "rpIdHash": "t8DGRTBfls-BhOH2QC404...",
      "flags": {
        "userPresent": true,
        "userVerified": true,
        "backupEligible": true,
        "backupStatus": true,
        "attestedData": false,
        "extensionData": false
      },
      "counter": 0
    },
    "clientDataJSON": {
      "type": "webauthn.get",
      "challenge": "EGYtAMgi8B2Ey1FNVfVF93....",
      "origin": "https://opotonniee.github.io",
      "crossOrigin": false
    },
    "signature": "MEQCICx9J-G4mmL3g0TFK3uVxQN5...",
    "userHandle": "YWxleCBtdWxsZXI"
  },
  "authenticatorAttachment": "platform"
}
```

Flag	Meaning, if set to 'true'	Notes
userPresent (UP)	User presence was tested by the authenticator.	⚠️ Only if **BOTH flags are set as true**, the attestation is a **2-Factor-Authentication**. If only UP is set to true, the login is considered a Single-Factor-Authentication.
userVerified (UV)	The user was verified by the authenticator, e.g. with a fingerprint or PIN.	
backupEligible (BE)	The public key credential source can be backed upon another authenticator. This is set on creation and can't be changed later on.	Possible combinations and their meanings are: • **BE=0; BS = 0:** The credential is a single-device credential • **BE=1; BS = 0:** The credential is a multi-device credential and currently not backed up • **BE=1; BS = 1:** The credential is a multi-device credential and currently backed up
backupStatus (BS)	The public key credential source is currently backed up. This can change later on and is only allowed, if BE is set to true.	

The **signature** is used to verify that the user trying to log in, actually has the private key.
It's the result of the concatenation of the authenticatorData and clientDataHash (i.e. the SHA-256 version of ClientDataJSON). A similar process is used to verify the user with the public key.



The **userHandle** is the actual user_id. Read more about the user_id in the "Database Schema" on the next page.

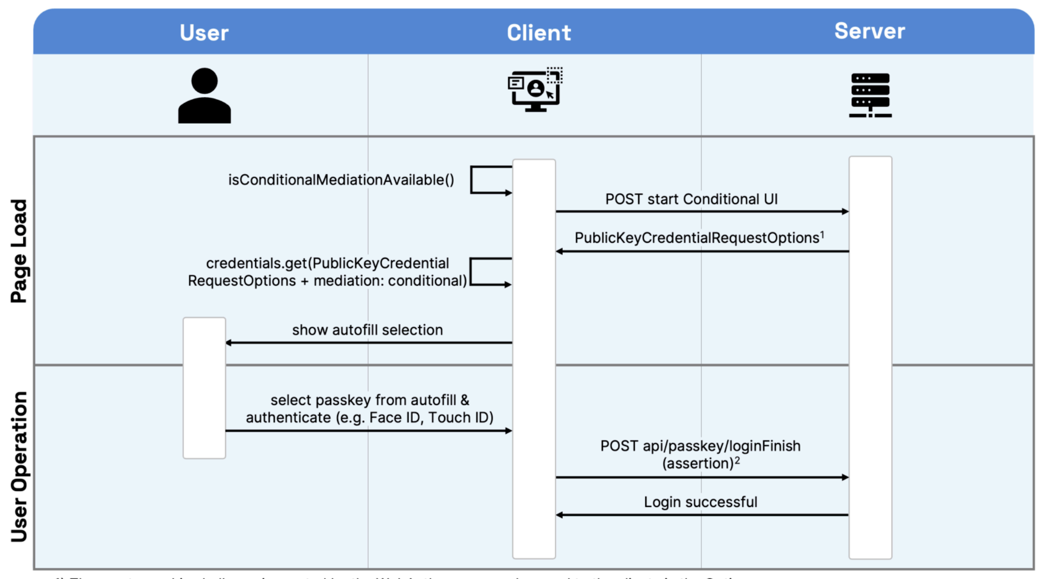
Passkeys Cheat Sheet - Deep Dive

Everything you need to know about WebAuthn and passkeys

Conditional UI - Overview

Conditional UI ("passkey autofill") displays available passkeys in a selection dropdown for the user, when a user has a resident key registered with the relying party. It improves the usability of passkeys, but requires additional development efforts and is not available for all OS / browser combinations.

Login with Conditional UI


User
Client
Server
Page Load
User Operation
isConditionalMediationAvailable()
POST start Conditional UI
PublicKeyCredentialRequestOptions¹
credentials.get(PublicKeyCredential RequestOptions + mediation: conditional)
show autofill selection
select passkey from autofill & authenticate (e.g. Face ID, Touch ID)
POST api/passkey/loginFinish (assertion)²
Login successful

1) The cryptographic challenge is created by the WebAuthn server and passed to the client via the Options
2) The challenge is signed by the authenticator and returned to the server via the attestation / assertion objects

Device Compatibility

Browser \ OS	Android	iOS	macOS	Windows	chromeOS	ubuntu
Chrome	✓	✓	✓	✓	✗	✗
Safari	✗	✓	✓	✗	✗	✗
Firefox	✗	✓	✓	✓	✗	✗
Edge	✗	✓	✓	✗	✗	✗

Database Schema

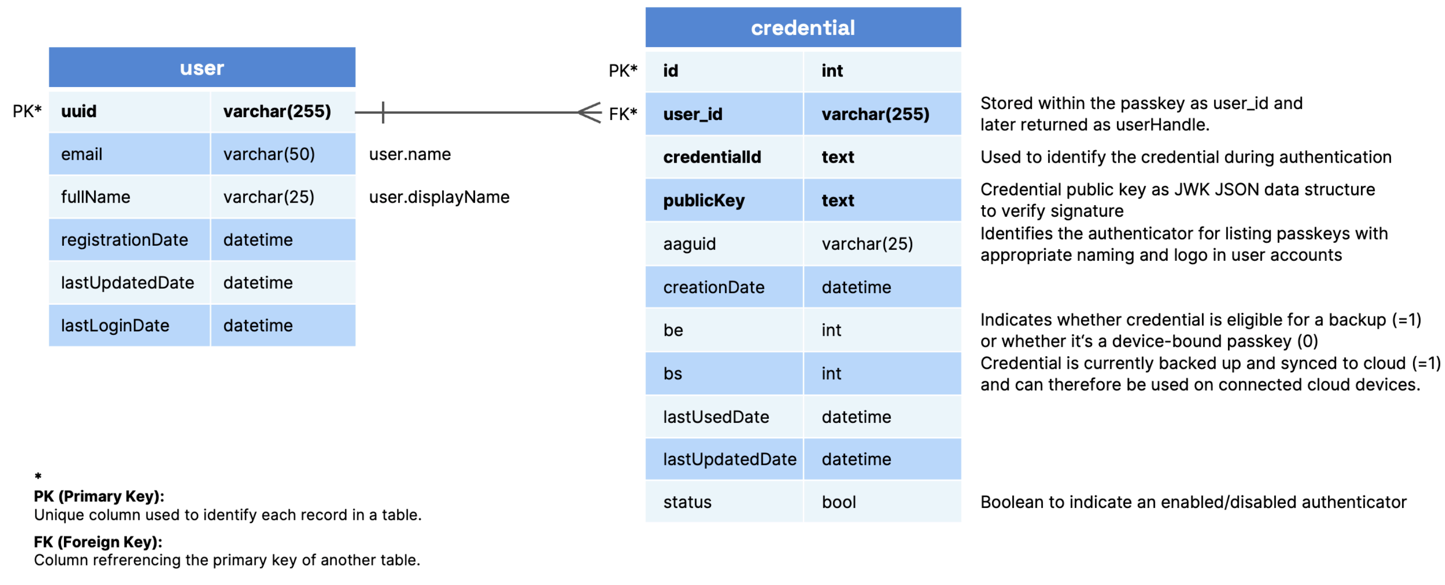
There is no mandatory or standardized database schema for WebAuthn servers. However, this example database schema can be used to store the required information and provide all functionalities of a WebAuthn server. Bold attributes are mandatory for a minimal viable implementation, while the others are only needed for optional, but helpful features.

Authentication-relevant data:

- **Credential ID:**
  This is a unique ID that's generated by the authenticator during registration of a passkey. It should be used to look up the actual user account that's associated with the passkey.
  Additionally the userHandle (from user_id) should then be compared to validate the account used for authentication.
  Never use the user.name attribute as it can change over time.
- **User ID (user_id)**
  Unique ID specified by the Relying Party to represent a user account in their system. It's returned as the userHandle within the assertion-object. Authenticators store only one Credential per user_id per Relying Party.

Metadata for display and selection of passkeys:

- **User DisplayName (user.displayName)**
  User-friendly, readable name that is typically the full name of the user. It's shown to the user, but not used during authentication.
- **User Name (user.name)**
  Unique and readable name that is typically an e-mail address or a username. It can be shown to the user, but it's not used during authentication.



user

PK*	**uuid**	**varchar(255)**	
	email	varchar(50)	user.name
	fullName	varchar(25)	user.displayName
	registrationDate	datetime	
	lastUpdatedDate	datetime	
	lastLoginDate	datetime	

credential

PK*	**id**	**int**	
FK*	**user_id**	**varchar(255)**	Stored within the passkey as user_id and later returned as userHandle.
	credentialId	**text**	Used to identify the credential during authentication
	publicKey	**text**	Credential public key as JWK JSON data structure to verify signature
	aaguid	varchar(25)	Identifies the authenticator for listing passkeys with appropriate naming and logo in user accounts
	creationDate	datetime	
	be	int	Indicates whether credential is eligible for a backup (=1) or whether it's a device-bound passkey (0)
	bs	int	Credential is currently backed up and synced to cloud (=1) and can therefore be used on connected cloud devices.
	lastUsedDate	datetime	
	lastUpdatedDate	datetime	
	status	bool	Boolean to indicate an enabled/disabled authenticator

*
PK (Primary Key):
Unique column used to identify each record in a table.
FK (Foreign Key):
Column refrerencing the primary key of another table.

Conditional UI - Implementation

Code Example

A full, minimalistic code for a Conditional UI method looks like this:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Conditional UI</title>
</head>
<body>

<input type="text" id="username" autoComplete="username webauthn" />

<script>
    async function passkeyLogin() {
        try {
            // retrieve the request options (incl. the challenge) from the WebAuthn server
            let options = await WebAuthnClient.getPublicKeyRequestOptions();

            const credential = await navigator.credentials.get({
                publicKey: options.publicKeyCredentialRequestOptions,
                mediation: "conditional",
            });

            const userData = await WebAuthnClient.sendSignedChallenge(credential);
            window.location.href='/logged-in';
        } catch (error) {
            console.log(error);
        }
    };

    passkeyLogin();
</script>

</body>
</html>
```

Technical Requirements

Conditional UI only works with **resident keys** / discoverable credentials
It's recommended to provide a **different server endpoint** to start the Conditional UI login.
The client needs to meet multiple requirements:
- The browser needs to support Conditional UI (see a detailed list here)
- JavaScript must be enabled and the web page must provide an HTML input field
- Timeout parameters should be disregarded

To avoid errors, the server should first test the client's availability with this function:

```javascript
// Availability of `window.PublicKeyCredential` means WebAuthn is usable.
if (
  window.PublicKeyCredential &&
  PublicKeyCredential.isConditionalMediationAvailable
) {
  // Check if conditional mediation is available.
  const isCMA = await PublicKeyCredential.isConditionalMediationAvailable();
  if (isCMA) {
    // Call WebAuthn authentication start endpoint

    let options = await WebAuthnClient.getPublicKeyRequestOptions();

    const credential = await navigator.credentials.get({
      publicKey: options.publicKeyCredentialRequestOptions,
      mediation: "conditional",
    });
    /*
    ...
    */
  }
}
```

Autocomplete Token in Input Fields

The input field should receive an HTML autofill token, that signals the client to populate passkeys to the ongoing request. Besides passkeys, the autofill tokens can be paired with existing tokens, e.g. usernames and passwords:

```html
<label for="name">Username:</label>
<input type="text" name="name" autocomplete="username webauthn">
<label for="password">Password:</label>
<input type="password" name="password" autocomplete="current-password webauthn">
```

authenticatorSelection

The **authenticatorSelection** - object allows the server to dictate settings for the authenticator and credential creation.

```json
{
  "rp": {
    "name": "corbado.com",
    "id": "corbado.com"
  },
  "user": {
    "id": "dGVzdefEyMjE",
    "name": "test-username",
    "displayName": "test-username"
  },
  "challenge": "mhanjsapJjCNaN_Ttasdfk1C0DymR-V_w_0
  "pubKeyCredParams": [
    {
      "type": "public-key",
      "alg": -7
    },
    {
      "type": "public-key",
      "alg": -257
    }
  ],
  "timeout": 60000,
  "excludeCredentials": [],
  "authenticatorSelection": {
    "authenticatorAttachment": "platform",
    "residentKey": "preferred",
    "requireResidentKey": false,
    "userVerification": "preferred"
  },
  "attestation": "none",
  "extensions": {
    "credProps": true
  }
}
```

Possible Values

authenticatorAttachment
- **Platform**: The authenticator is attached to the client's platform and is therefore not removable.
- **Cross-platform:** The authenticator is not bound to the client's platform and can be used on multiple devices.

residentKey
- **Required:** The authenticator must create a resident key (if not possible the operation should fail).
- **Preferred:** The authenticator should try to create a resident key (if not possible it should create a non-resident key)
- **Discouraged:** The authenticator must create a non-resident key (if not possible the operation should fail).

userVerification
- **Preferred:** The operation should have user verification, but can proceed without it. (standard option)
- **Required:** The operation must have user verification.
- **Discouraged:** The operation should not have user verification.

⚠ **Warning:**
If set to "Preferred" the user or his device can skip the user verification in the authentication process. Only choose this value if user-friendliness is more important than security. (Read more in this article)

Resident Keys vs. Non-Resident Keys

There are two types of passkeys that differ in their storage and retrieval mechanisms:

- **Resident Keys (also called Discoverable Credential)**
  Resident keys are stored on the authenticator and retrieved during authentication. This way the client can "discover" a list of possible keys, which is why Conditional UI requires resident keys.
- **Non-Resident Key (also called Non-Discoverable Credential)**
  In case of non-resident keys, the public key and the credential ID are stored on the server and not on the authenticator. During each authentication, the authenticator derives the private key from a seed within the credential ID and an internal master key that is saved on the authenticator.

Relying Party ID

The **Relying Party ID** is a domain stored within the passkey, ensuring the passkey only works for the correct browser URLs. During authentication, the rpID is checked against the browser URL and only allowed in these two cases:

1. The URL matches precisely the rpID OR
2. The URL is a subdomain that matches the rpID and the parent domain is not on the Public Suffix List

e.g.

Relying Party ID	originalHost (= Browser URL)	Allowed ?	
"0.0.0.0"	0.0.0.0	✅	
"0x10203"	0.1.2.3	✅	
"[0::1]"	::1	✅	
"example.com"	example.com	✅	
"example.com"	example.com.	❌	*(trailing dot)*
"example.com."	example.com	❌	*(trailing dot)*
"example.com"	www.example.com	✅	
"com"	example.com	❌	*(public suffix)*
"example"	example	✅	
"compute.amazonaws.com"	example.compute.amazonaws.com	❌	*(public suffix)*
"example.compute.amazonaws.com"	www.example.compute.amazonaws.com	❌	*(public suffix)*
"amazonaws.com"	www.example.compute.amazonaws.com	❌	*(public suffix)*
"amazonaws.com"	test.amazonaws.com	✅	

Helpful Tools

These are helpful tools that you can access by clicking on their title.

WebAuthn Debugger:	Tool for Debugging the WebAuthn Response as JSON.
WebAuthn Playground:	Tool for testing WebAuthn operations with different options.
Passkey Glossary:	Explanation of passkey-related terms & concepts.
Chrome Device Log:	Log of your WebAuthn operations. (only on Chrome: chrome://device-log)

Corbado