

# Webアプリケーション特有の仕組み

---

## TIS株式会社

テクノロジー&イノベーション本部

テクノロジー&エンジニアリングセンター



- NablarchはTIS株式会社の登録商標です。
- OracleおよびJava、JavaBeans、JavaScriptは、オラクルおよびその関連会社の登録商標です。
- Google ChromeはGoogle LLC の商標です。  
なお、本文中ではChromeと表記しています。
- Microsoft Edgeはマイクロソフトの製品です。  
なお、本文中ではEdgeと表記しています。
- FirefoxはMozilla Foundationの米国およびその他の国における商標または登録商標です。

その他の社名、製品名などは、一般にそれぞれの会社の商標または登録商標です。  
なお、本文中では、TMマーク、Rマークは明記しておりません。

- 本コンテンツでは、Webアプリケーションの画面例としてnablarch-example-webを使用しています。
  - nablarch-example-web  
<https://github.com/nablarch/nablarch-example-web>

- はじめに
- 認証・認可
- 入力値のバリデーション
- 二重サブミット対策
- エラーハンドリング
- 戻る遷移
- ファイルアップロード・ダウンロード

# はじめに

---

# 本コンテンツで学ぶこと



ログイン状態は  
どのように保たれる？

ヒストリバックできるのに  
なぜ戻るボタンがある？

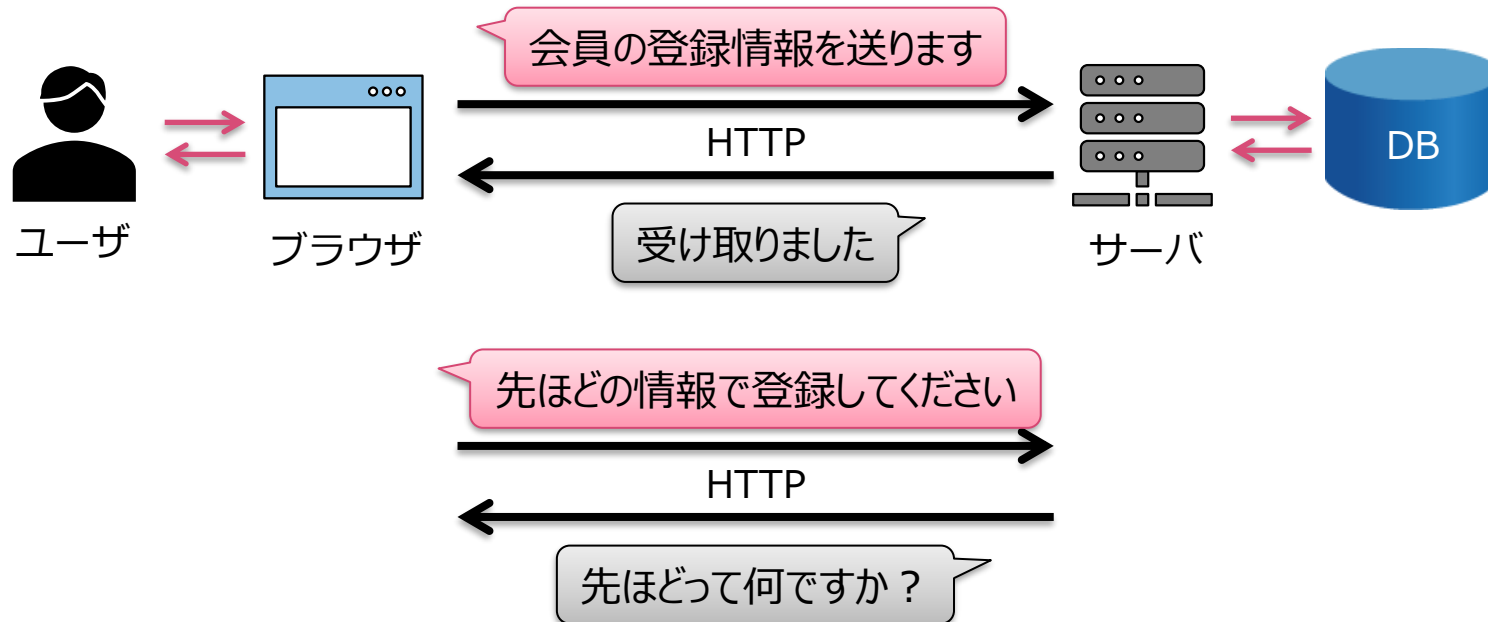
クライアント側だけではなく  
サーバ側でもバリデーション  
をするのはなぜ？

何気なく使用しているWebアプリケーションの仕組みは  
「なぜ」そうになっているのか、意外と知らないものです。  
多くのシステムで採用される「定石」を理解しましょう！

# 【前提】HTTPはステートレス

Webアプリケーションで使用する「HTTP」はステートレスなプロトコル。  
前のリクエストで送信した情報を、後のリクエストで参照することはできない（覚えていない）。

→ 参照したければ、何らかの工夫が必要。



# 認証・認可

---

認証・認可について、以下の4項目を説明します。

- 認証とは
- Form認証に関するトピック
- シングルサインオン
- 認可とは

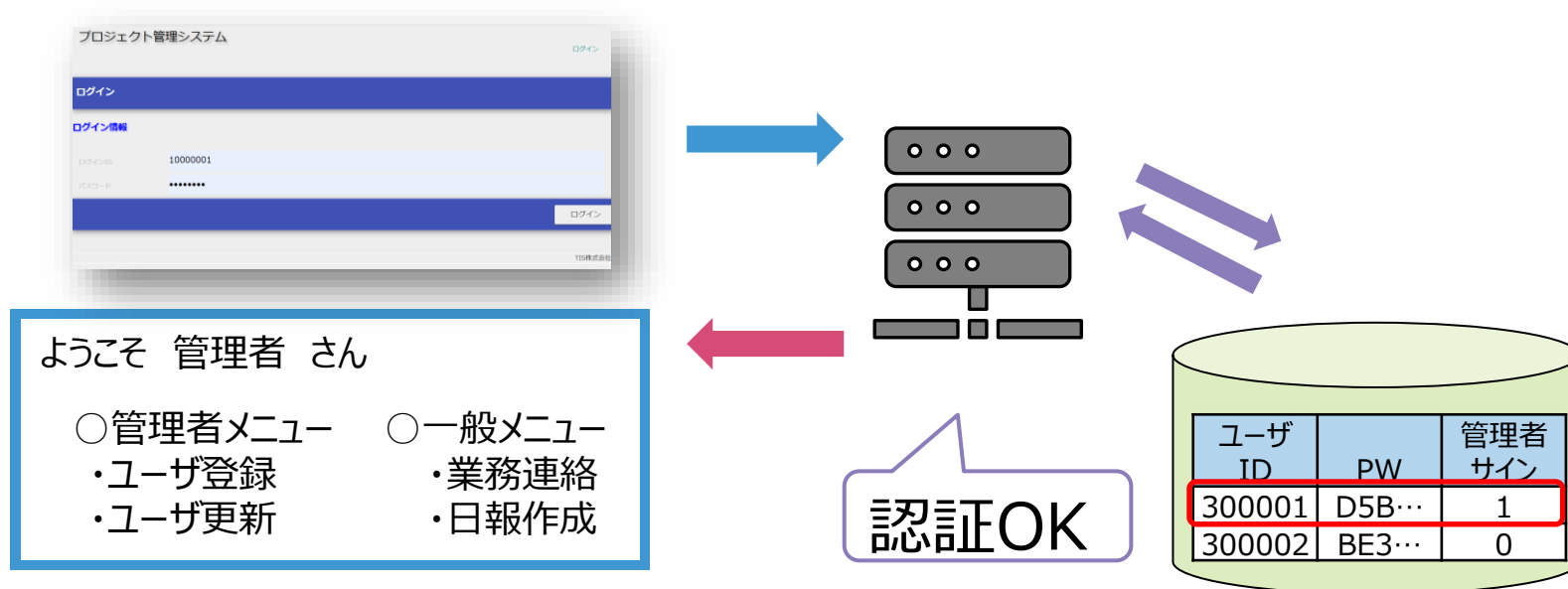


- 認証とは
- Form認証に関するトピック
- シングルサインオン
- 認可とは

## 認証【Authentication】

ネットワークやサーバへ接続する際に本人性をチェックし  
正規の利用者であることを確認すること。

一般には利用者IDとパスワードの組み合わせによりチェックする。

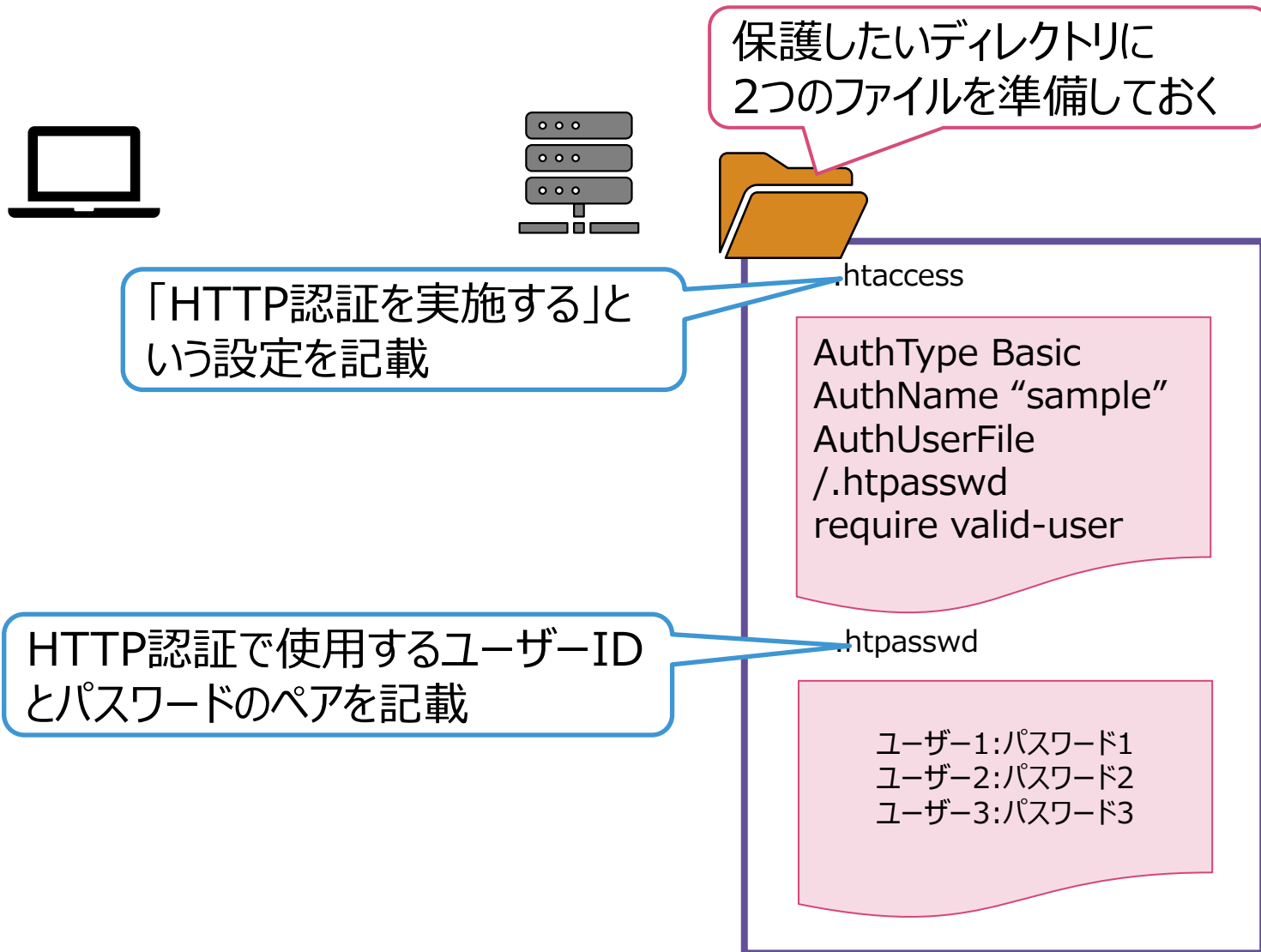


この認証を実現する「仕組み」として代表的な以下の2つの方法について説明します。

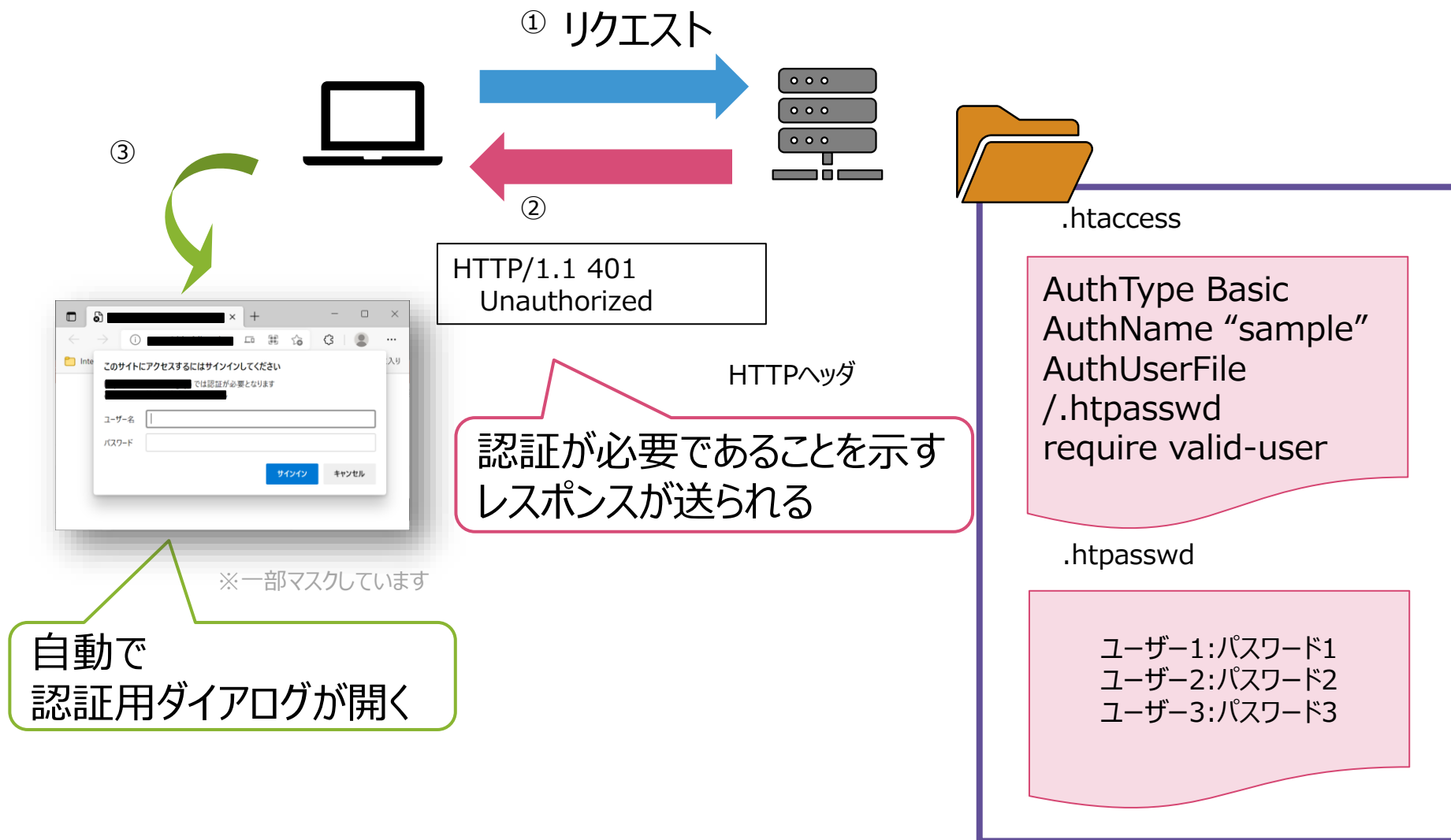
- HTTP認証（Basic認証とも呼ばれる）
- Form認証

- HTTP認証 (Basic認証)
- Form認証

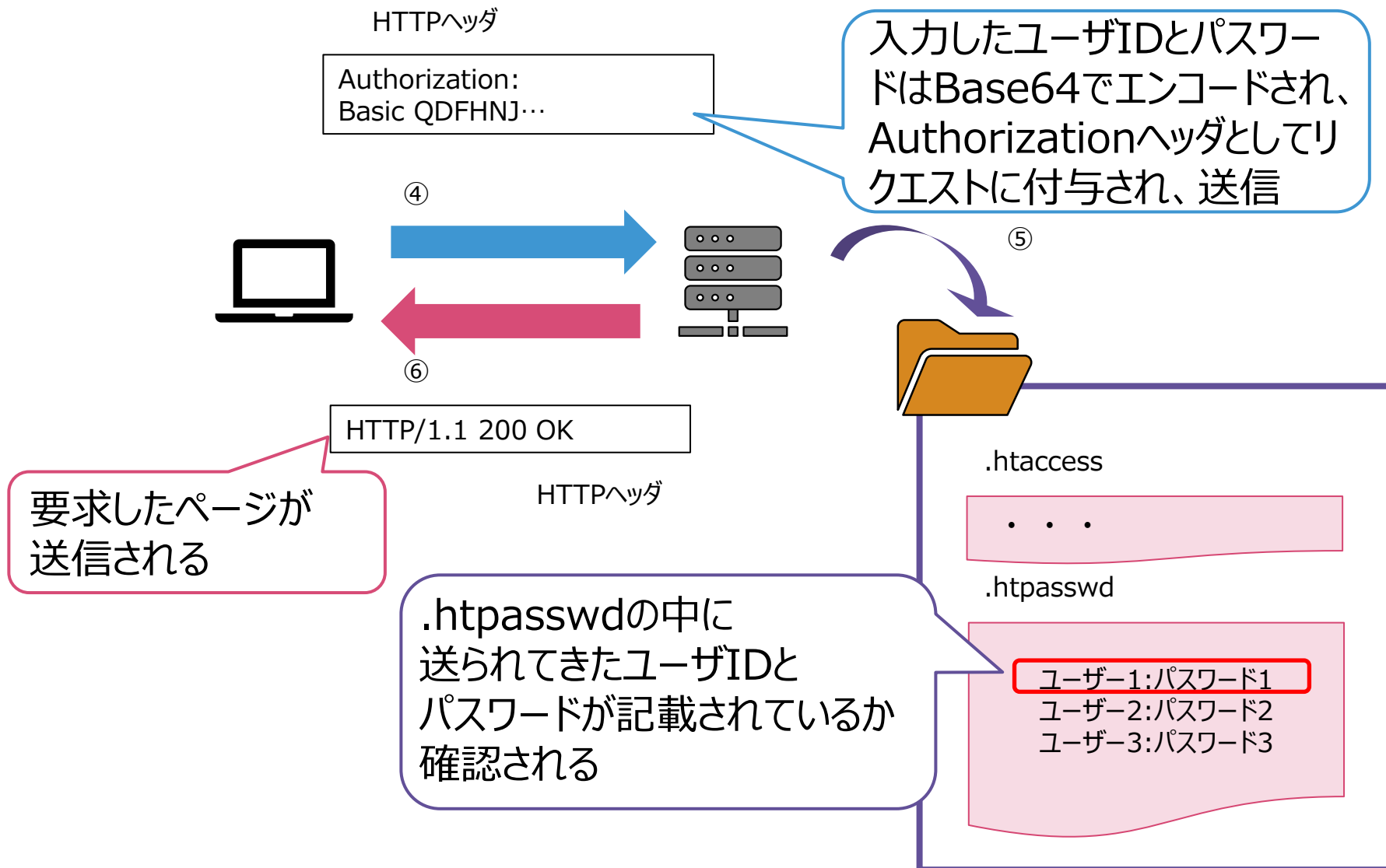
# HTTP認証の事前準備



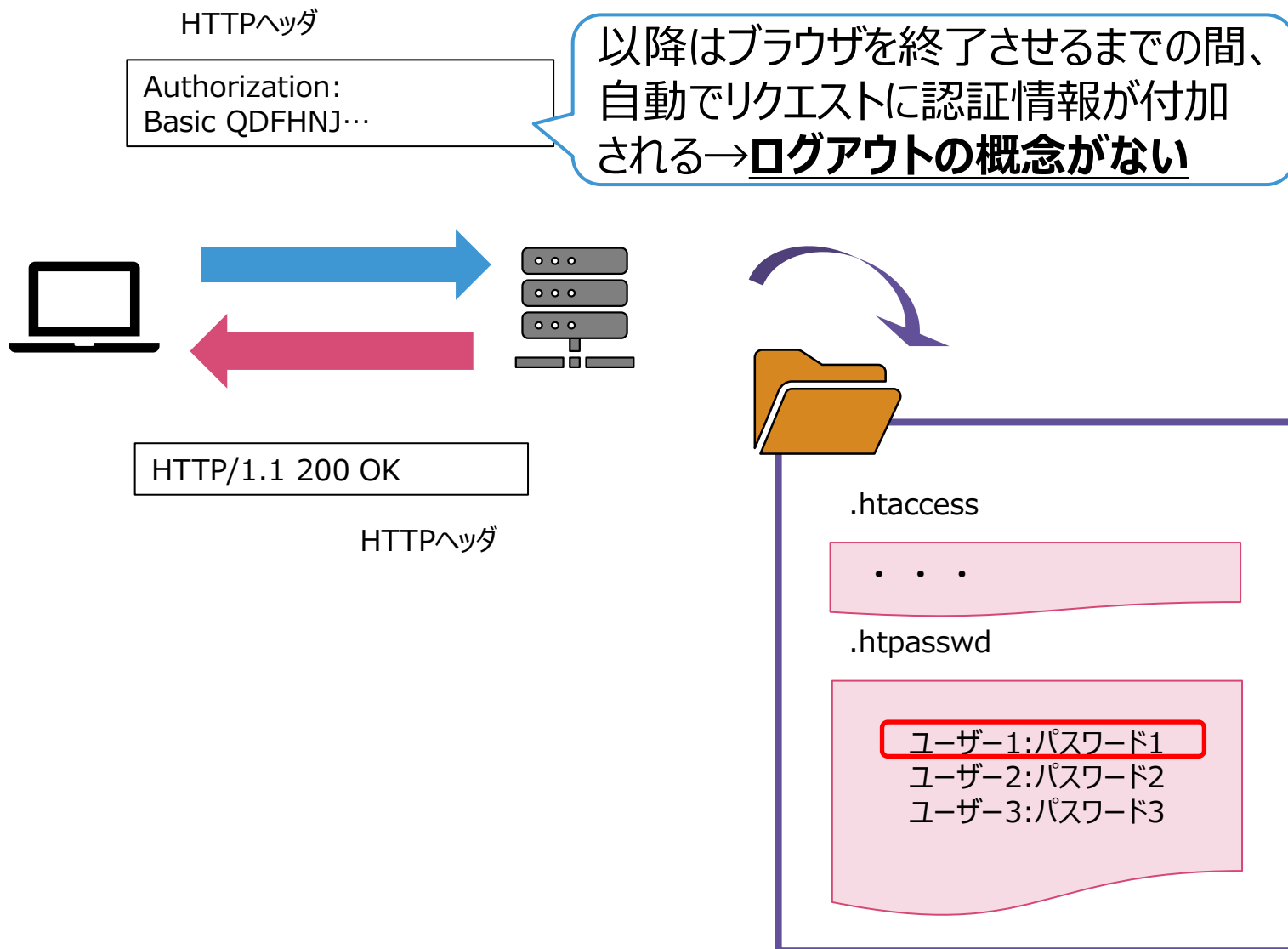
# HTTP認証の流れ (1/3)



# HTTP認証の流れ (2/3)



# HTTP認証の流れ (3/3)





一般公開されておらず、利用者が限定されている内部的なWebアプリケーションで利用される認証方法。

## メリット

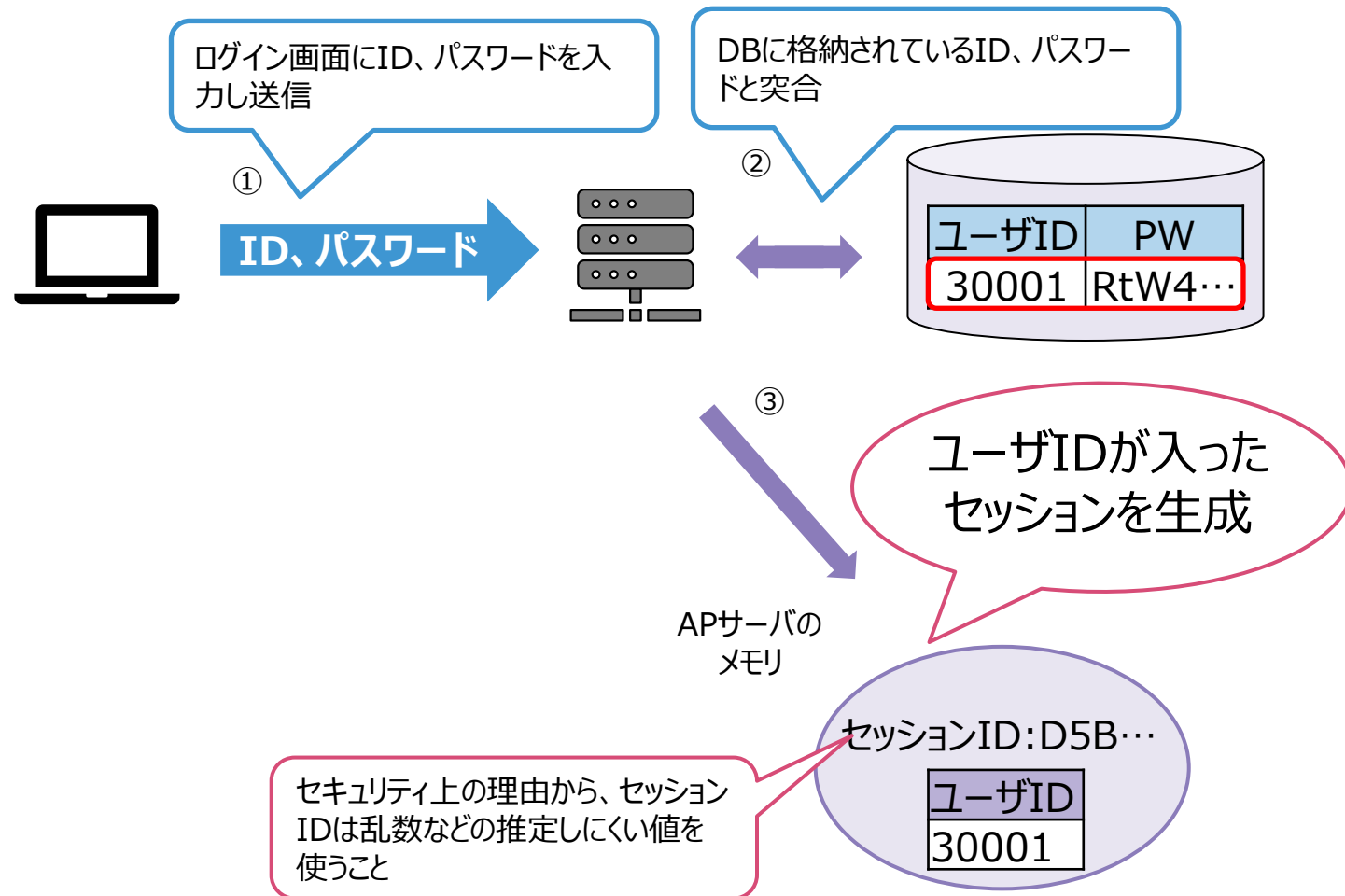
HTTPサーバの設定だけで簡単に実現できる。

## デメリット

- 通信の度にユーザIDとパスワードを送信するため、SSL化しないと情報漏洩の可能性が高い。
- ファイルやディレクトリ単位でしか制御できないため、規模の大きなシステムでの実用に耐えない。
- ユーザ名とパスワードをファイルで管理するため、ユーザの追加や削除、パスワード変更の手間がかかる。

- HTTP認証（Basic認証）
- Form認証

# Form認証の流れ ログイン時 (1/2)

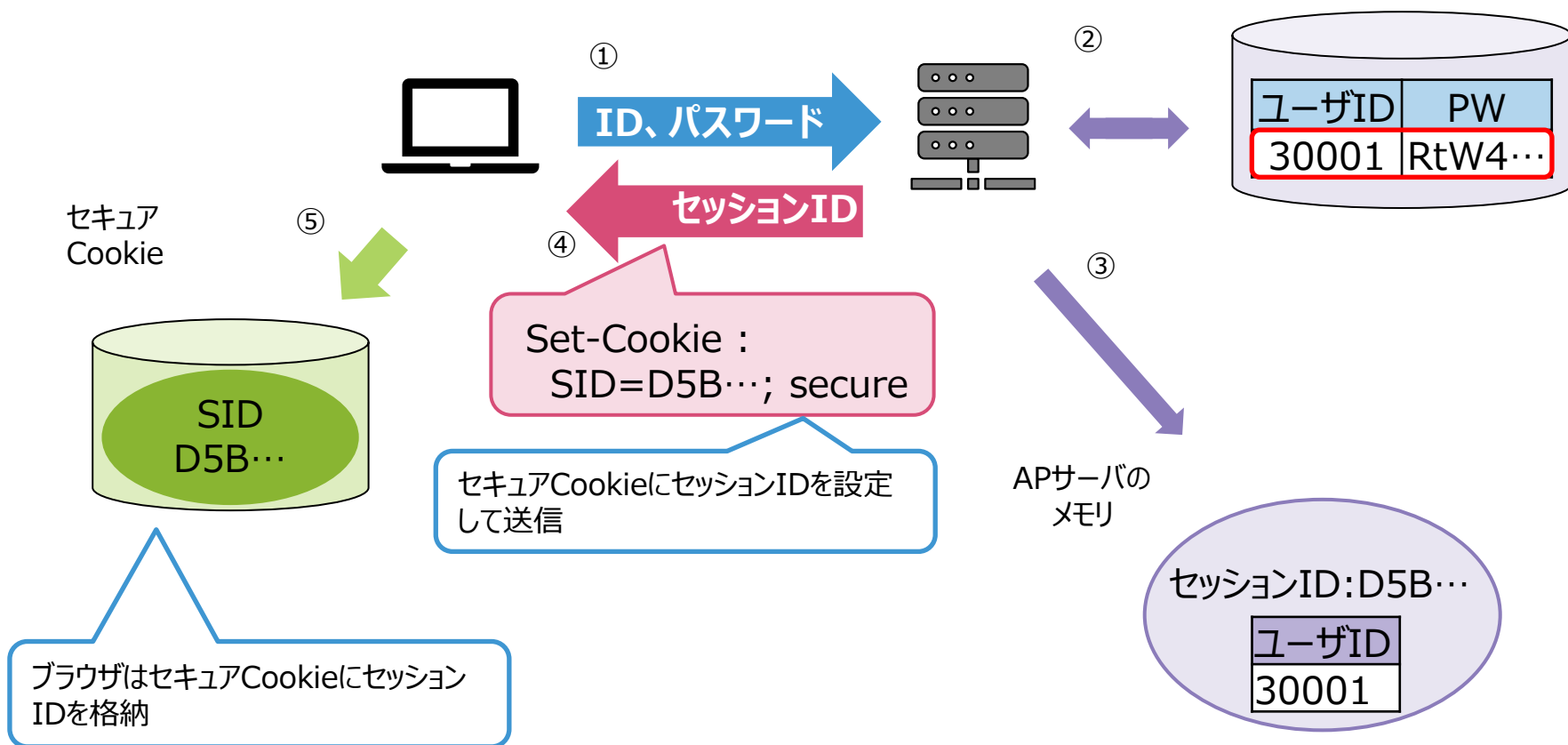


※セッションについて詳しく知りたい方は以下を参照ください。

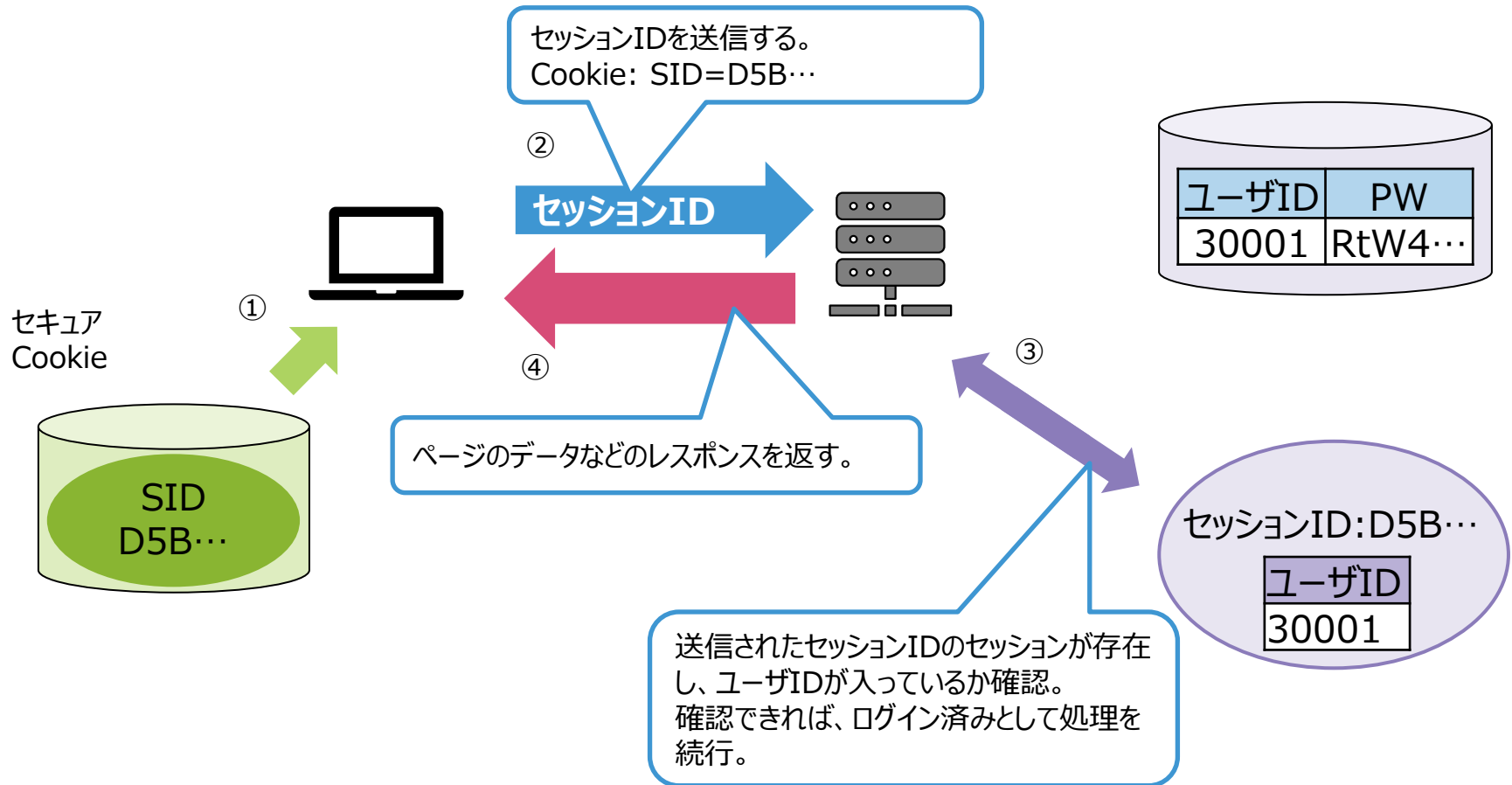
Fintan > アプリケーションアーキテクチャの学習コンテンツ > Webアプリケーションのセッション管理

<https://fintan.jp/?p=8376>

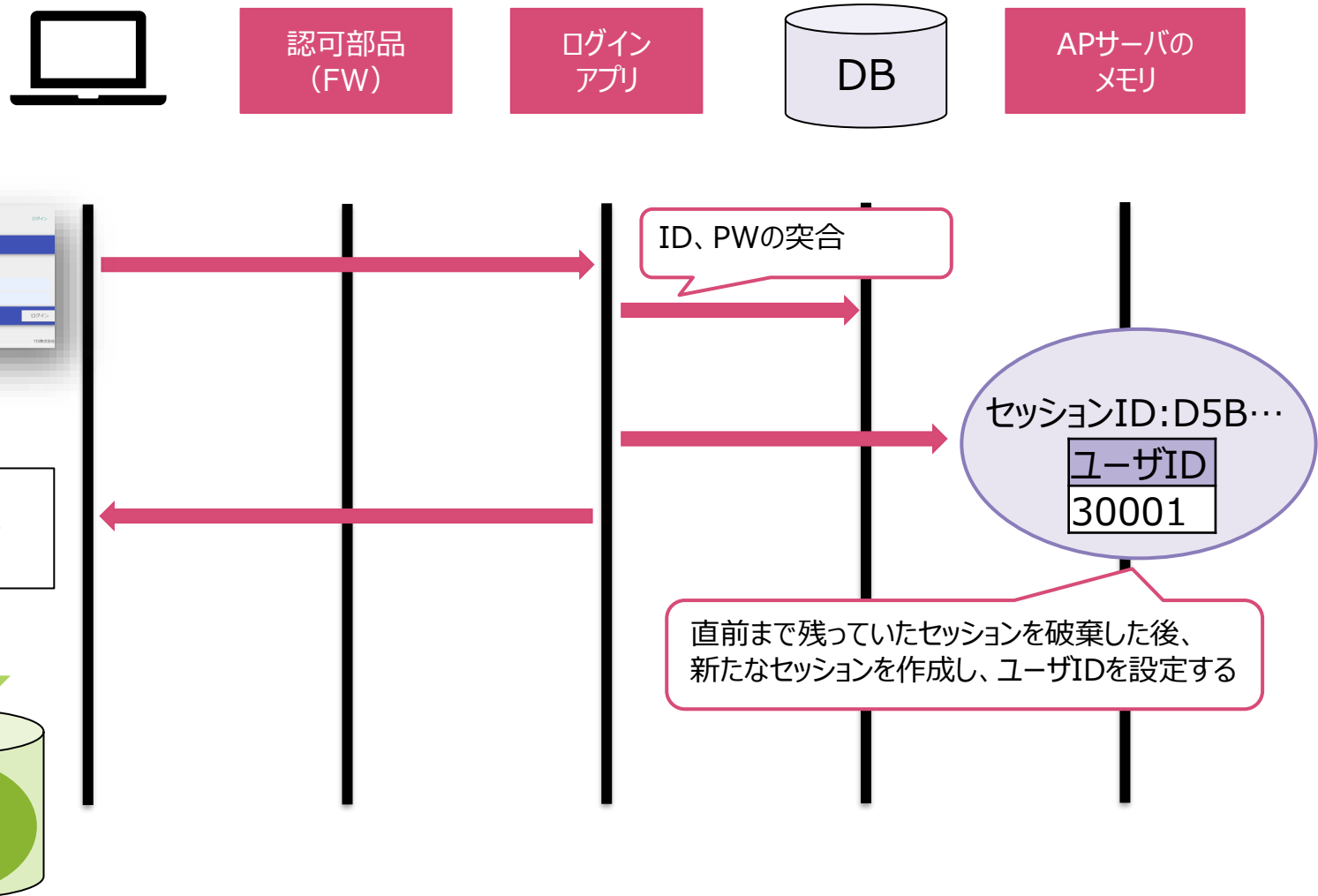
# Form認証の流れ ログイン時 (2/2)



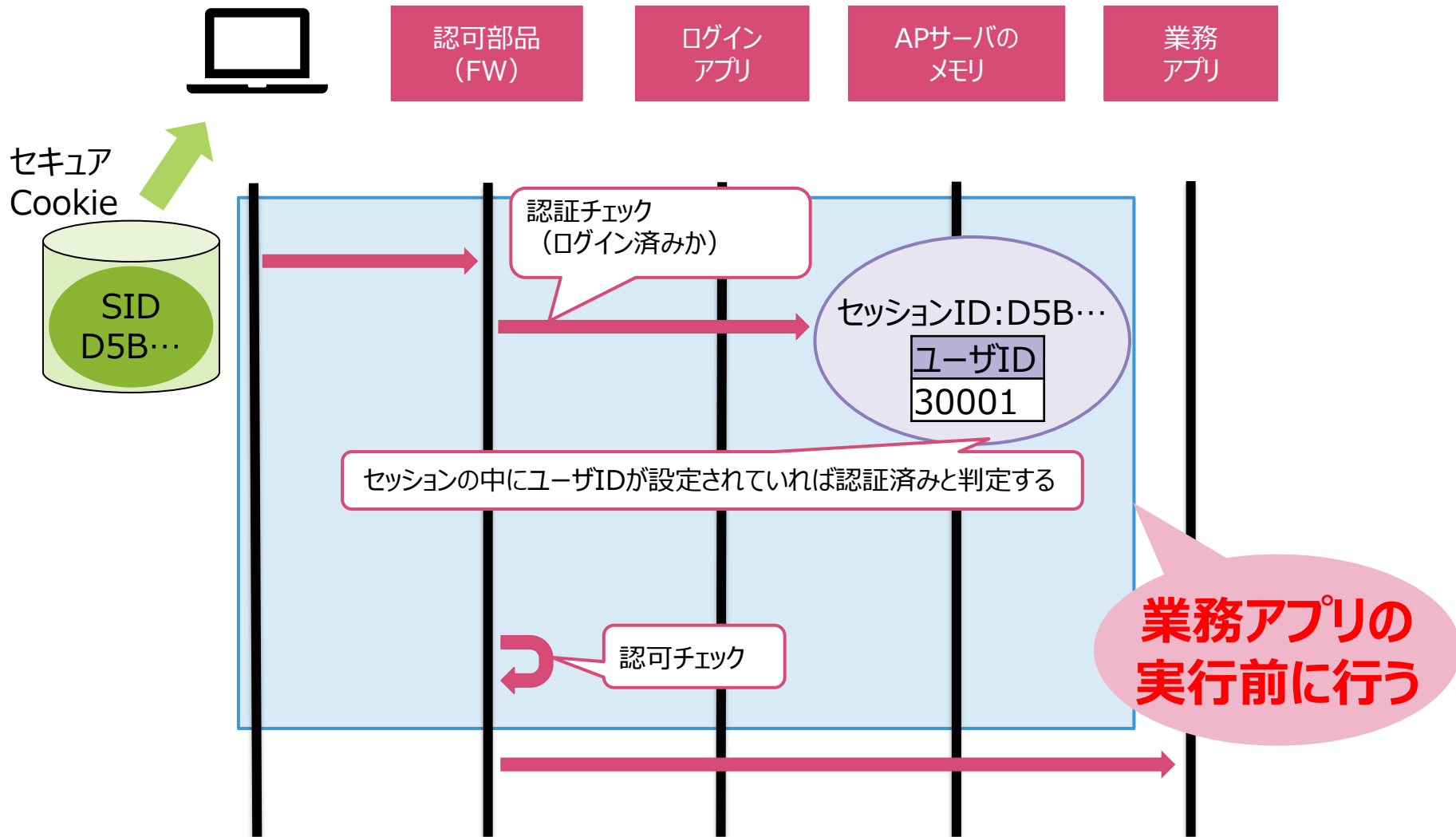
# Form認証の流れ ログイン後



# アプリから見たForm認証の流れ ログイン時



# アプリから見たForm認証の流れ ログイン後



ユーザ認証にログインフォームを用いる認証方法。  
**現在のWebアプリケーションにおける認証方法の主流。**

## メリット

ユーザIDやパスワードなど、認証のための情報をDBに格納して認証処理を行うため、規模の大きなシステムでの実用に耐える。

## デメリット

- ユーザID、パスワード、セッションIDを送信するため、SSL化しないと情報漏洩の可能性が高い。
- 認証の仕組みを実装する必要がある。



## ○DBに保存するパスワードデータ

万が一パスワードデータが漏洩した場合に備えて、DBにはパスワードをそのまま保存せず、ハッシュ化して保存することが一般的。ハッシュ化により元のパスワードを割り出すことは極めて難しくなる。



しかし

同じデータ(文字列)からは常に同じハッシュ値が得られるため、数万人規模のデータが漏洩した場合、同じハッシュ値の人は世間でよく使われているようなパスワードを使っている可能性が高いという推測ができ、アタックの対象になりえる。



そこで

一人ひとりのパスワードに異なる文字列を追加してハッシュ化する手法がとられる。この、追加する文字列をソルトと呼ぶ。

安全にDBに保存するため、

ソルトを用い、HMAC方式でハッシュ化すること。

- HMAC方式：ハッシュ化するための数学的方法。
- ソルトを設定するときの注意点
  - ソルトはユーザ毎に異なる値であること
  - ソルトは充分長い文字列であること
  - ソルトは、パスワード文字列の後ろに付与すること

そうすることで

万が一パスワードデータが漏洩しても、  
元のパスワードが解析される可能性が  
低くなる

現在の主流であるForm認証について、2つ、トピックを紹介します。

- 認証とは
- Form認証に関するトピック
  - マルチタブ・マルチウィンドウ操作時の挙動
  - オートログインの仕組み
- シングルサインオン
- 認可とは

Form認証では、Cookieを用いてセッションIDを保持する仕組みになっており、マルチタブやマルチウィンドウで操作を行った場合、認証状態が共有されます。



具体的に何が起こるのか、2つの部門を兼務しているユーザを例に見てみましょう。

# 認証状態が共有される例: 兼務者 (1/3)

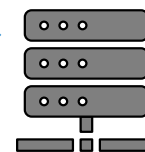
主務のログインIDでログイン



1つ目のタブで  
ログイン



①



②

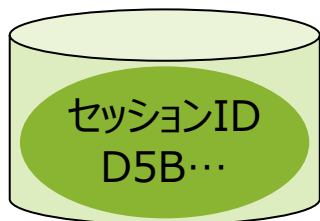
セッションID:D5B...  
ユーザID  
10000001

③

Set-Cookie :  
SID=D5B...; secure

検索結果191件

セキュア  
Cookie



④

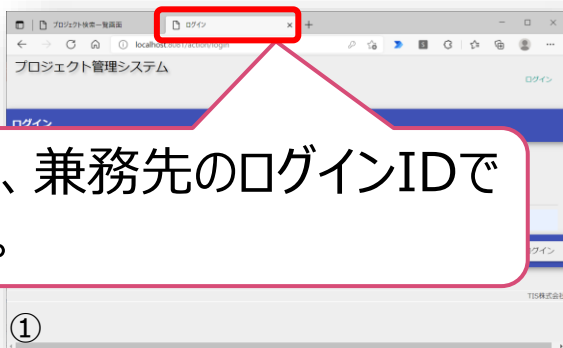


# 認証状態が共有される例: 兼務者 (2/3)

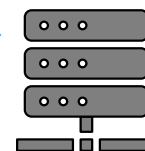
兼務のログインIDでログイン



新しいタブで、兼務先のログインIDでログインする。



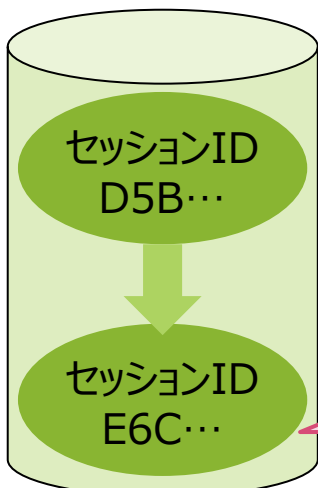
①



②

セッションID: E6C...  
ユーザID  
10000002

セキュア  
Cookie



④



Set-Cookie :  
SID=E6C...; secure

検索結果190件

主務も兼務も同じWebサイト→同じドメイン。  
また、タブ間でCookieは共有される。  
このため、Cookieが上書きされる。

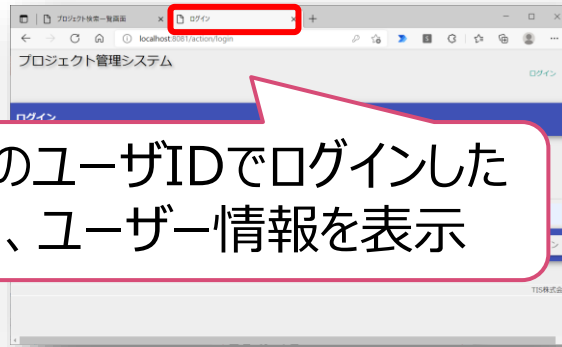
# 認証状態が共有される例: 兼務者 (3/3)

主務のタブで、ユーザー情報を表示

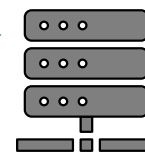


Cookie: SID=E6C...

主務のユーザIDでログインしたタブで、ユーザー情報を表示



②



③

セッションID: E6C...  
ユーザID  
10000002

④

セキュア  
Cookie

セッションID  
E6C...

セッションIDは上書きされているから、兼務でログインした時のもの



タブを戻しても  
検索結果190件

兼務の  
ユーザーの情報が  
表示される

# 認証状態が共有される例 まとめ

兼務者が1つのタブ（またはウィンドウ）で主務のIDでログインし、別タブで兼務のIDでログインする。  
この後、主務のIDでログインしたタブで操作を行ったとき

Cookie がタブ（ウィンドウ）間で共有されるため、Cookieに格納されているセッションIDも共有される。

このため、主務のIDでログインしたタブであっても、後にログインした**兼務のID**として扱われる。  
例えば、ユーザ情報を表示すると兼務のユーザ情報が表示される。

※ブラウザによっては、タブ間、ウィンドウ間でCookieを共有しないこともできる。



- 認証とは
- Form認証に関するトピック
  - マルチタブ・マルチウィンドウ操作時の挙動
  - オートログインの仕組み
- シングルサインオン
- 認可とは

# オートログインとは

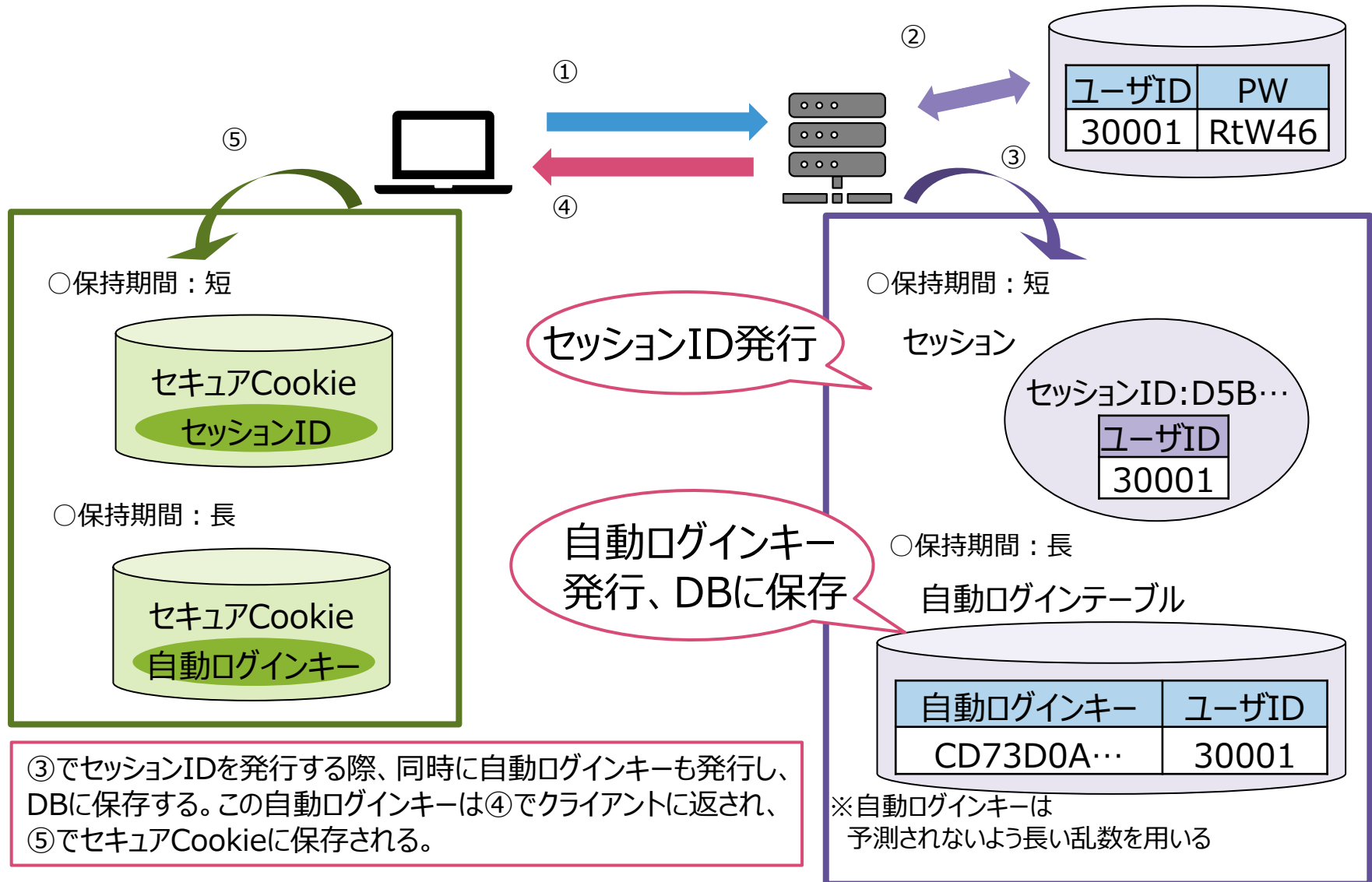
画面操作しないまま数時間～数日経っても、ユーザがログアウトしていなければ、操作を再開した時にID/パスワードの再入力を省略できる仕組み。

ユーザにとっては、ログイン操作が不要となるため、利便性が上がる。

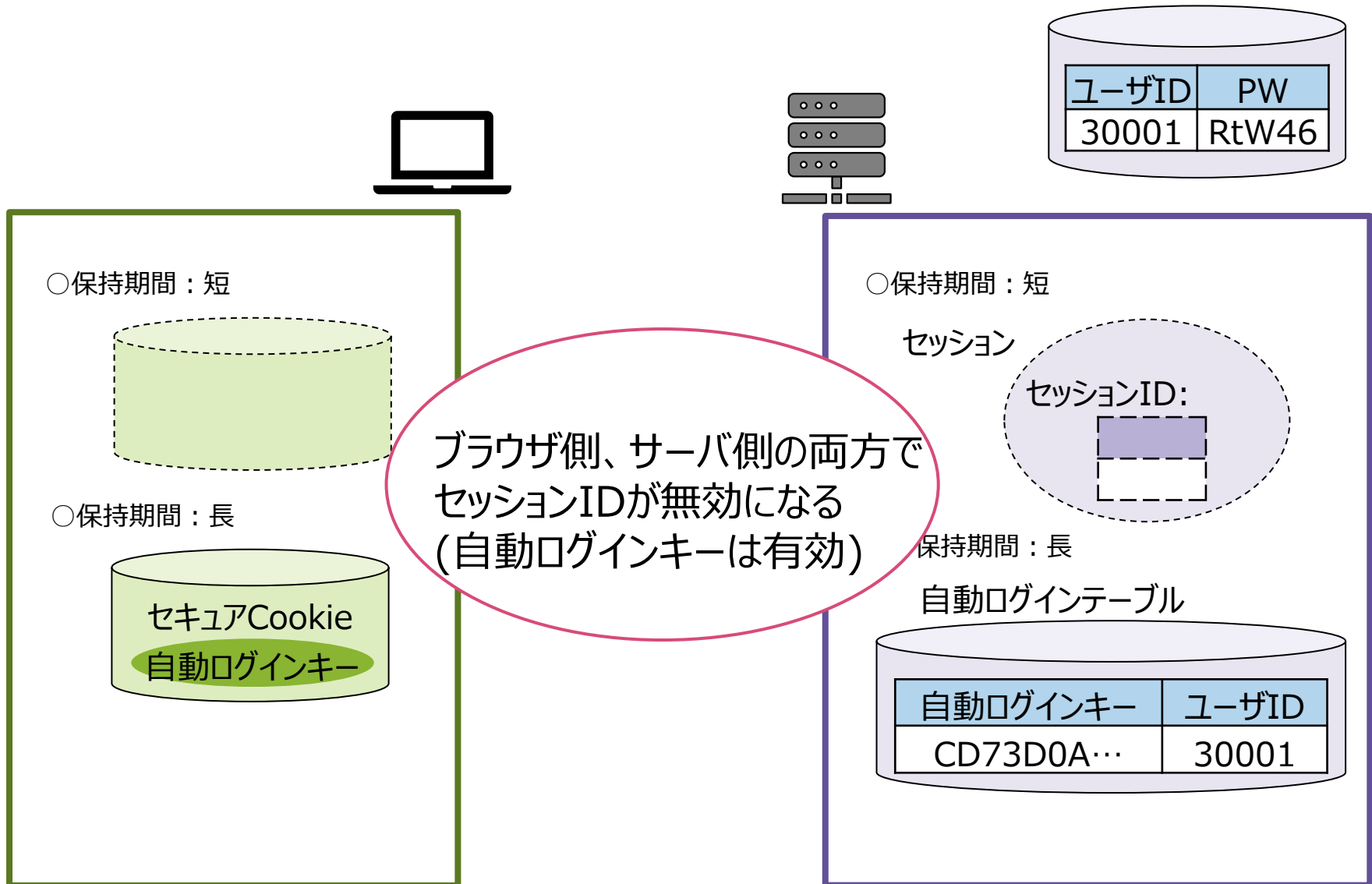
※ログイン画面に「次回から自動ログイン」などのチェックボックスがあることが多い。

ユーザがチェックすることでオートログインが有効になる。

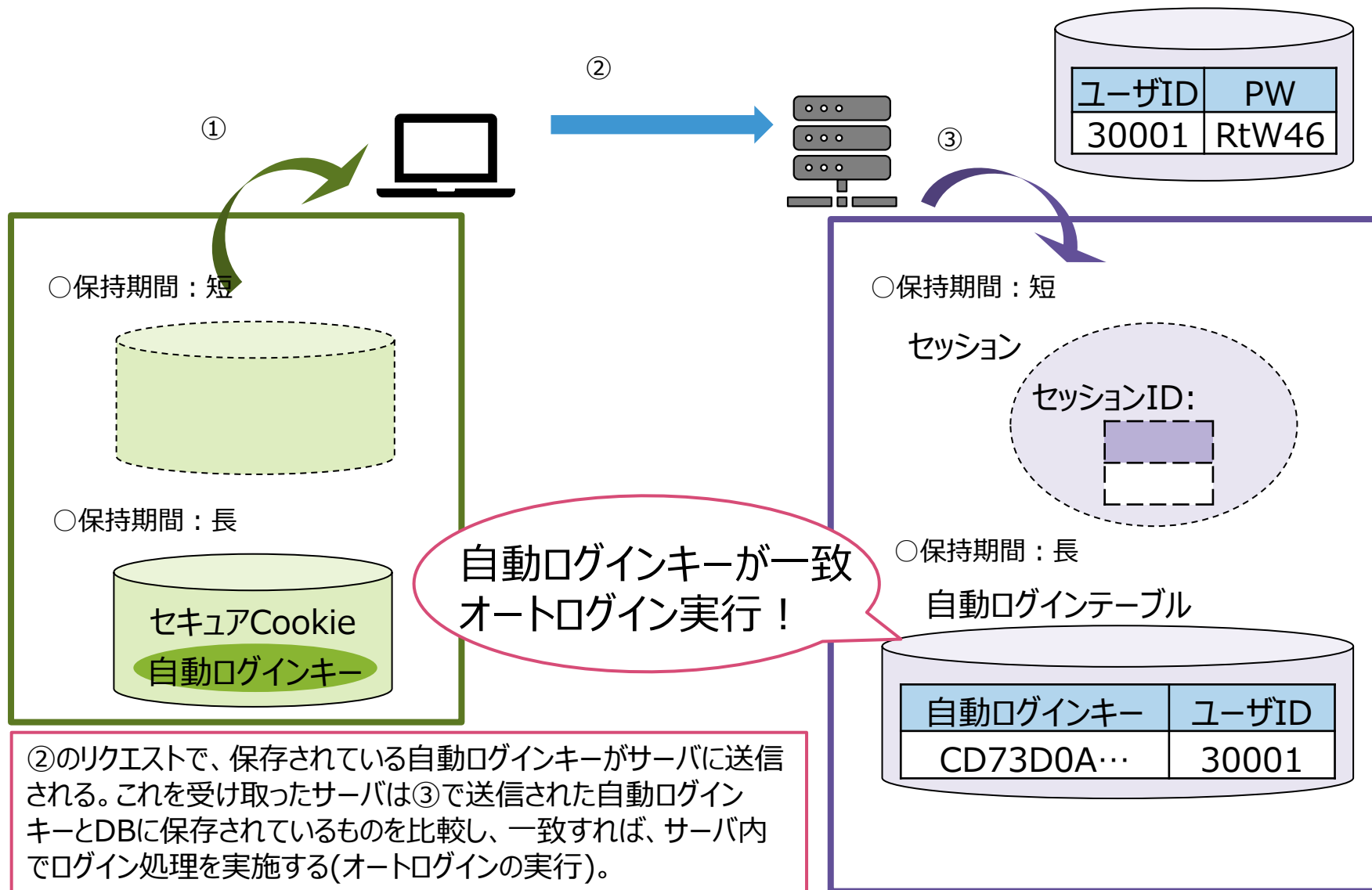
# ログイン時



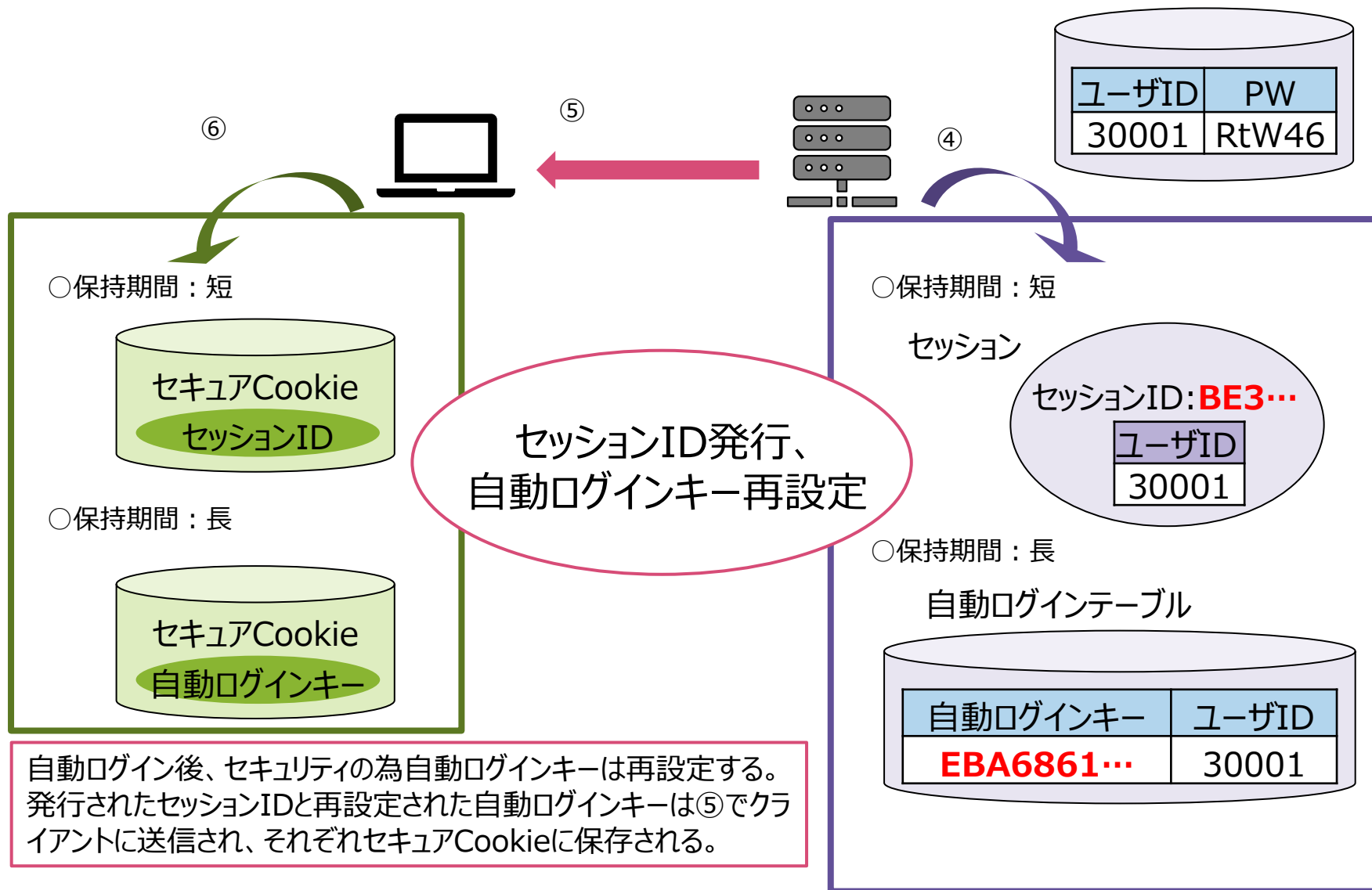
# 長期間経過時（ユーザ操作なし）



# ユーザ再操作時 (1/2)



# ユーザ再操作時 (2/2)



- ログアウト時  
ブラウザ側、サーバ側の両方で  
セッションと自動ログインキーを無効にすること。
- オートログインオプション変更時  
オートログインの設定が「オン→オフ」に変更されたときは  
対象ユーザの自動ログインテーブルのレコードと  
Cookieの自動ログインキーを削除する処理が必要になる。

- 認証とは
- Form認証に関するトピック
- シングルサインオン
- 認可とは



# シングルサインオン（SSO）とは

---

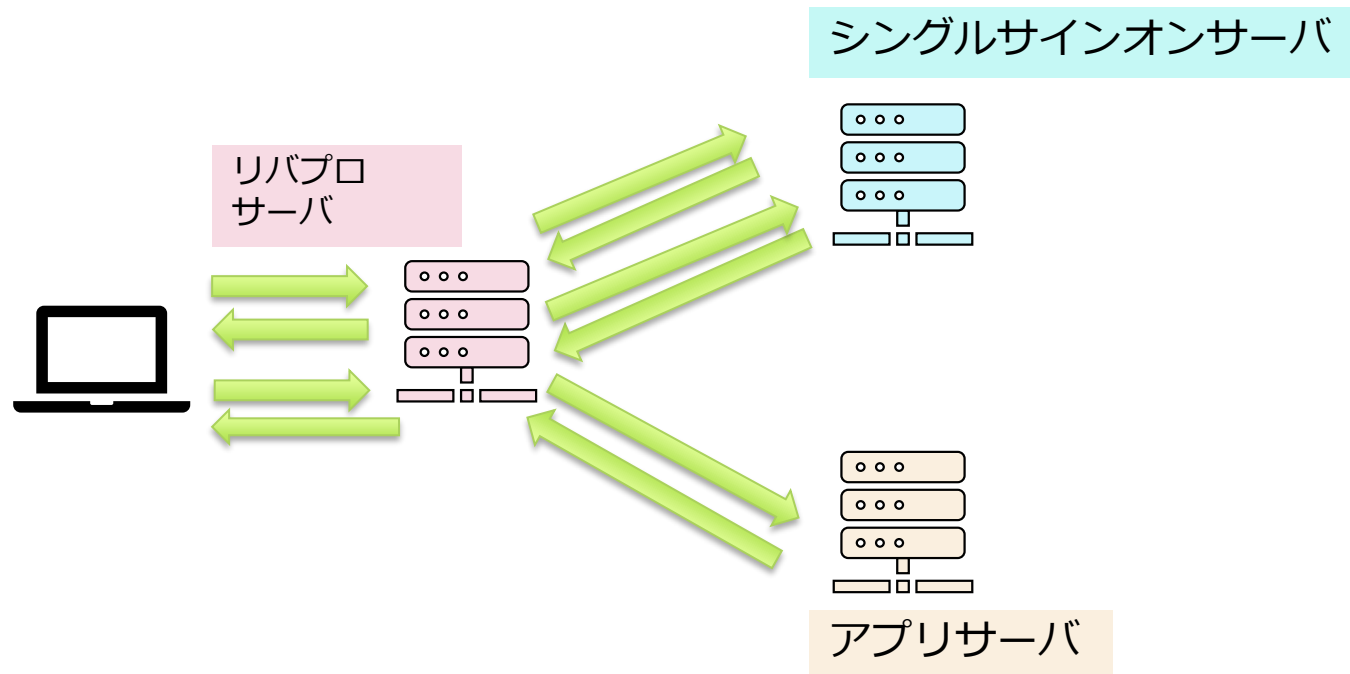
1回の認証手続きで複数のアプリケーション、サービスなどにアクセスできること。  
略して「SSO」と呼ばれることもある。

- リバースプロキシ(リバプロ)方式
- エージェント方式
- 代理認証方式
- フェデレーション方式
- 透過型方式  
など

今回は、既存システムへの変更点が少なく、導入しやすいメリットがある、リバプロ方式の仕組みを詳しく説明します。

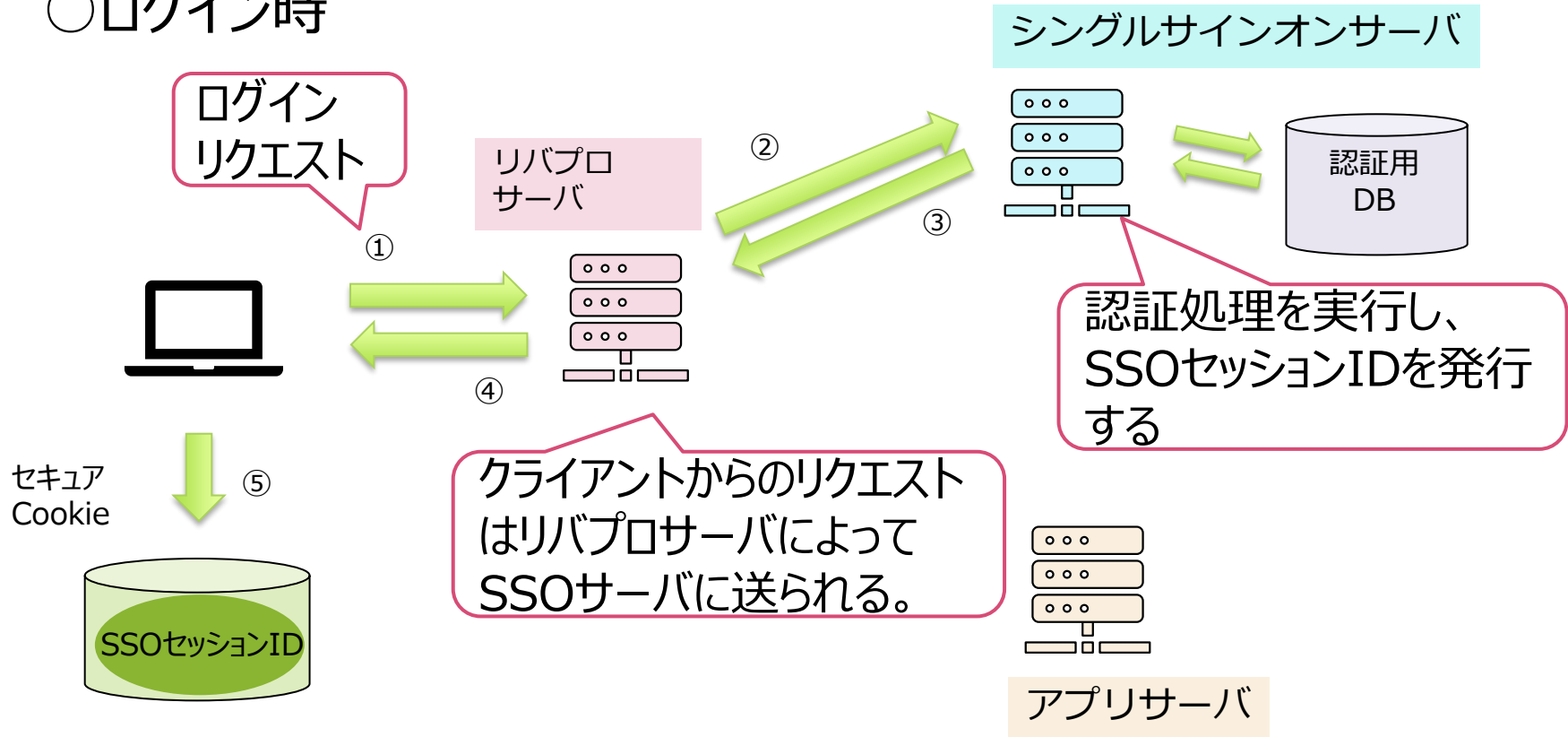
# リバースプロキシ(リバプロ)方式の仕組み (1/3)

Webブラウザからのアクセスを一度リバースプロキシサーバーが受け、そのリクエストをバックエンドに置かれたシングルサインオンサーバやアプリサーバーに中継する。



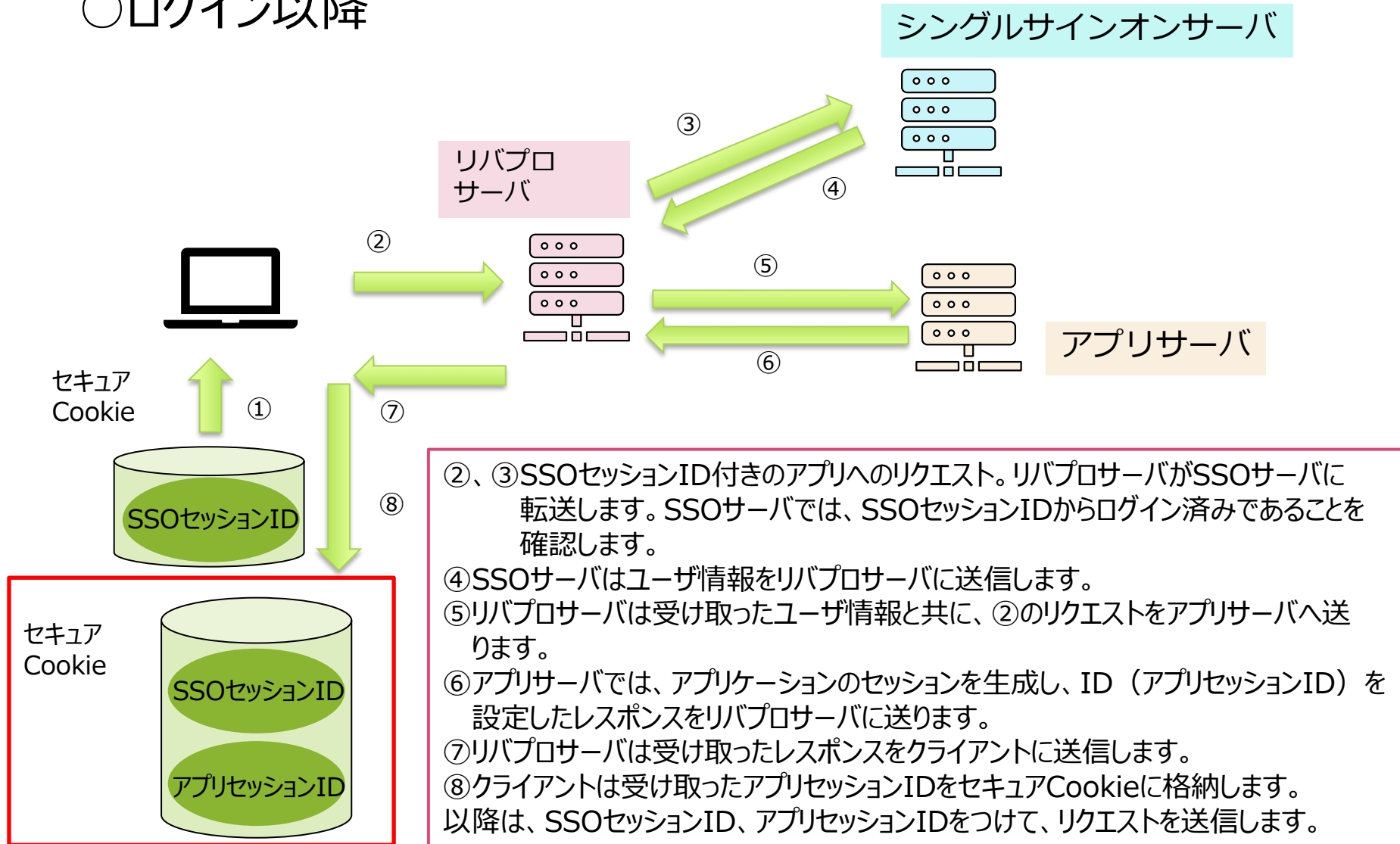
# リバースプロキシ(リバプロ)方式の仕組み (2/3)

## ○ログイン時



# リバースプロキシ(リバプロ)方式の仕組み (3/3)

## ○ログイン以降



- パスワード管理の負担が減る  
ユーザは1つのパスワードを覚えるだけで済むため、パスワードを忘れる可能性が低くなる。
- 認証システムの管理コストが減る
  - 個々のシステムで認証の仕組みを作る必要がなくなる。
  - ユーザが使用するアカウントが統一されるため、パスワードのリセットなど、ユーザからの依頼作業が減る。
- セキュリティを高められる  
ログインに関するログが、全てSSOサーバに記録されるため、不正アクセスが発見しやすくなる（各APサーバのログをそれぞれ見て回らなくても、SSOサーバのログだけ見れば、ログイン状況や履歴を確認できる）。

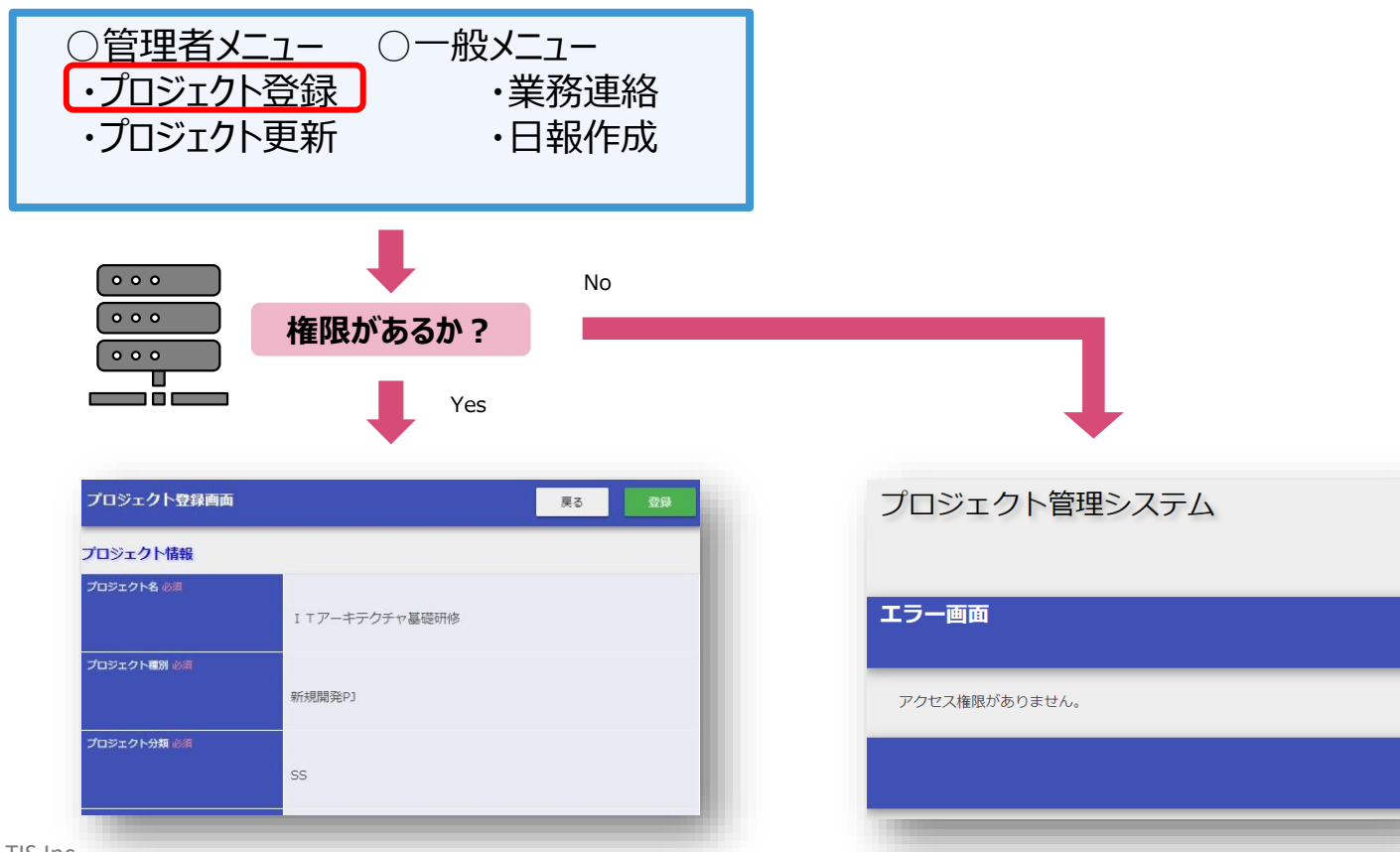
- ID、パスワード漏洩時のリスクが大きい  
シングルサインオンで使用するアカウントの情報が漏れると複数のアプリケーションにログインできてしまう。
- システムダウン時のリスクが大きい  
SSOを実現するシステムが停止すると、複数のアプリケーションにログインできなくなってしまう。
- コストが高い  
オンプレミス型の場合は初期費用が高額な場合が多く、クラウド型は初期費用は安価だがランニングコストがかかることが多い。

- 認証とは
- Form認証に関するトピック
- シングルサインオン
- 認可とは



## 認可【Authorization】

アクセス権限の制御を行い、ログインユーザに使わせていい機能だけ使わせる。



要件にあった認可を実現するために重要なこと

## データモデル

必要なデータをどのように格納するか？



一例として、Nablarchの場合を見てみましょう。

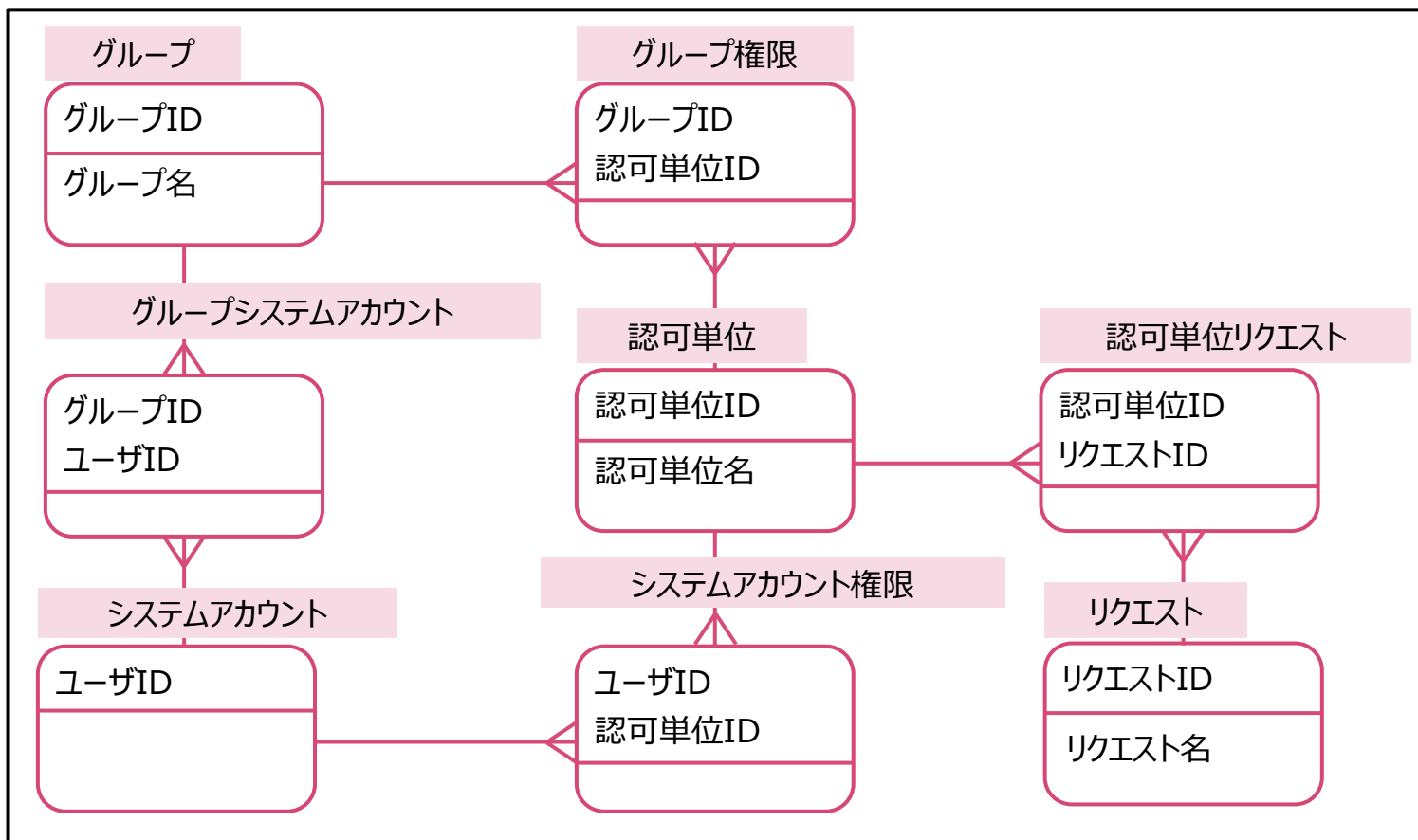
※Nablarch は、Javaアプリケーション開発/実行基盤です。  
詳しく知りたい方は以下を参照ください。

Fintan > Nablarch アプリケーションフレームワーク

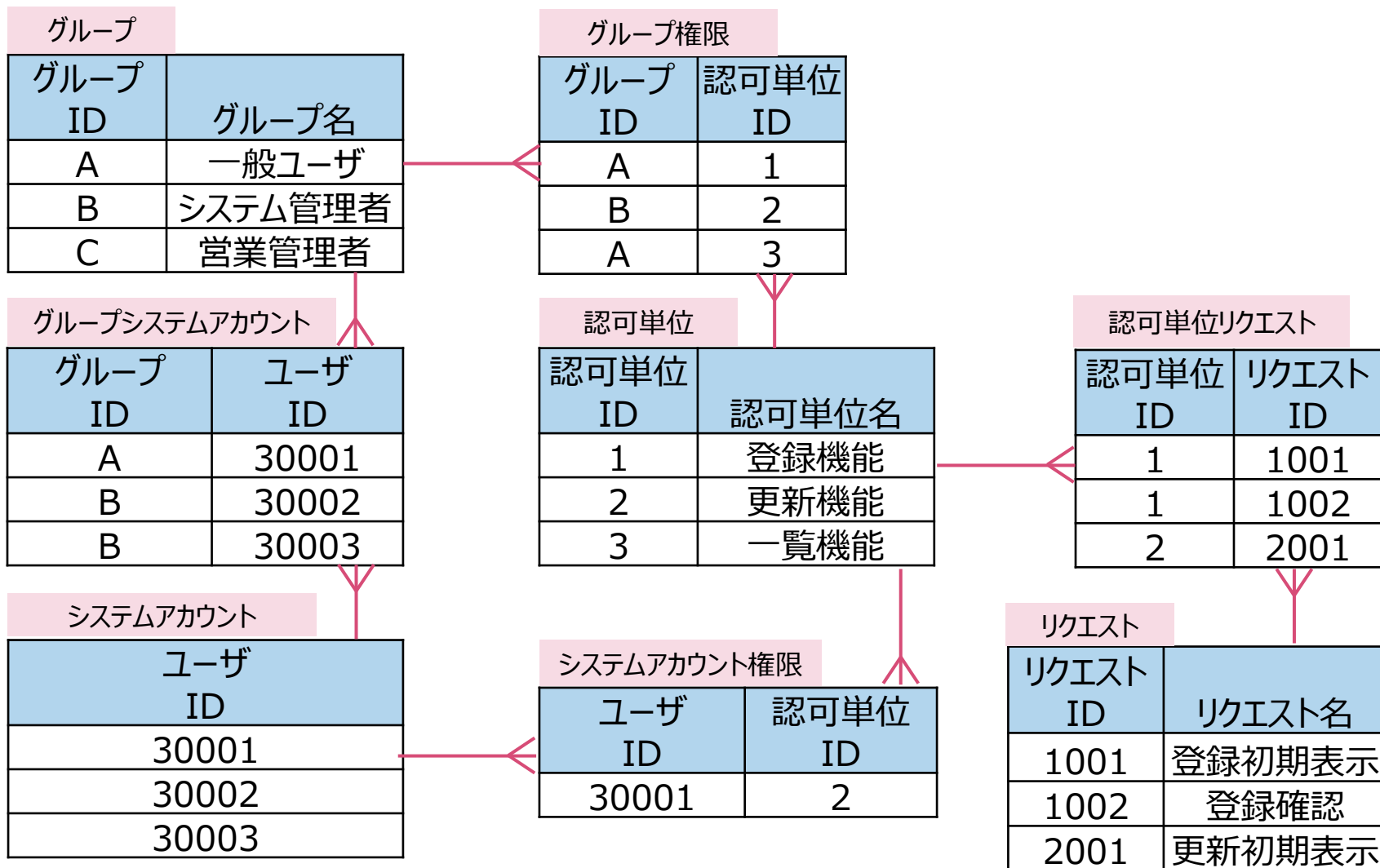
<https://fintan.jp/?p=304>

## •データの持ち方

Nablarchの認可チェック機能では、以下のようなデータモデルで認可チェックに使用する権限データを管理している。



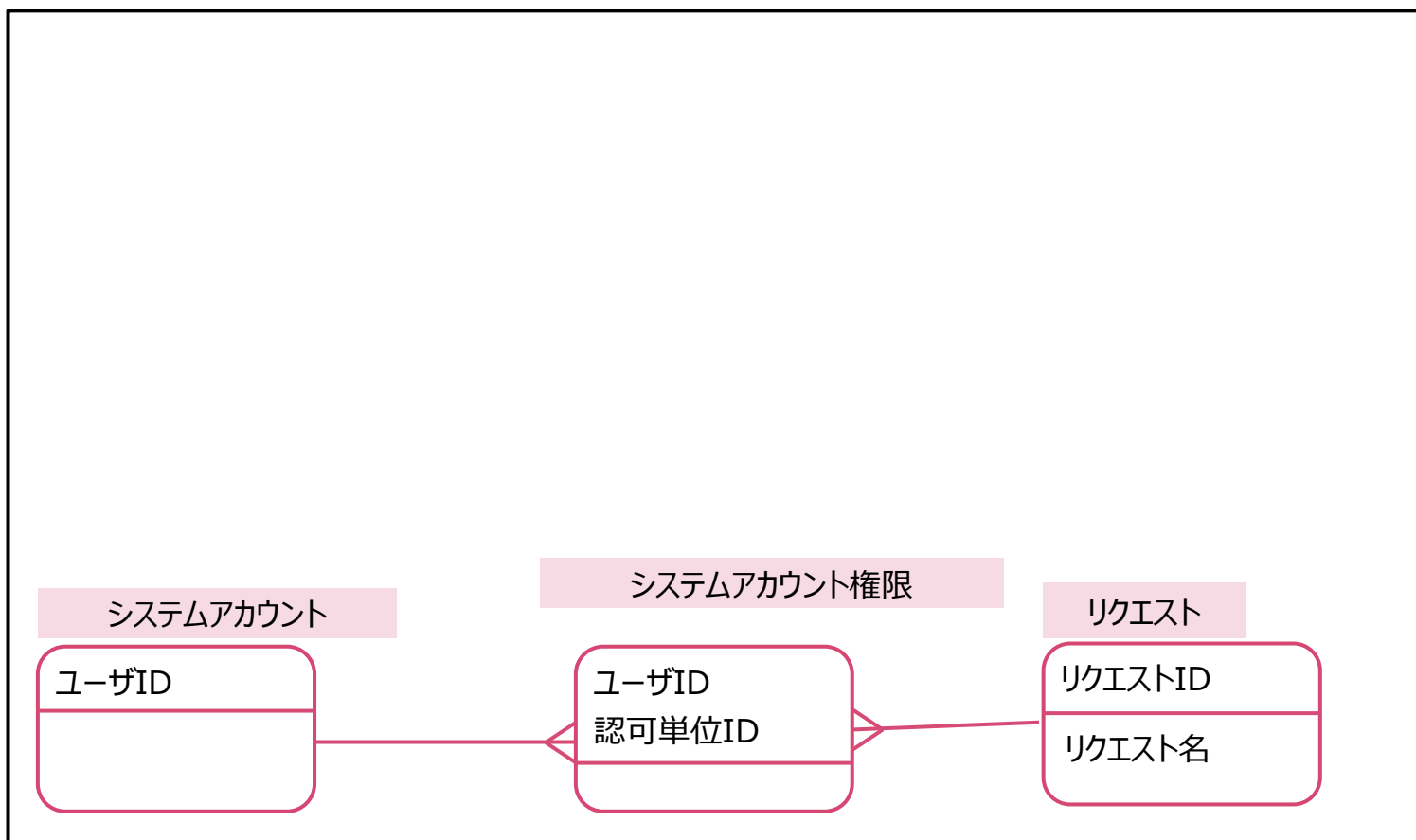
# Nablarchの認可チェックのデータの例



単純な要件から少しずつ複雑にしていって  
見て行きましょう。

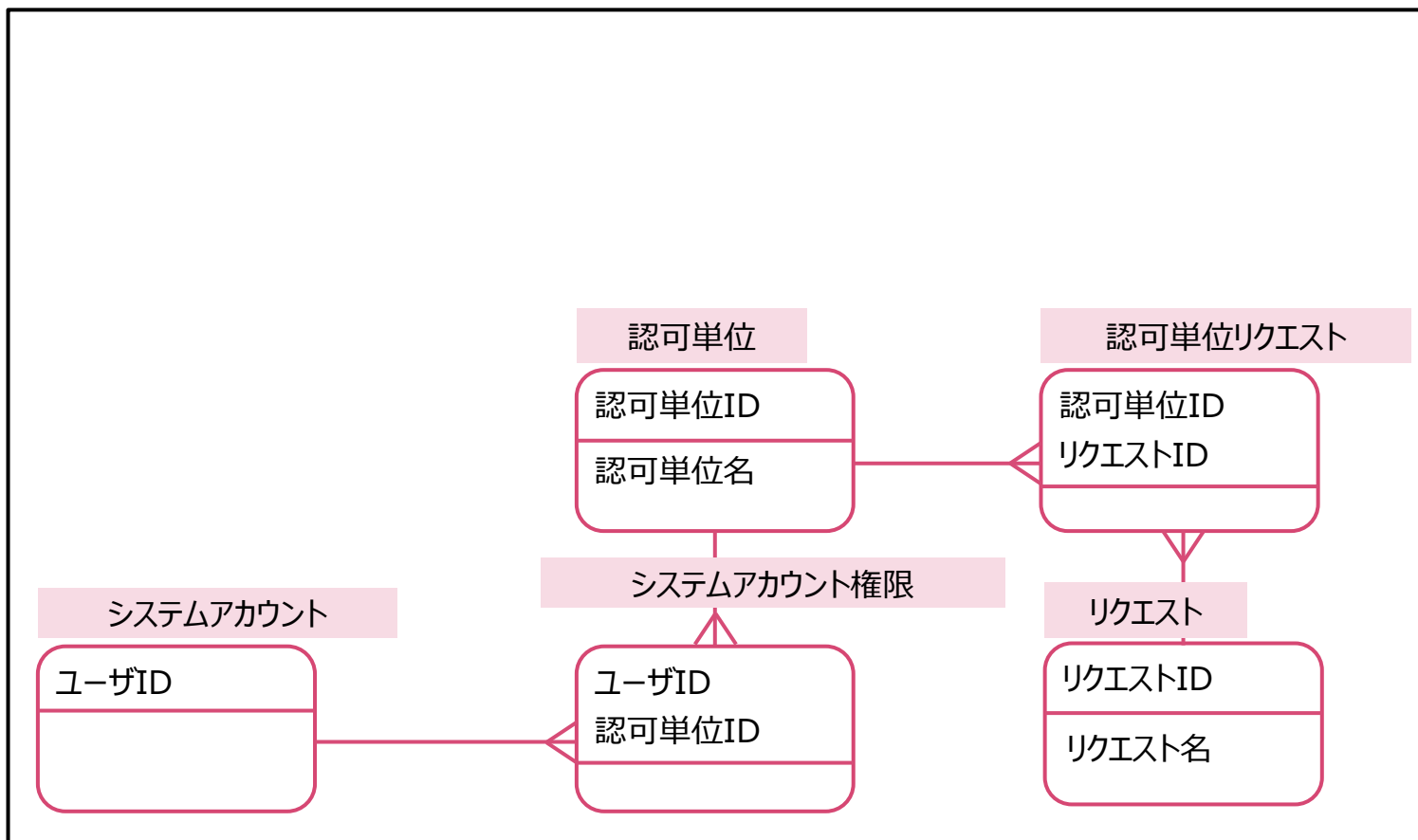
# 画面を表示できるアカウントを制限したい

ある画面を表示するためのリクエスト(URL)と、それを使用できるアカウントを結びつける



# 1つの機能は複数のリクエストからできている

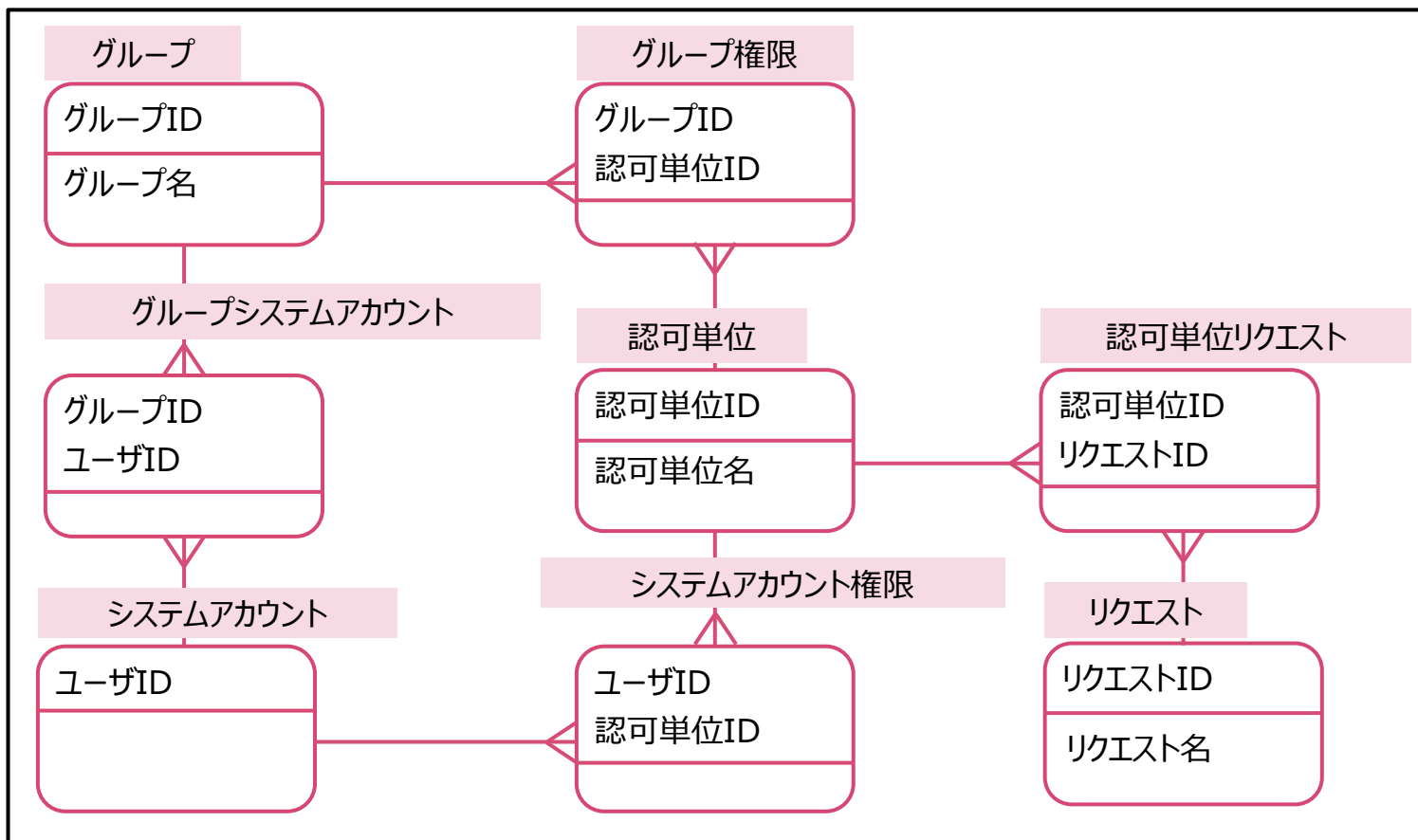
リクエストの塊（認可単位リクエスト）と、それを使用できるアカウントを結びつける



# 同じ役割を持ったユーザをまとめたい

システムアカウントをまとめたグループを作り、このグループと認可単位リクエストを結びつける

※グループ：管理者やある部門のメンバといった、役割毎に定義することが一般的

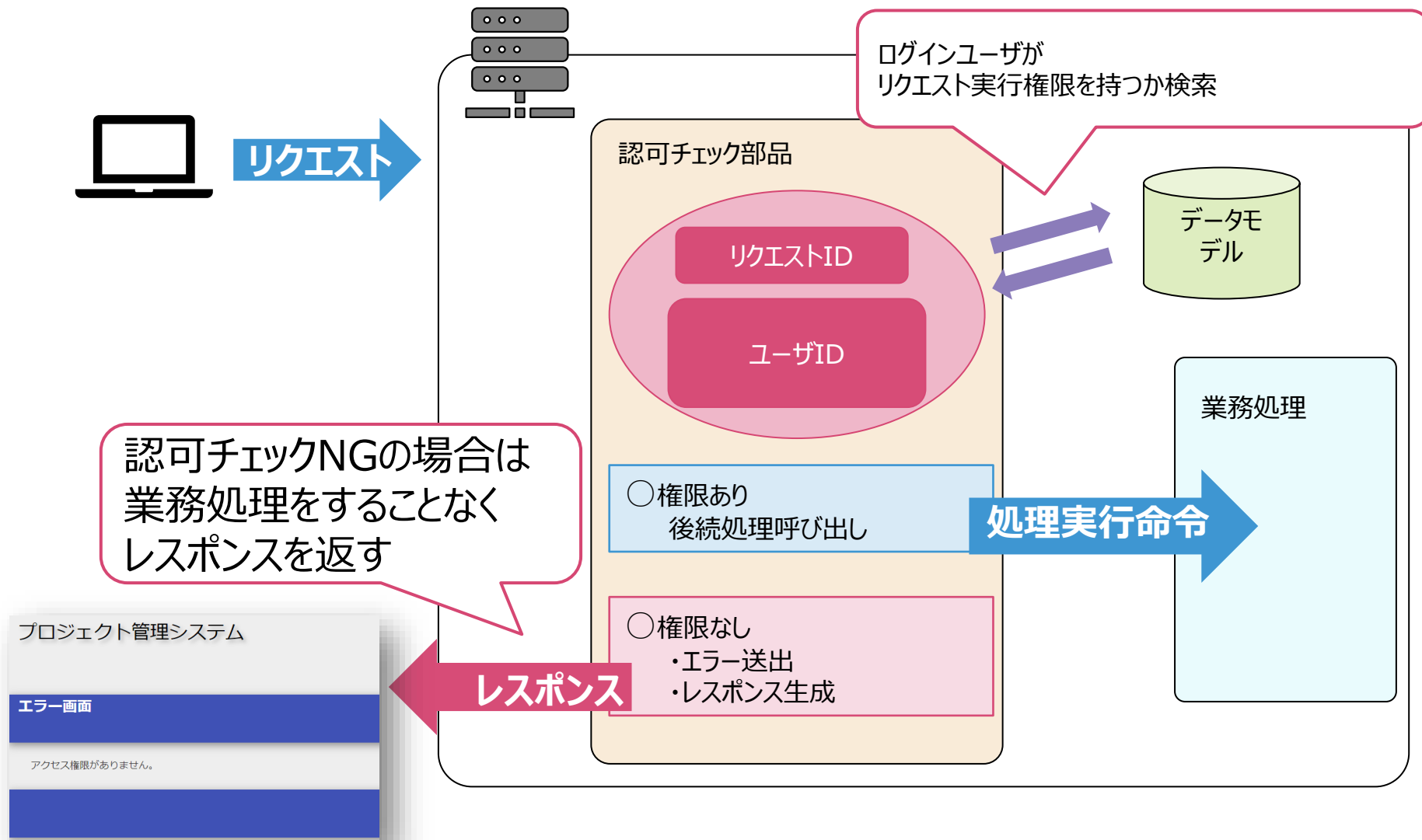




認可チェック処理そのもののポイントは、

どんな場合でも、認可チェック処理を漏らさないこと

# Nablarchの認可チェック処理例



# 入力値のバリデーション

---

入力値のバリデーションは、実行する場所によって、大きく二つに分けることができます。

- クライアントで行うバリデーション
- サーバで行うバリデーション



それぞれ、どのような実現方法があるのか説明します。

## HTML5の入力値チェック

HTMLタグにバリデーションルールを設定。

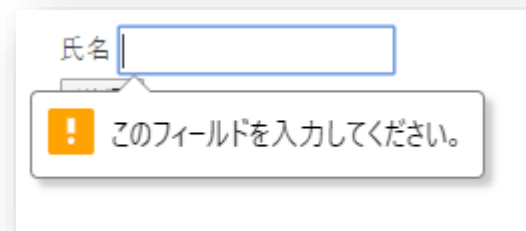
- require属性：必須項目
- maxlength属性：最大文字数の指定
- pattern属性：正規表現パターンによるバリデーションなど

例：必須項目



A form with a text input field labeled '氏名' (Name) and a '送信' (Submit) button. A red arrow points to the input field.

送信ボタン押下



The same form after submission. The input field is highlighted with a blue border, and an error message box appears below it: 'このフィールドを入力してください。' (Please enter this field).

`<input type="text" required>`

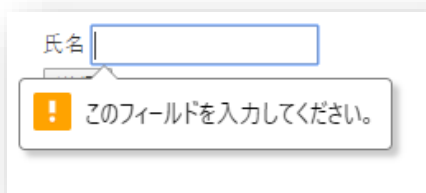
# クライアント側の手法① HTML5 (2/2)

## メリット

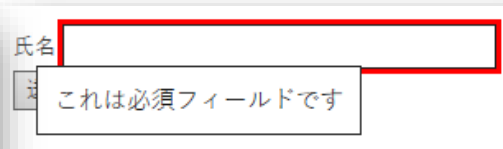
- 実装がとても簡単。
- 入力長や入力タイプの制御も同時に行える。

## デメリット

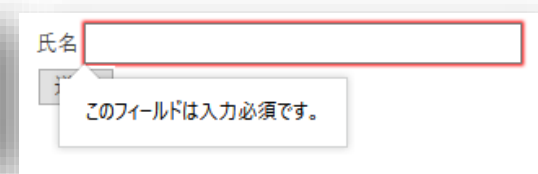
- 複雑なバリデーションは実現できない。
- 送信ボタンを押下するまで実行されない。
- エラー表示のレイアウトなどブラウザによって挙動が異なる。



Chrome 69

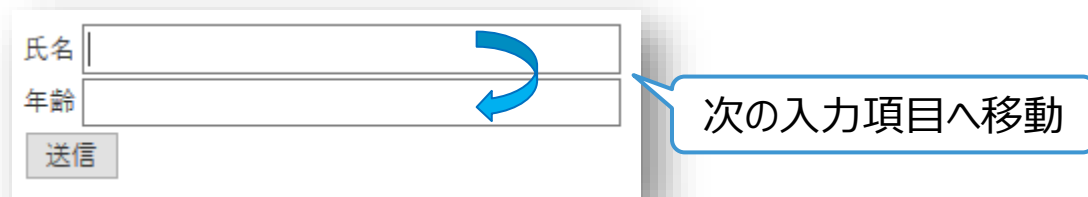


Edge 41



Firefox 62

## JavaScriptによる入力値チェック



氏名

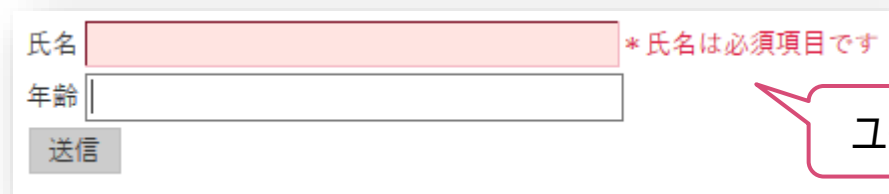
年齢

送信

次の入力項目へ移動

The diagram shows a form with two input fields: '氏名' (Name) and '年齢' (Age). A blue arrow points from the '氏名' field to the '年齢' field, indicating the focus moves to the next input item. A callout bubble points to this arrow with the text '次の入力項目へ移動' (Move to the next input item).

JavaScriptによるバリデーション処理実行  
(氏名項目の必須チェック)



氏名  \* 氏名は必須項目です

年齢

送信

The diagram shows the same form after a validation error. The '氏名' field is highlighted with a red border, and a red message '\* 氏名は必須項目です' (Name is a required item) is displayed next to it. A callout bubble points to this message with the text 'ユーザに入力値の修正を促す' (Prompt user to correct input value).

ユーザに入力値の修正を促す

## メリット

- 入力直後にユーザへバリデーション結果をフィードバックできる。
- 複雑なバリデーションルールの実装や項目間での相関チェック、AjaxによるDB精査などができる。
- よりユーザビリティの高い挙動を実現できる。  
(エラー時に送信ボタンを非活性化するなど)

## デメリット

実装難易度が高く、処理が複雑になりがち。  
(ライブラリの活用で複雑度を抑えることはできる)



単項目のバリデーションでは基本的にフレームワークで提供されるバリデーション機能を利用する。

- 文字種や文字数のバリデーション
- メール形式などのパターンチェック
- 必須項目 など

アプリでバリデーション処理を実装する必要はないが、バリデーションルールの設定は必要。

例) BeanValidationの場合はJavaBeansクラスにアノテーションを設定する。

フレームワークのバリデーション機能で実現できない場合は、その処理を実装する。

- DBに対応する値が存在するか
- 業務仕様上許容される値の組み合わせか など

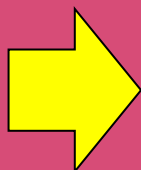
クライアント側でどれだけバリデーションをしても、サーバ側のバリデーションは必須！



なぜでしょうか？

Webページに入力値チェック・制御の仕組みをどれだけ施しても、不正な値の送信を防ぐことはできないため。

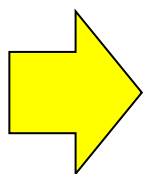
- ブラウザでHTMLを書き換えて送信する
- ツールを使って直接リクエストを送信する



クライアント側のバリデーションはいくらでもすり抜けられる

クライアント側のバリデーションはあくまでユーザが誤ったデータを入力しないよう補助するためのもの。

サーバ側でバリデーションをしないと、不正な入力値をそのままサーバ側で処理してしまう危険性がある。



想定外のエラーによるシステム障害や不正データの登録などの危険性がある。

クライアント側もサーバ側も、入力値チェックですが、目的が異なります。

- クライアント側のバリデーション
  - ユーザの利便性向上のため。
- サーバ側のバリデーション
  - 異常な値を業務処理で使用しないようにするため。

# 二重サブミット対策

---

二重サブミットについて、以下の順序で説明します。

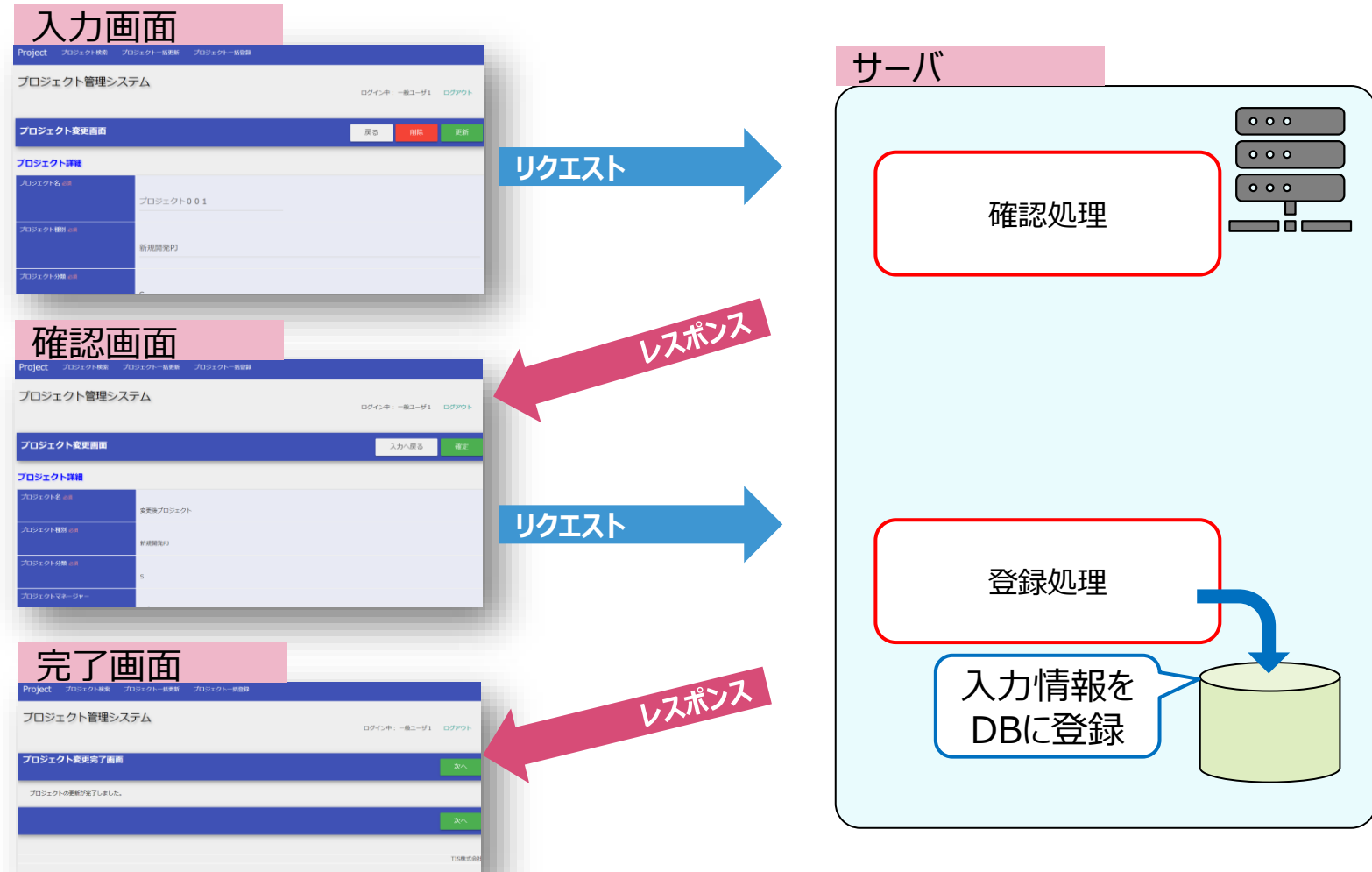
- 二重サブミットが発生する状況
- 二重サブミット対策の手法



- 二重サブミットが発生する状況
- 二重サブミット対策の手法

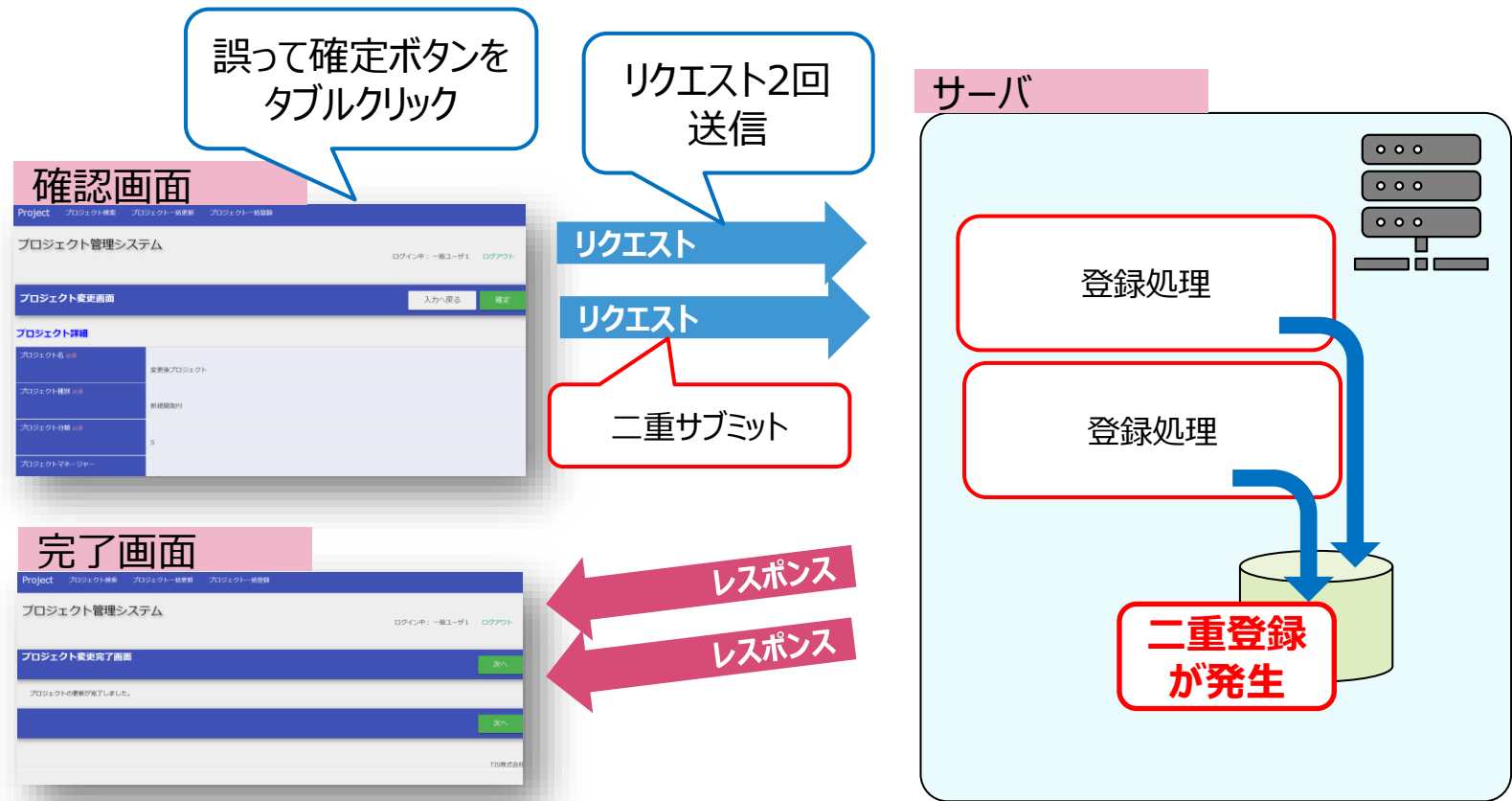
# 二重サブミットが発生する状況

## 入力⇒確認⇒完了というフローの登録処理の場合



# 二重サブミットが発生する状況 その①

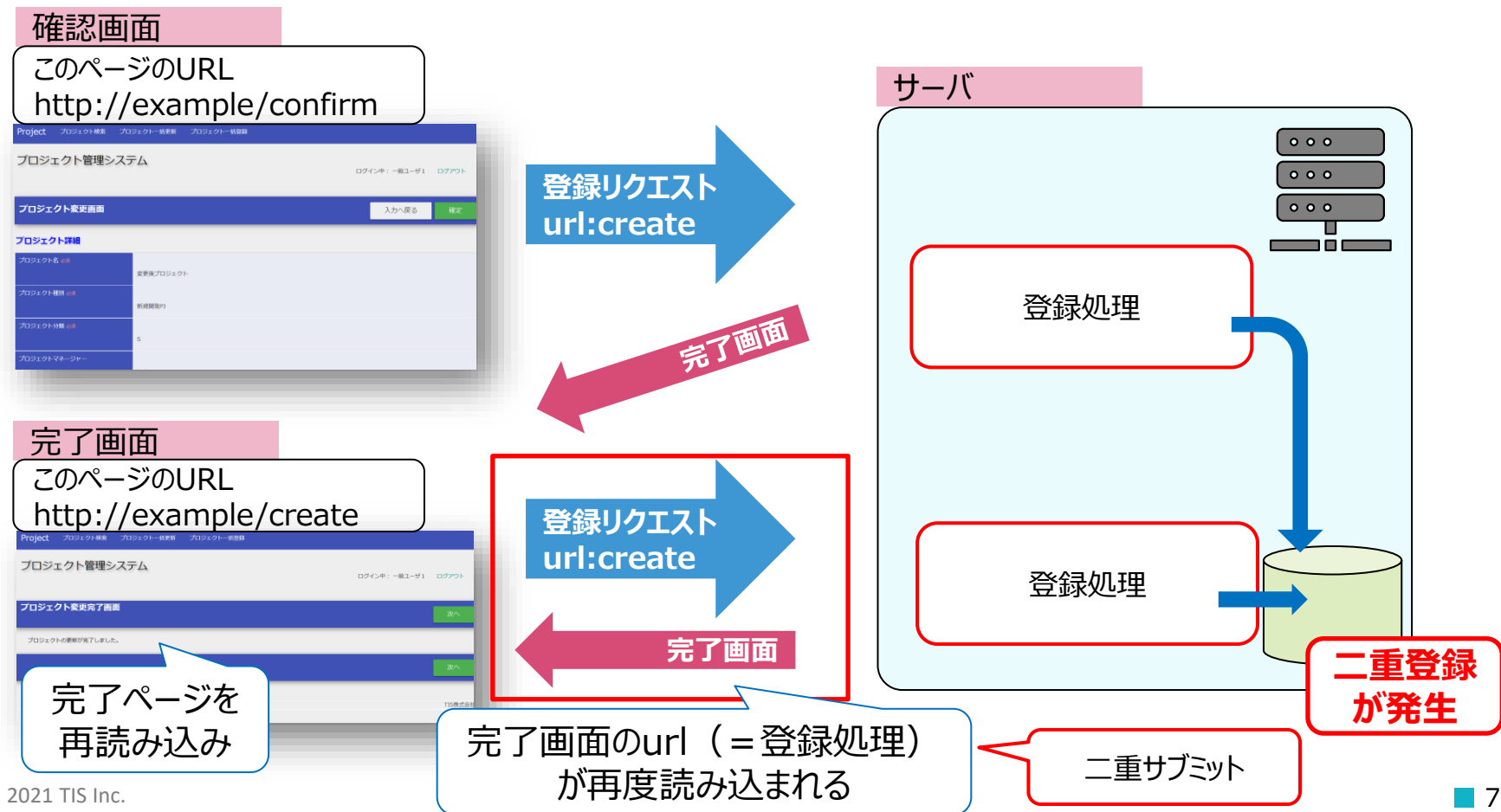
- ① 登録確定ボタンを連続して押下。  
(1回目のレスポンスが返ってくる前にもう一度クリック)



# 二重サブミットが発生する状況 その②

## ② 登録完了ページの再読み込み。

- 完了ページで更新ボタン(F5)を押下した。
- ブラウザを起動した際、最後に開いていたページとして再読み込みしてしまった。 など

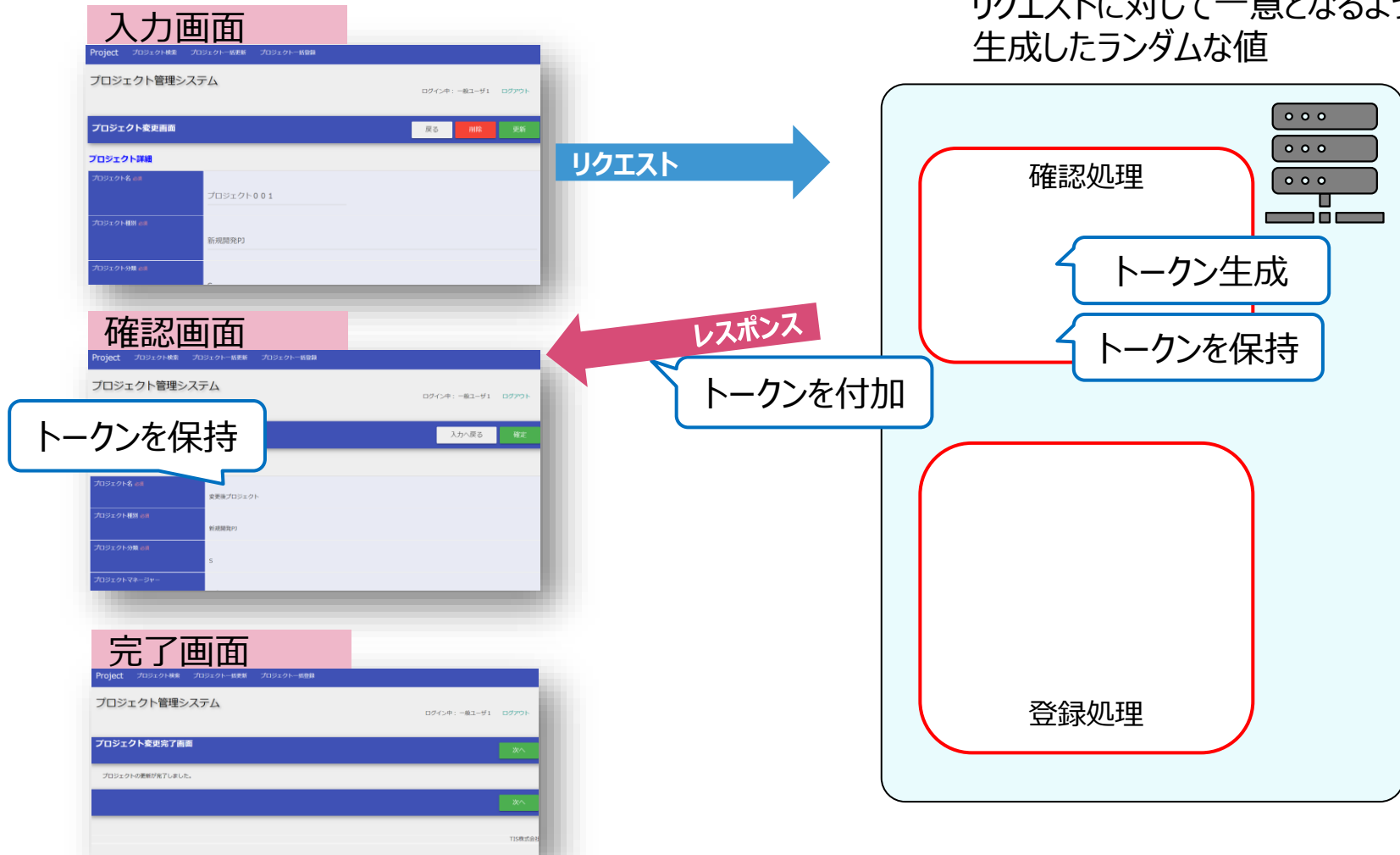


- 二重サブミットが発生する状況
- 二重サブミット対策の手法

# 二重サブミット対策の手法

トークンによるチェックにより実現する。

※トークン  
リクエストに対して一意となるように  
生成したランダムな値

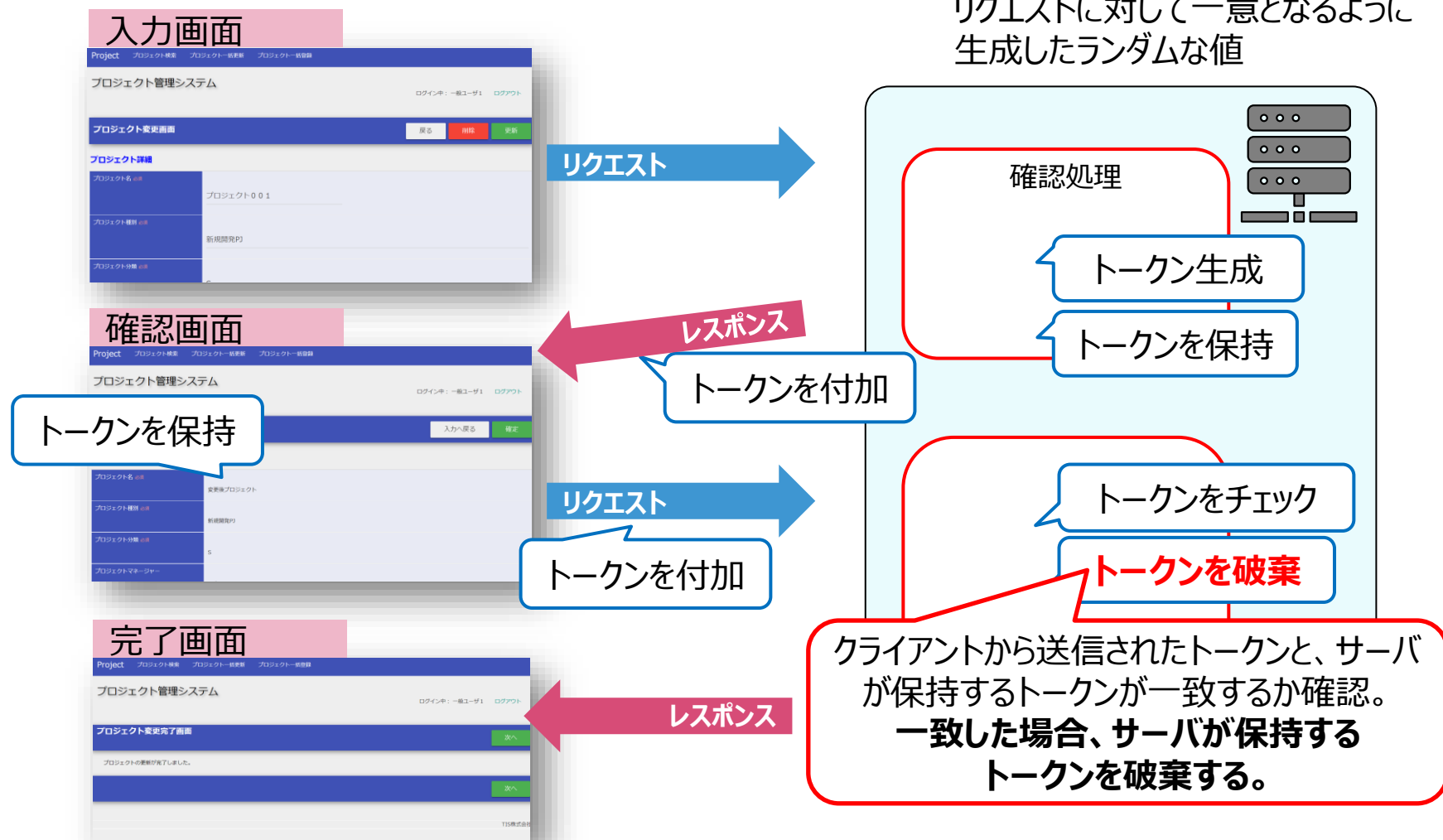


## 二重サブミット対策の手法

トークンによるチェックにより実現する。

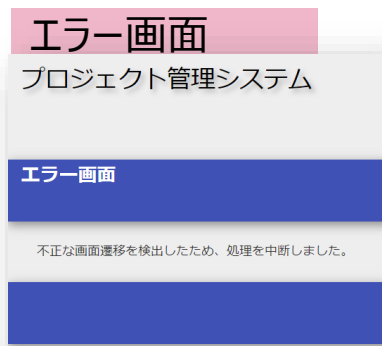
## ※トークン

リクエストに対して一意となるように  
生成したランダムな値



# 二重サブミットが発生した場合

Aのリクエストに対するレスポンスを受信する前に、Bのリクエストを送信してしまった場合。BのリクエストではトークンチェックがNGとなり、二重サブミットエラーのレスポンスが送信される。



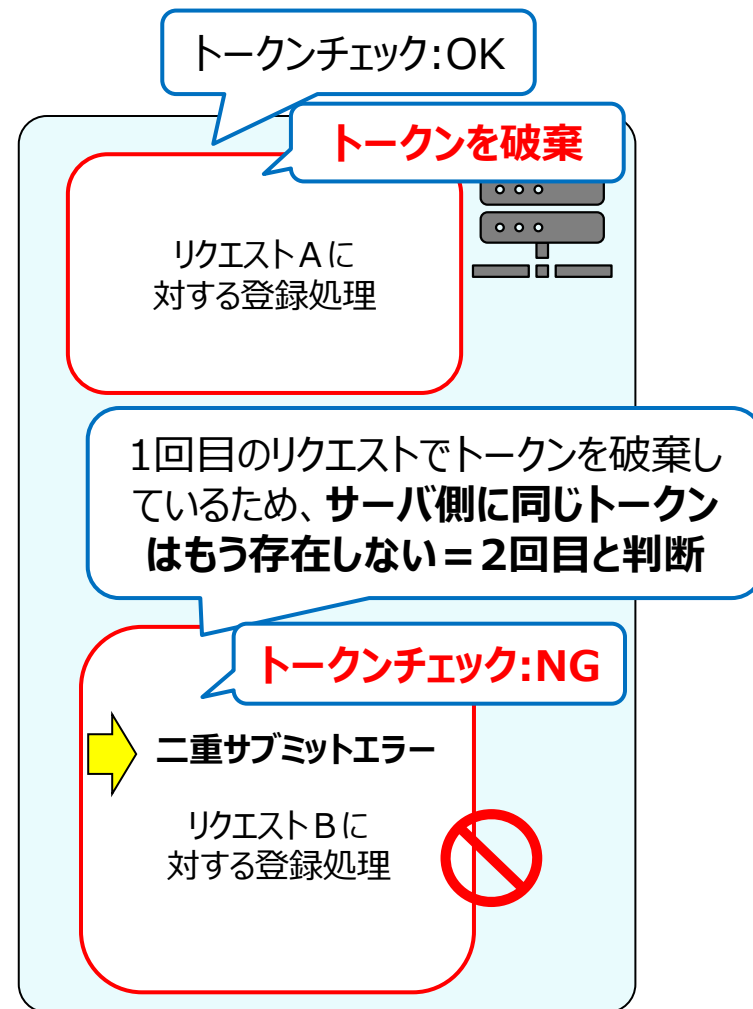
リクエストA  
リクエスト

リクエストB  
リクエスト

Aのレスポンス  
レスポンス

エラーページ

Bのレスポンス  
レスポンス

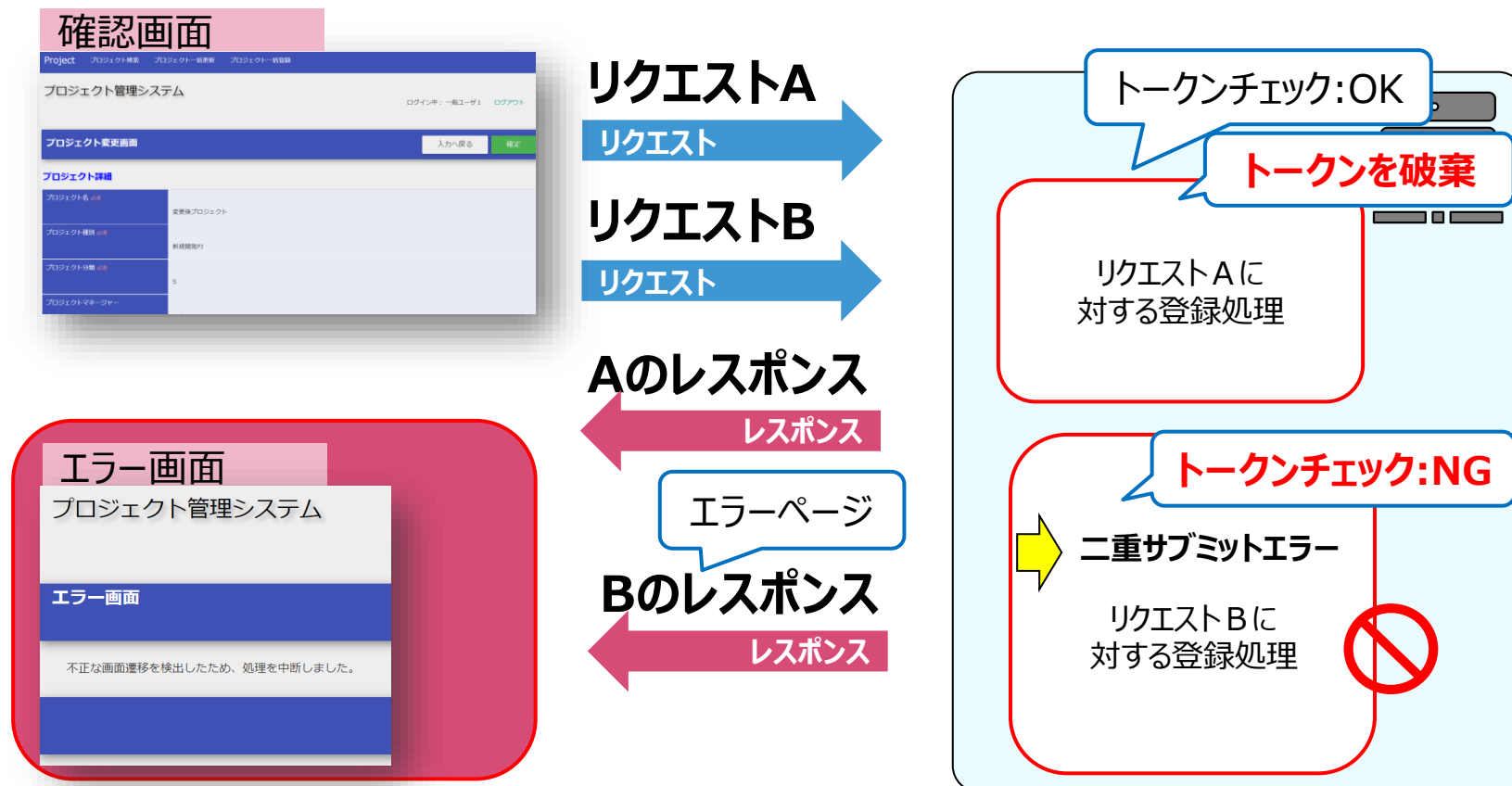




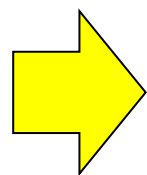
# 二重サブミットが発生した場合

二重サブミット発生時は、最終的に「不正な画面遷移」といったエラー画面が表示される。このため、この画面を見て、登録自体に失敗したと勘違いし、再び入力画面から同じ情報を再登録してしまう可能性がある。

二重サブミットは防げて、ユーザビリティの面で問題がある。



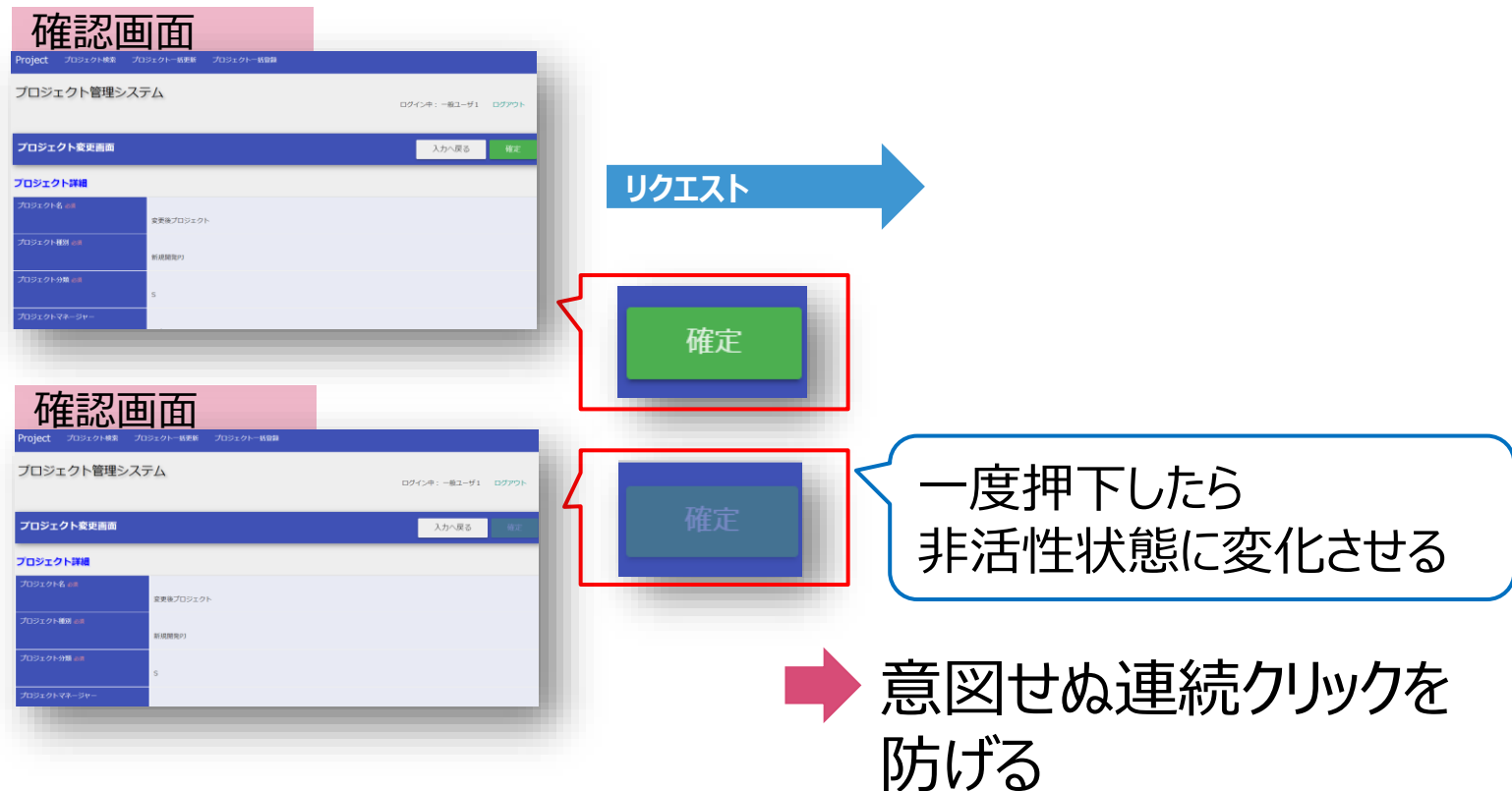
ユーザの誤操作を防ぎ、ユーザビリティを向上させるために、二重サブミットの発生そのものを防ぐ対応策も併用する。



- サブミットボタンの非活性化  
発生状況①「登録確定ボタンを連続して押下」を防止
- PRGパターンによる再送信防止  
発生状況②「登録完了ページの再読み込み」を防止

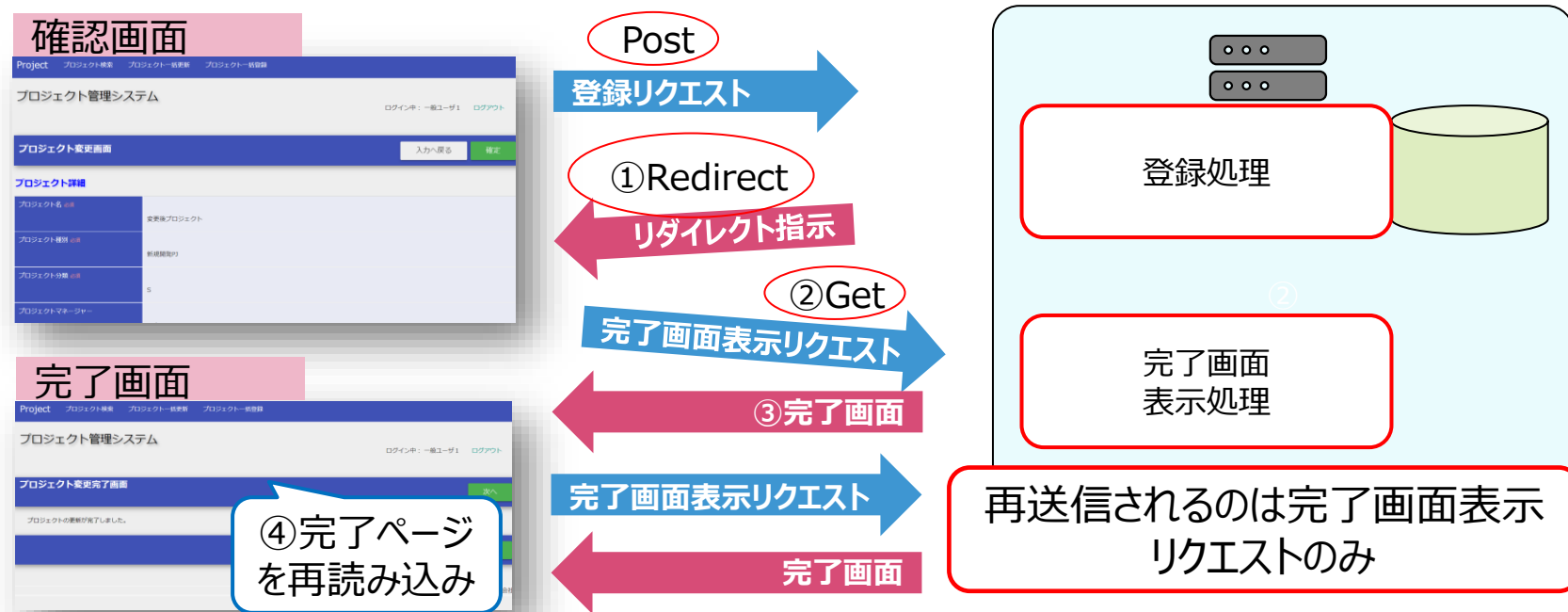
# 対応策① サブミットボタンの非活性化

誤って二重クリックしてしまわないよう、  
登録ボタン押下時にボタンを非活性化する。  
(JavaScriptによる制御)



# 対応策② PRGパターンによる再送信防止

- Post/Redirect/Get。
- 完了画面再読み込みによる二重登録を防ぐ。



- ①登録処理完了後、直接完了画面を表示するのではなく、完了画面を表示する処理へリダイレクトするようレスポンスを返します。
- ②リダイレクトのレスポンスを受け取ったブラウザは、指定されたURLへリクエストを送信します。この時、ブラウザのアドレス欄は指定された（完了画面表示処理の）URLになります。
- ③完了画面がレスポンスとして返ってきます。
- ④ページ再読み込みを行っても、ブラウザのURL欄は完了画面表示処理ですから、完了画面が再表示されるだけです。

# エラーハンドリング

---

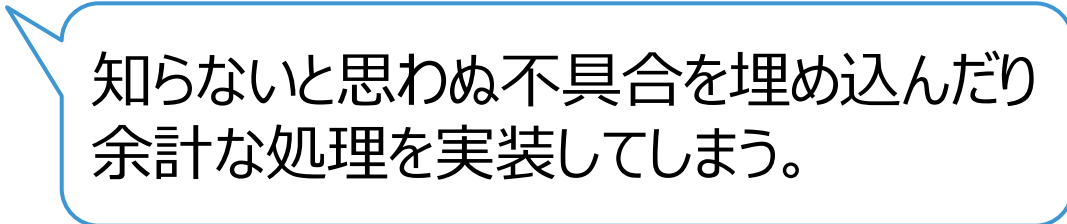
一般にWebアプリケーションケーションフレームワーク（FW）には、エラーハンドリング（例外処理）機能が備わっている。

アプリケーションに共通的なエラー処理はFWの機能で処理できることが多い。



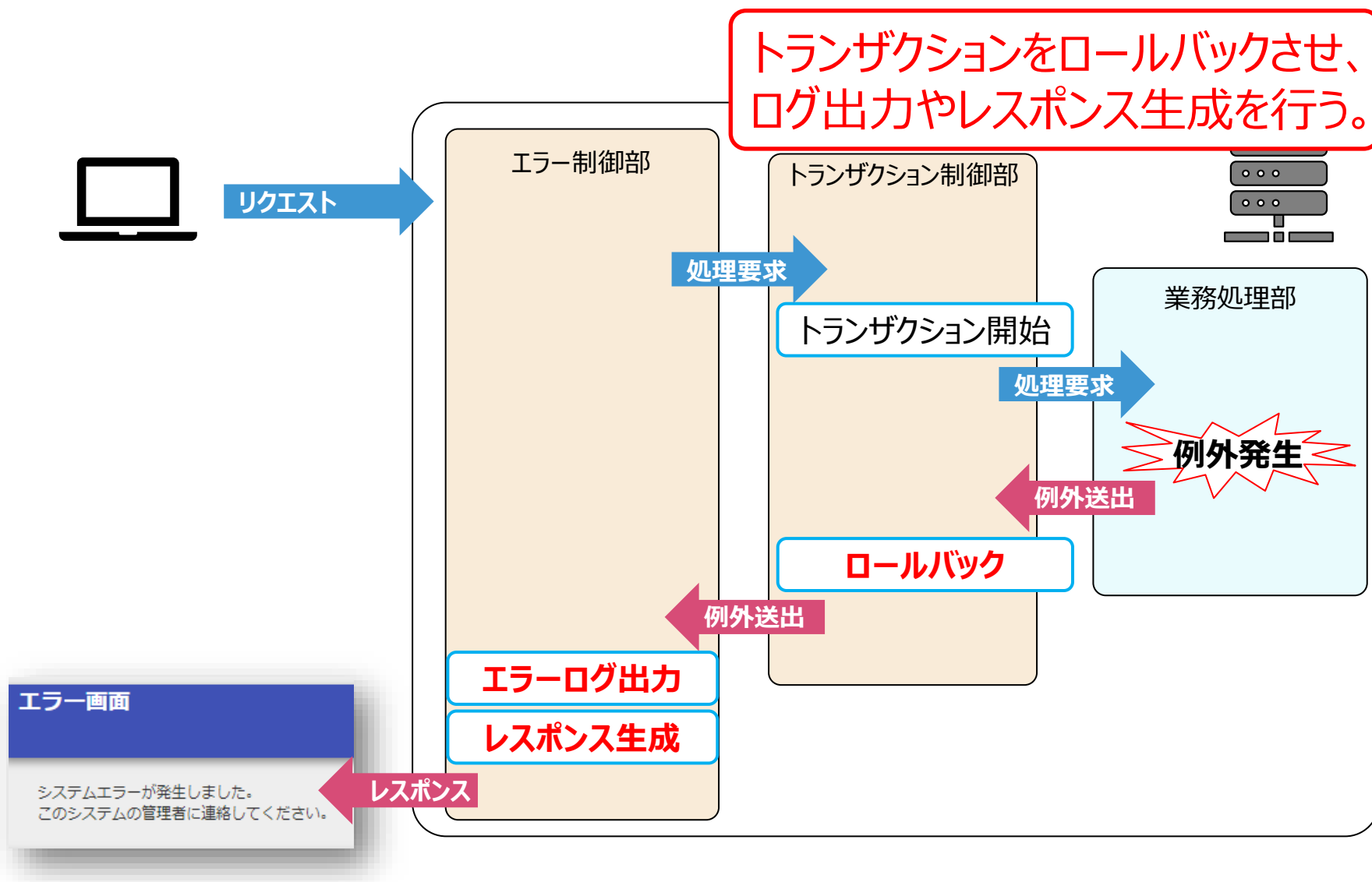
ただし

業務に合わせた処理を行うには、FWのエラーハンドリングが何をしているか知っておく必要がある。

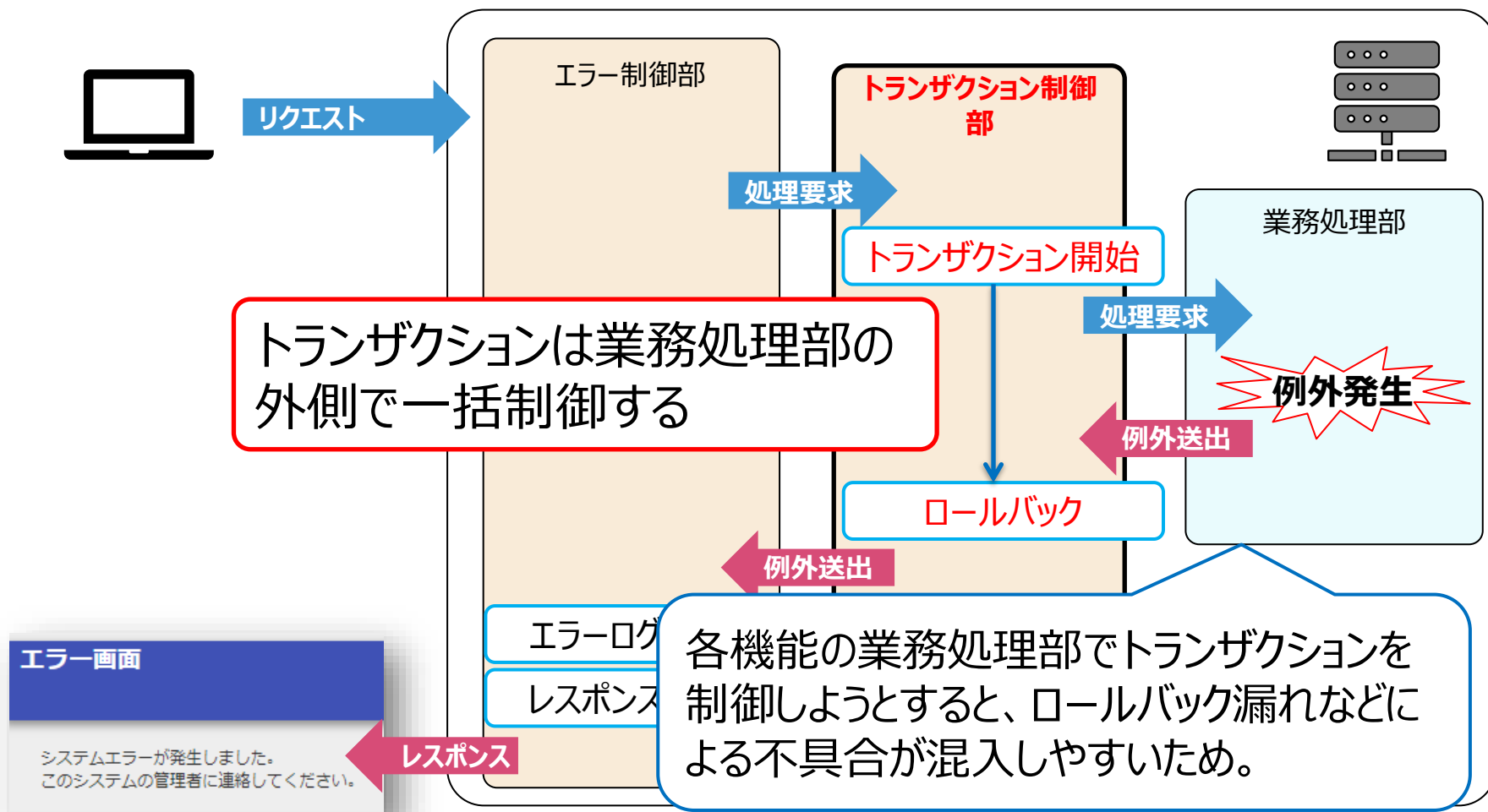


知らないと思わぬ不具合を埋め込んだり余計な処理を実装してしまう。

# エラーハンドリングの例

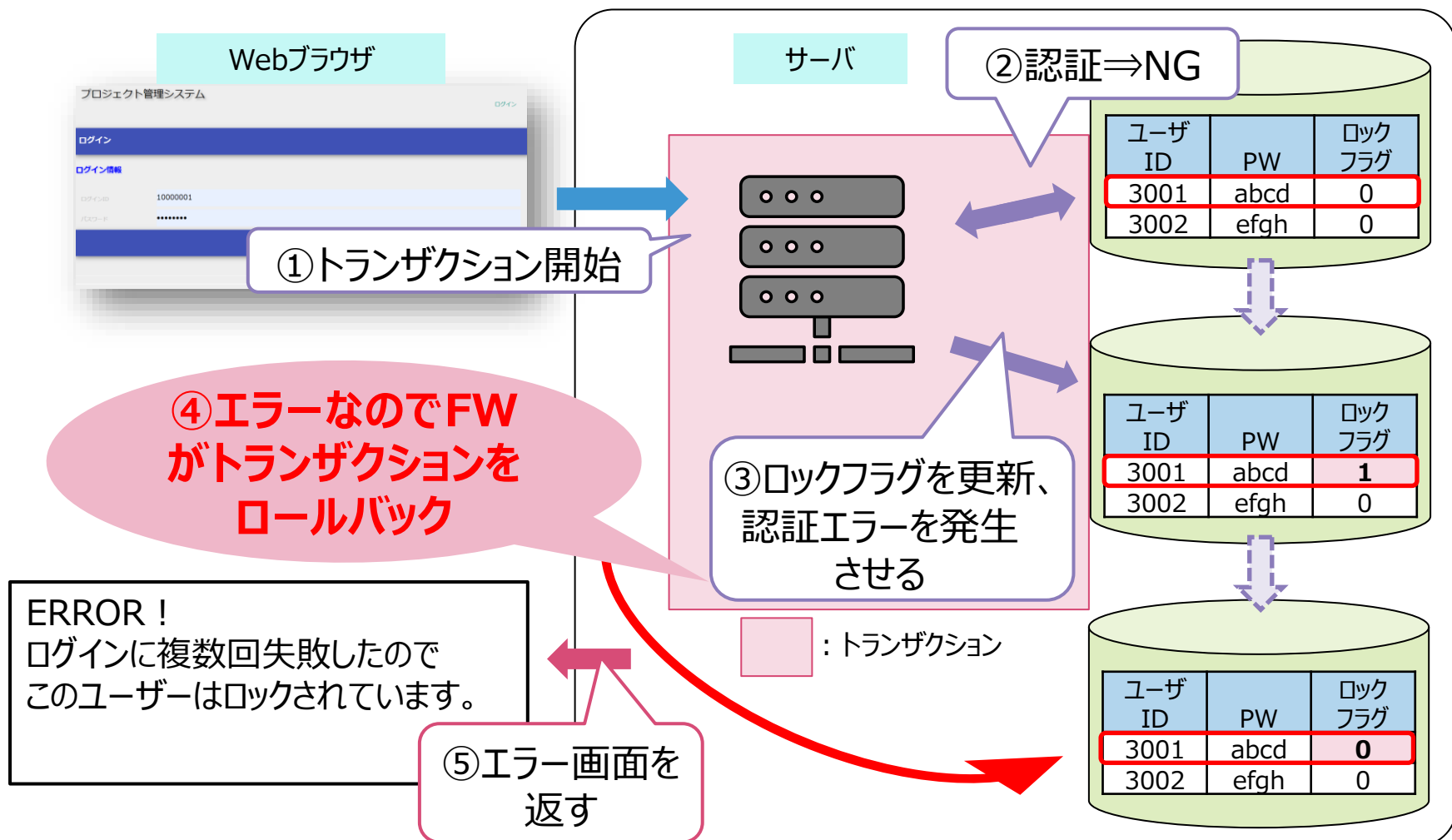


# トランザクション制御の例



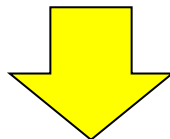


## ログイン失敗時にユーザをロックしてエラーを返す、という処理



# エラー発生時の挙動を知らないと・・・

プログラム上、「ロックフラグ」を更新しているのに、実行してみると、なぜか更新されない。



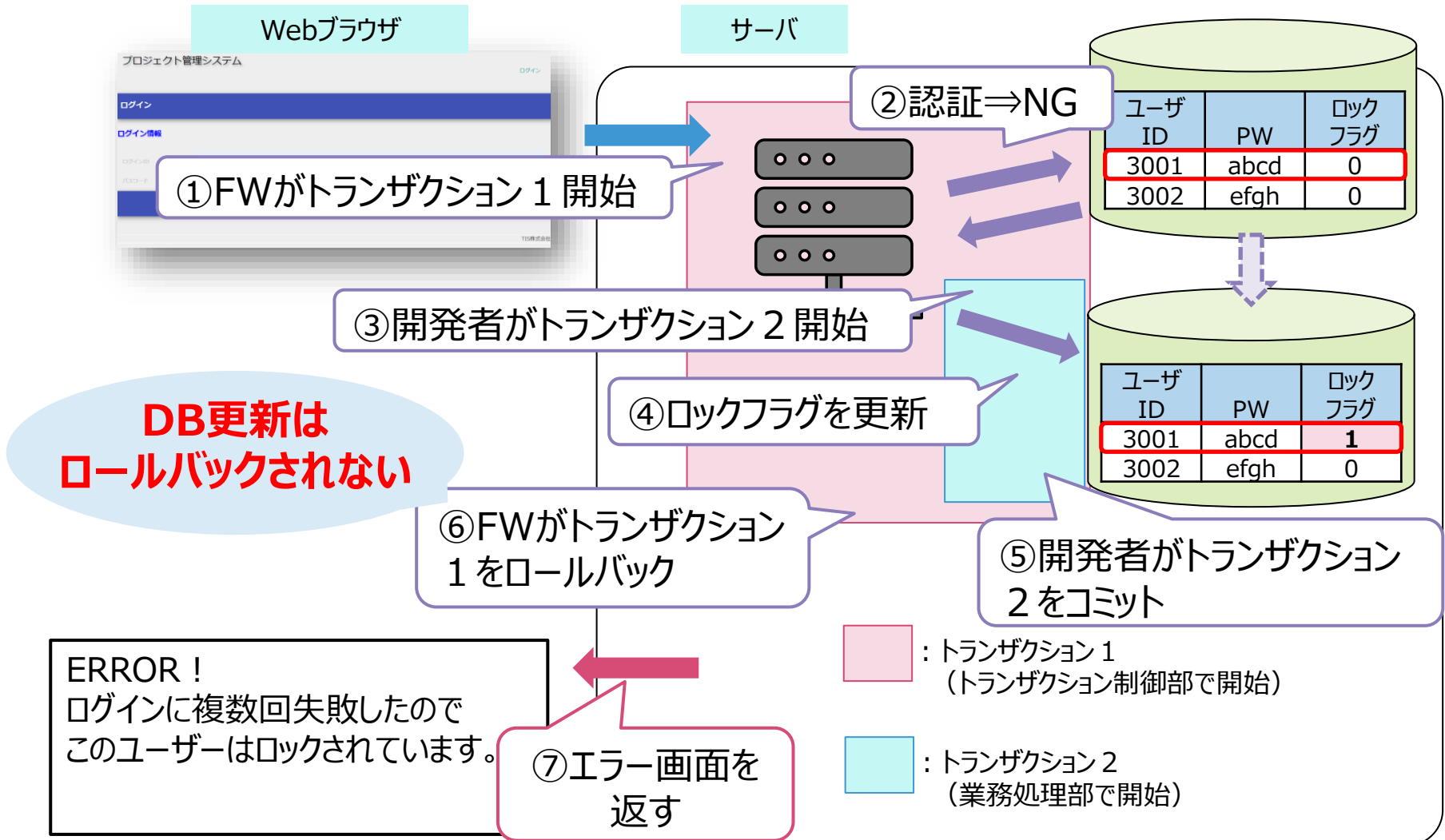
エラー発生時、フレームワークがトランザクションをロールバックさせることを知らない限り、原因不明のバグになる。

だから、フレームワークの**挙動**は、知っておく必要がある。

ちなみに、エラーを返しつつコミットしたい処理がある場合の解決策は

例外的に**別トランザクションとして実施**する。

# 別トランザクションでの処理実行



# 戻る処理

---

ヒストリバック（ブラウザの「戻る」ボタンでの遷移）とブラウザキャッシュの関係について説明します。

- ヒストリバックによる遷移
- ヒストリバックによる遷移の問題点

- ヒストリバックによる遷移
- ヒストリバックによる遷移の問題点

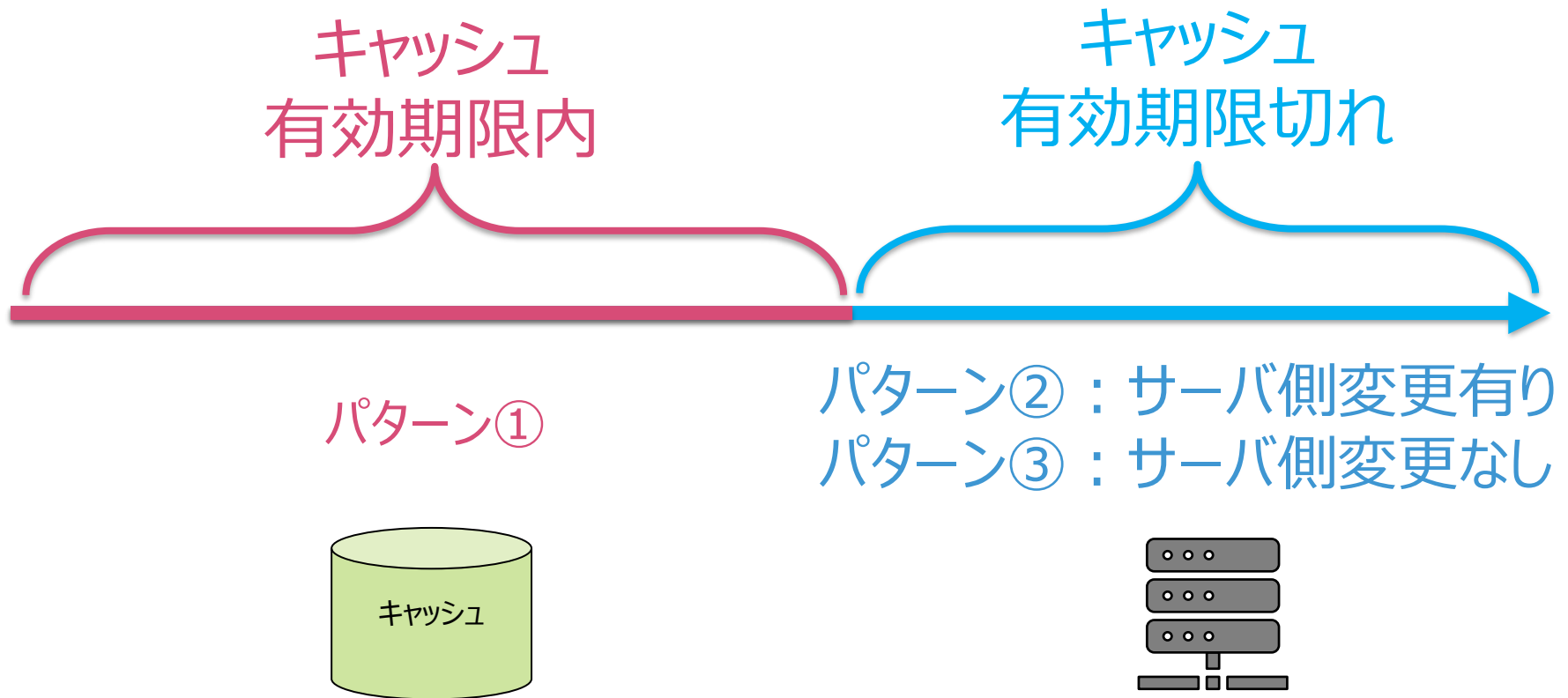
# ヒストリバックによる遷移

ヒストリバックにより前の画面が表示されるが、  
実はキャッシュ（ブラウザキャッシュ）の有効・無効によって  
動作の仕組みが異なっている。

キャッシュの有効・無効による動作の特徴と  
気を付けるべきポイントを理解しましょう。

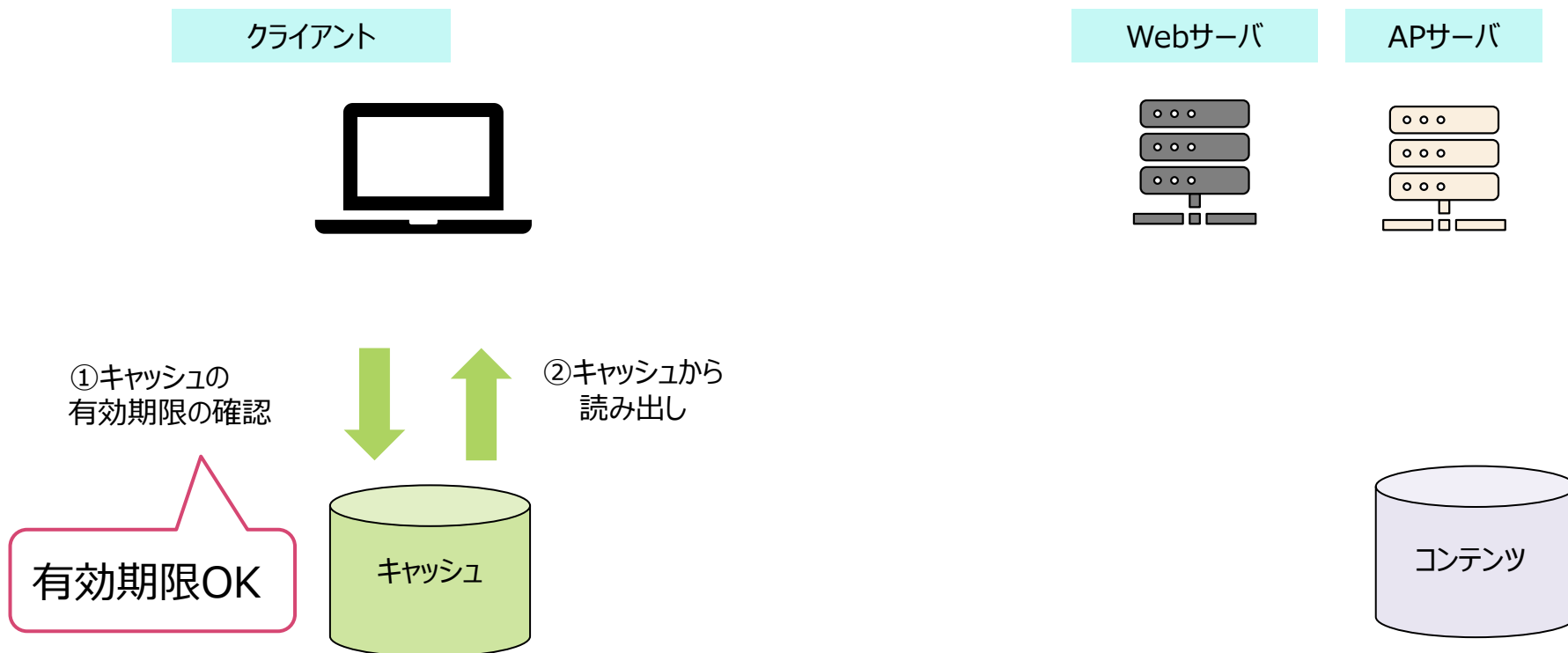


# キャッシュが有効な場合



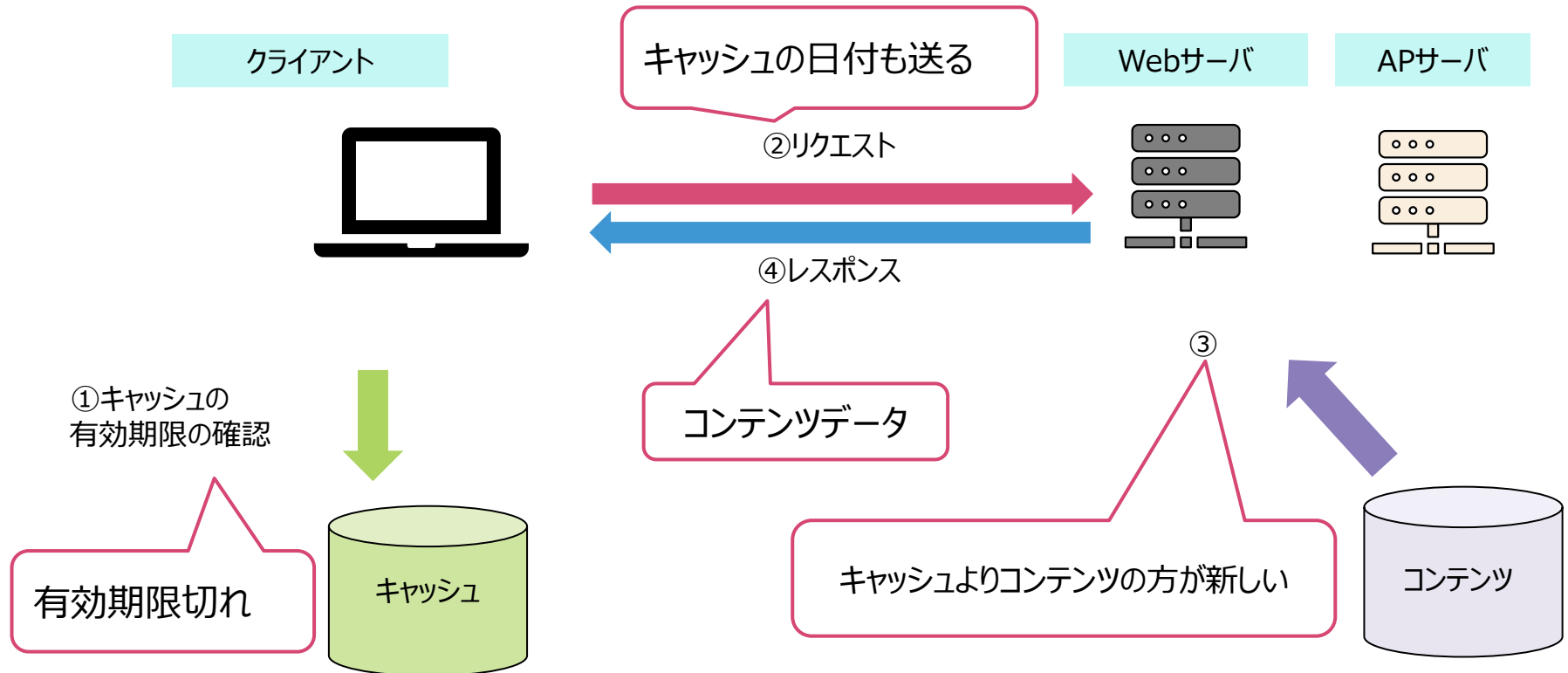
# キャッシュが有効な場合（パターン①）

キャッシュの有効期限内だった場合、リクエストは送信せず、キャッシュから読み出しを行い、ページを表示します。



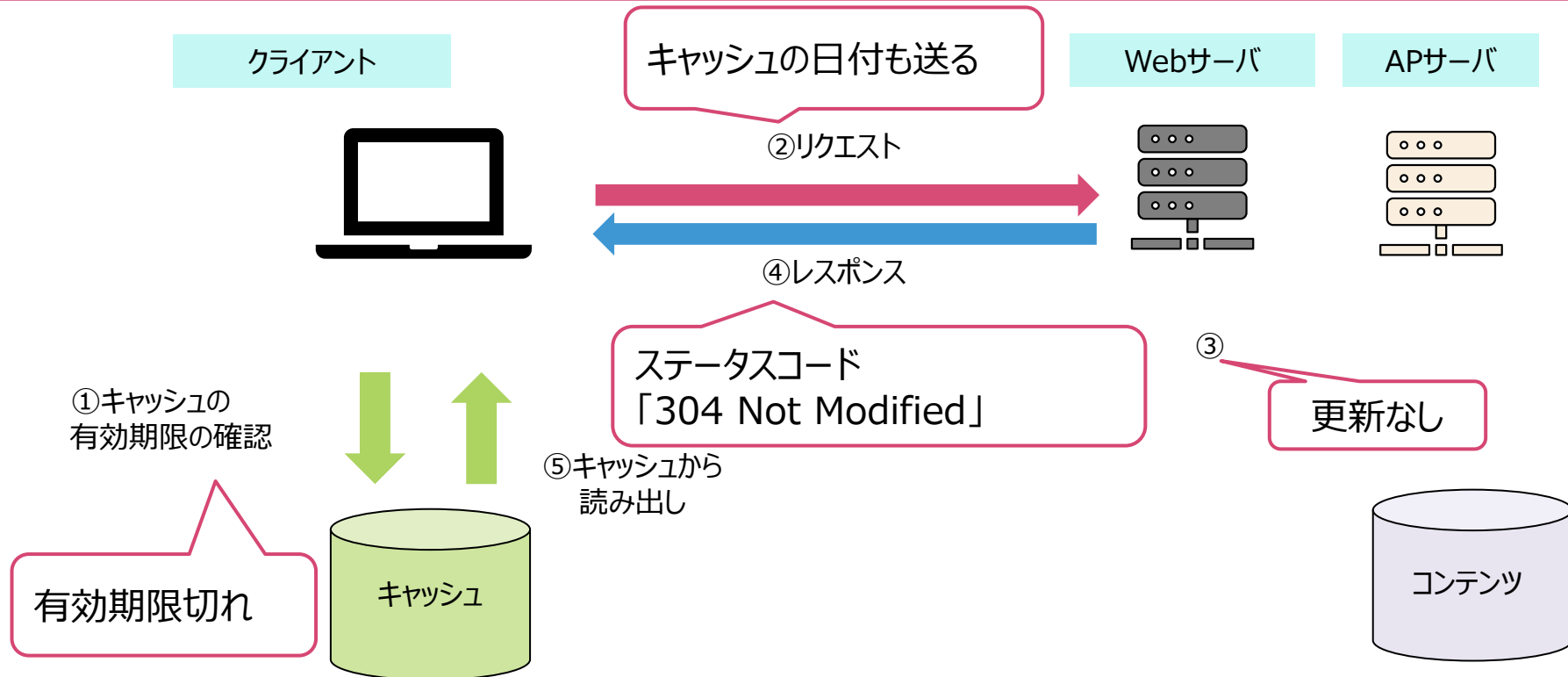
# キャッシュが有効な場合（パターン②）

- ①、② キャッシュが有効期限切れだった場合、キャッシュの日付もつけて、リクエストをサーバに送信します。
- ③ 送信されたキャッシュの日付とページの更新日付を比較し、ページの更新日時の方が新しい場合、コンテンツを読み込みます。
- ④ コンテンツデータをレスポンスとして送信します。



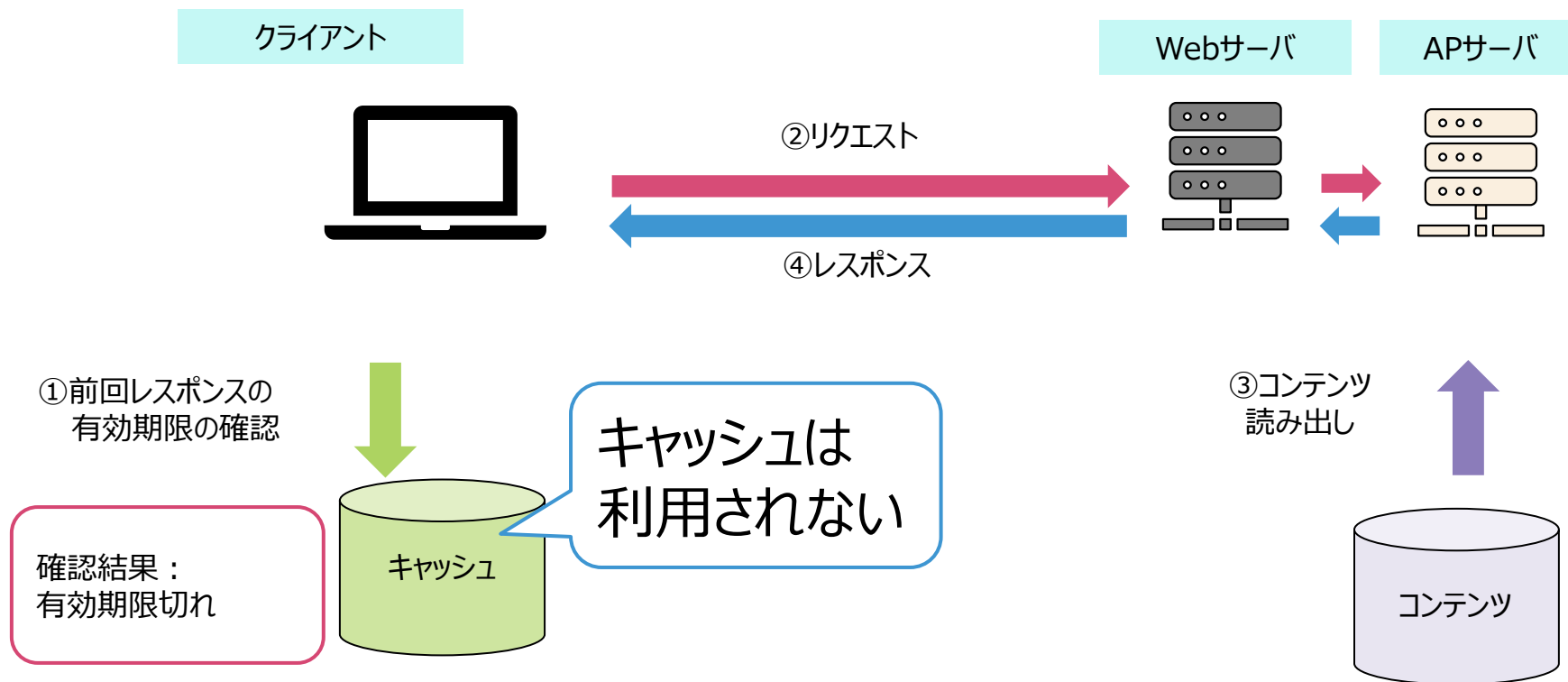
# キャッシュが有効な場合（パターン③）

- ①、② キャッシュが有効期限切れだった場合、キャッシュの日付もつけて、リクエストをサーバに送信します。
- ③、④ 送信されたキャッシュの日付とページの更新日付を比較し、変更がなければ「304 Not Modified」というレスポンスを送信します。
- ⑤ 「304 Not Modified」が返信されてきた場合、キャッシュから読み出しを行います。



# キャッシュが無効な場合

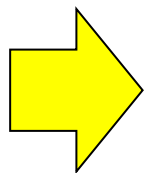
「キャッシュを常に有効期限切れにする」などの設定をすることで、実質的にキャッシュが無効にできる（詳しくは後述）。  
この場合、クライアントは常にサーバからコンテンツを取得する（下図）。  
POST送信の場合も、キャッシュは利用されない。



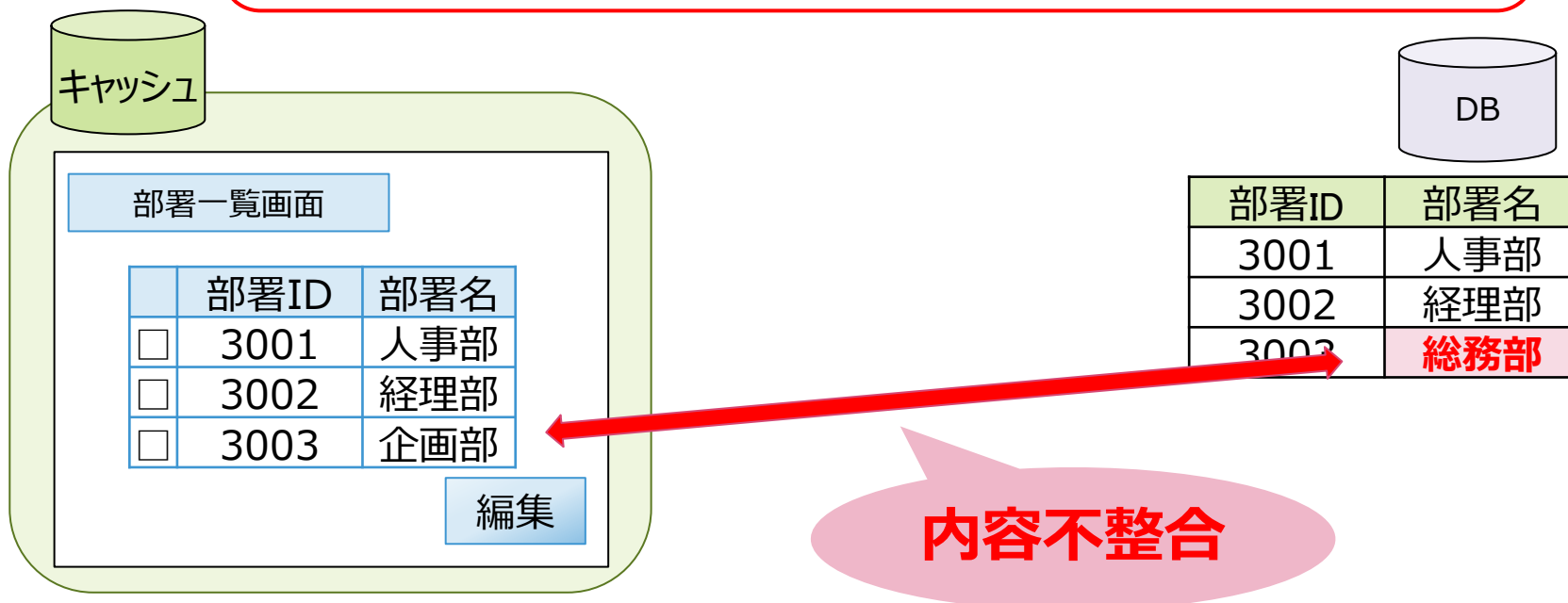
- ヒストリバックによる遷移
- ヒストリバックによる遷移の問題点と解決策
  - キャッシュが有効な場合
  - キャッシュが無効な場合

# キャッシュが有効な場合の問題点と解決策

あるユーザが参照していた情報を他のユーザが更新した場合、参照を行うユーザ側がヒストリバックで画面遷移をしたときにキャッシュされた古い情報が表示されてしまうという問題が起こる。



最新のDB状態を表示したい画面は  
キャッシュを利用しない設定にする必要がある



## ①HTTPレスポンスヘッダで制御

### キャッシュを使用しない設定例

```
Expires: Thu, 01 Jan 1970 00:00:00 GMT  
Cache-Control: no-store, no-cache, must-revalidate,  
               post-check=0, pre-check=0  
Pragma: no-cache
```

- Expiresヘッダ  
 キャッシュの有効期限を指定する。
- Cache-Controlヘッダ  
 no-store、no-cacheなどのディレクティブにより、  
 キャッシュが期限切れだった場合の制御や  
 リクエスト、レスポンスの保存可否を指定する。
- Pragmaヘッダ  
 no-cache指定の場合、リソースをキャッシュしてはならないことを示す。

Cache-Controlヘッダは環境により動作しないことがあるため、同様の効果を持つ他のヘッダも設定すること。



## ②HTMLの<meta>タグで制御

### キャッシュを使用しない設定例

```
<head>  
  <meta http-equiv="pragma" content="no-cache">  
  <meta http-equiv="cache-control" content="no-cache">  
  <meta http-equiv="expires" content="0">  
</head>
```

HTTPの仕様上はHTTPレスポンスヘッダでの制御のみで良いが、これに準拠しない古いブラウザが使われている可能性があるため、必要に応じてmetaタグを用いた制御もしておく必要がある。

キャッシュが利用されてしまうと困る場合には、  
キャッシュを制御するようにする。

- ①HTTPレスポンスヘッダで制御
- ②HTMLの<meta>タグで制御

Nablarchの場合、キャッシュを利用したくない画面のJSPに  
<noCache>タグを付与することで、上記2つの実装が実現される。

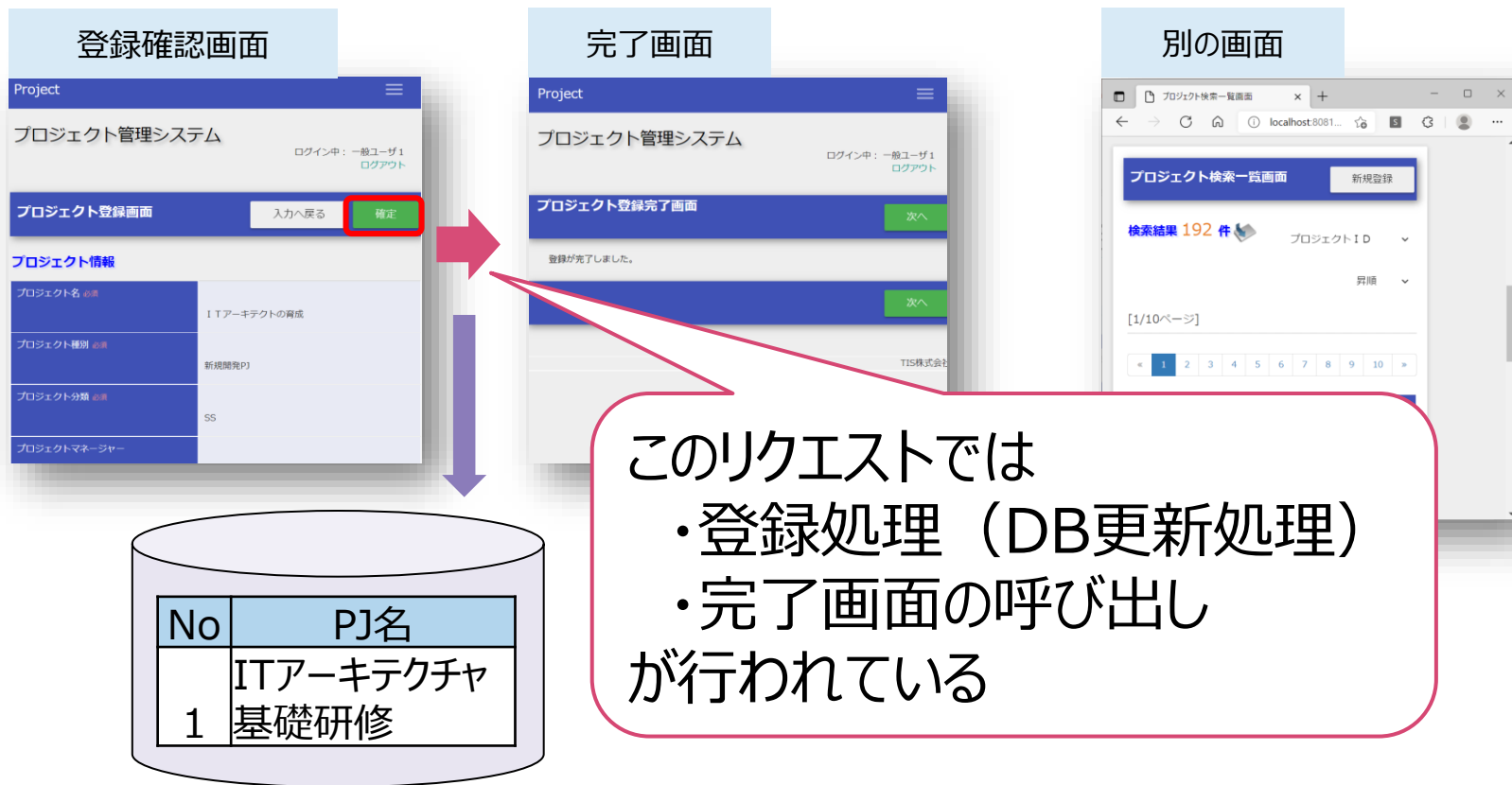
なお、ユーザ側（ブラウザ側）からキャッシュを利用しない  
設定をすることもできるが、全てのユーザに設定を統一させるのは  
困難なため、ユーザ依存にせずサーバで制御すること。

- ヒストリバックによる遷移
- ヒストリバックによる遷移の問題点と解決策
  - キャッシュが有効な場合
  - キャッシュが無効な場合

登録画面→確認画面→完了画面と遷移し、  
他の画面に遷移してからヒストリバックで完了画面に戻ると、  
完了画面を表示するためのリクエストが呼ばれる。

このリクエストが登録処理も含んでいる場合、  
ユーザが意図しない登録処理の再実行（二重サブミット）が  
発生してしまう。

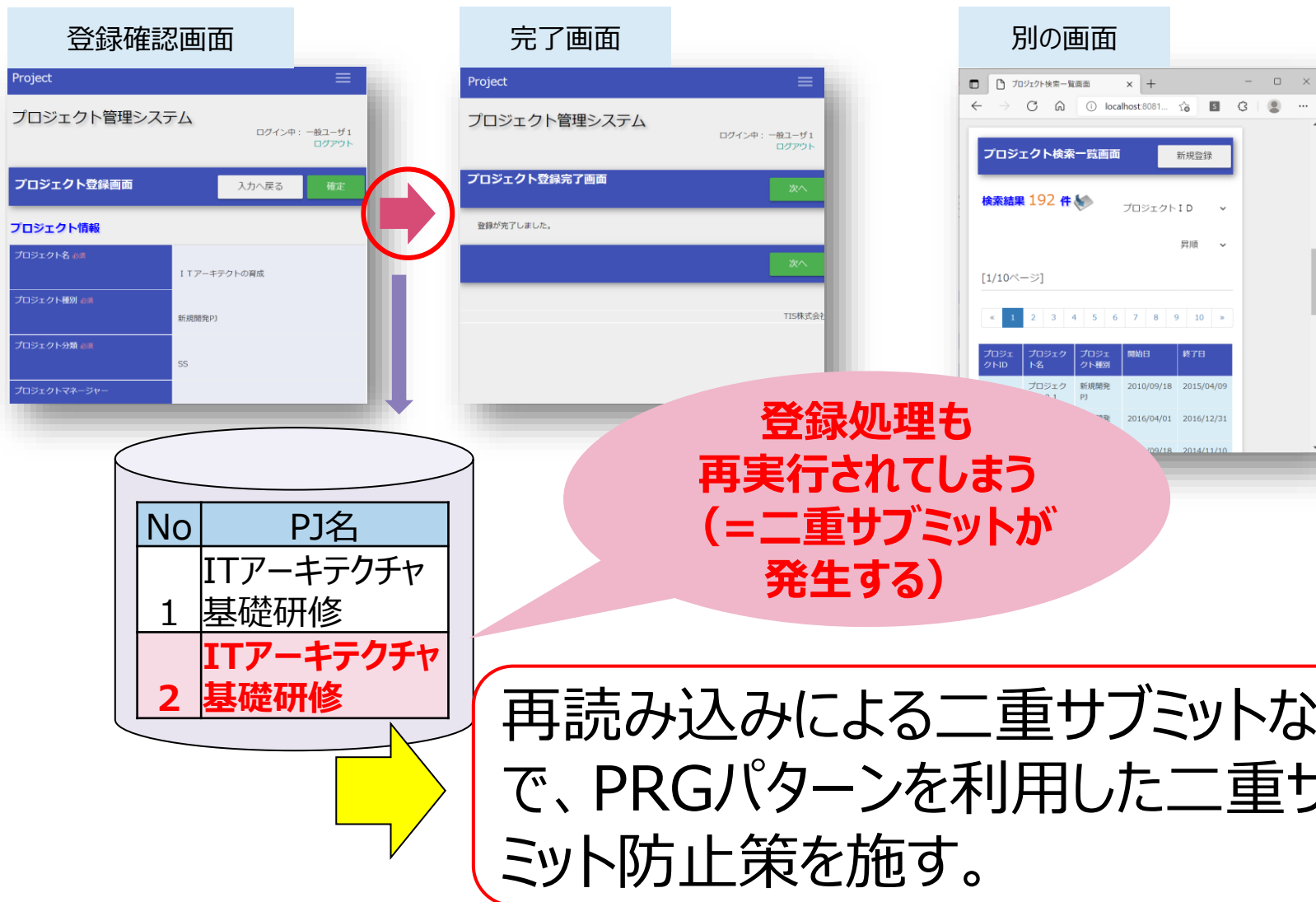
# キャッシュが無効な場合の問題点① 2/4



# キャッシュが無効な場合の問題点① 3/4

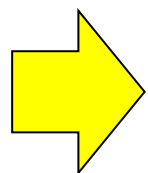


# キャッシュが無効な場合の問題点① 4/4



# キャッシュが無効な場合の問題点②

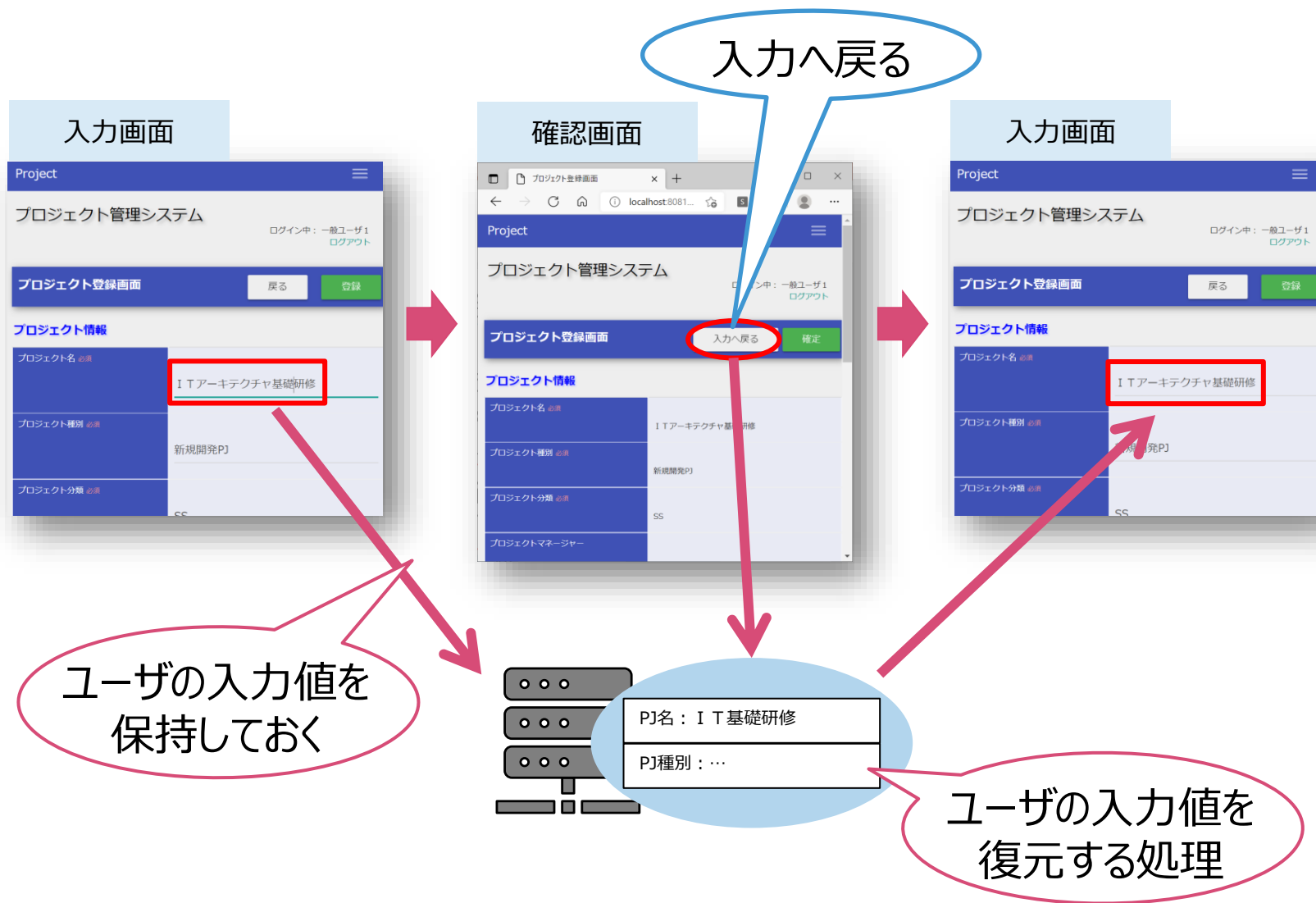
確認画面でヒストリバックを行うと、登録画面を初期表示するリクエストをブラウザが再送信するため、ユーザの入力値が消えてしまう。



ユーザの入力値を復元する設定を加えた  
**戻る処理**を作成する必要がある







# ファイルアップロード・ダウンロード

---

一言でファイルのアップロードやダウンロードといっても、やらなければならない処理はいくつもあります。

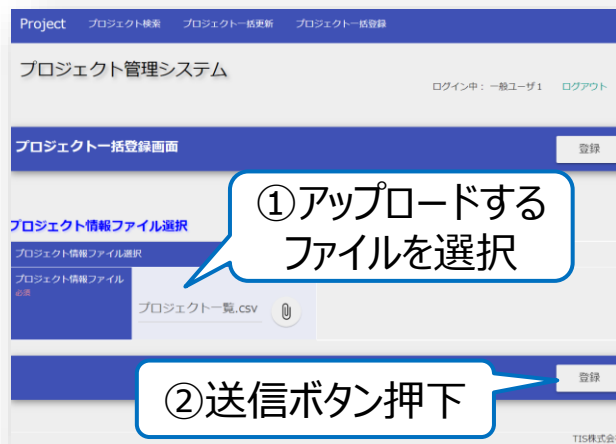
それらを説明します。

- ファイルアップロード
- ファイルダウンロード

- ファイルアップロード
- ファイルダウンロード

# ファイルアップロードの流れ

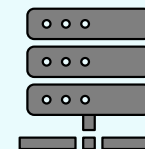
クライアント



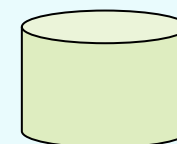
③リクエストボディに  
ファイルデータを付加

リクエスト

サーバ



業務処理



④ファイルに対する処理  
・ データベースへの登録  
・ ファイルの保存 など

レスポンス

⑤処理結果

# 実際に送信されるアップロードリクエストの例

## ヘッダ部

Host: localhost:9080  
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:61.0) Gecko/20100101 Firefox/61.0  
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8  
Accept-Language: ja,en-US;q=0.7,en;q=0.3  
Accept-Encoding: gzip, deflate  
Referer: http://localhost:9080/action/project/create  
Content-Type: multipart/form-data; boundary=-----4827543632391  
Content-Length: 448  
Cookie: JSESSIONID=9AC1D9B8662CECA52254BE9507C37932; NABLARCH\_SID=e1ed57b8-adf7-4bae-bd0c-92c412045c01  
Connection: keep-alive  
Upgrade-Insecure-Requests: 1

## ボディ部

-----4827543632391  
Content-Disposition: form-data; name="comment"  
  
hello world!  
-----4827543632391  
Content-Disposition: form-data; name="uploadFile"; filename="登録データ.csv"  
Content-Type: application/vnd.ms-excel

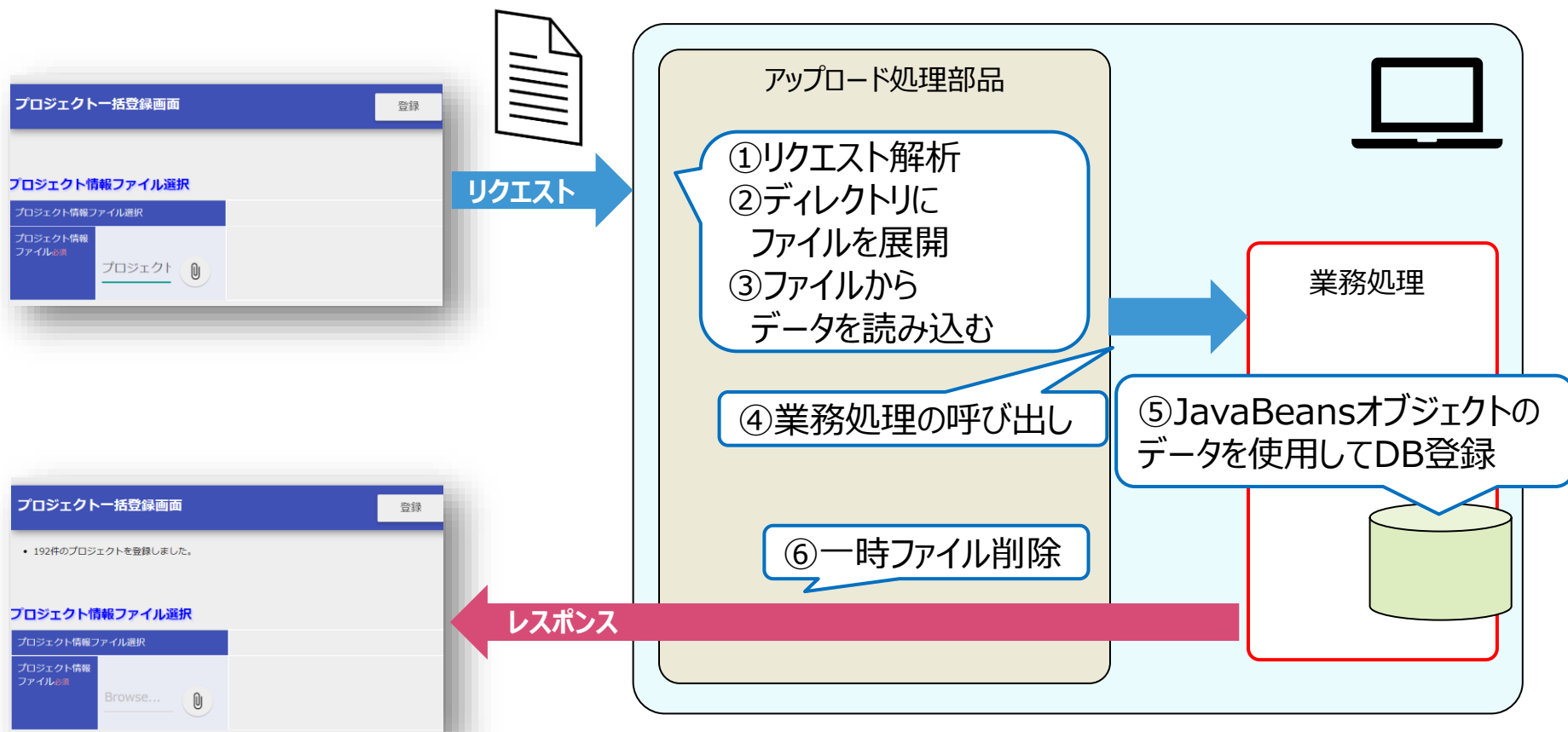
"projectName","projectType","projectClass"  
"プロジェクト 1","development","a"  
"プロジェクト 2","development","a"

-----4827543632391--

送信ファイルの中身

# Nablarchでの処理例（1/5）

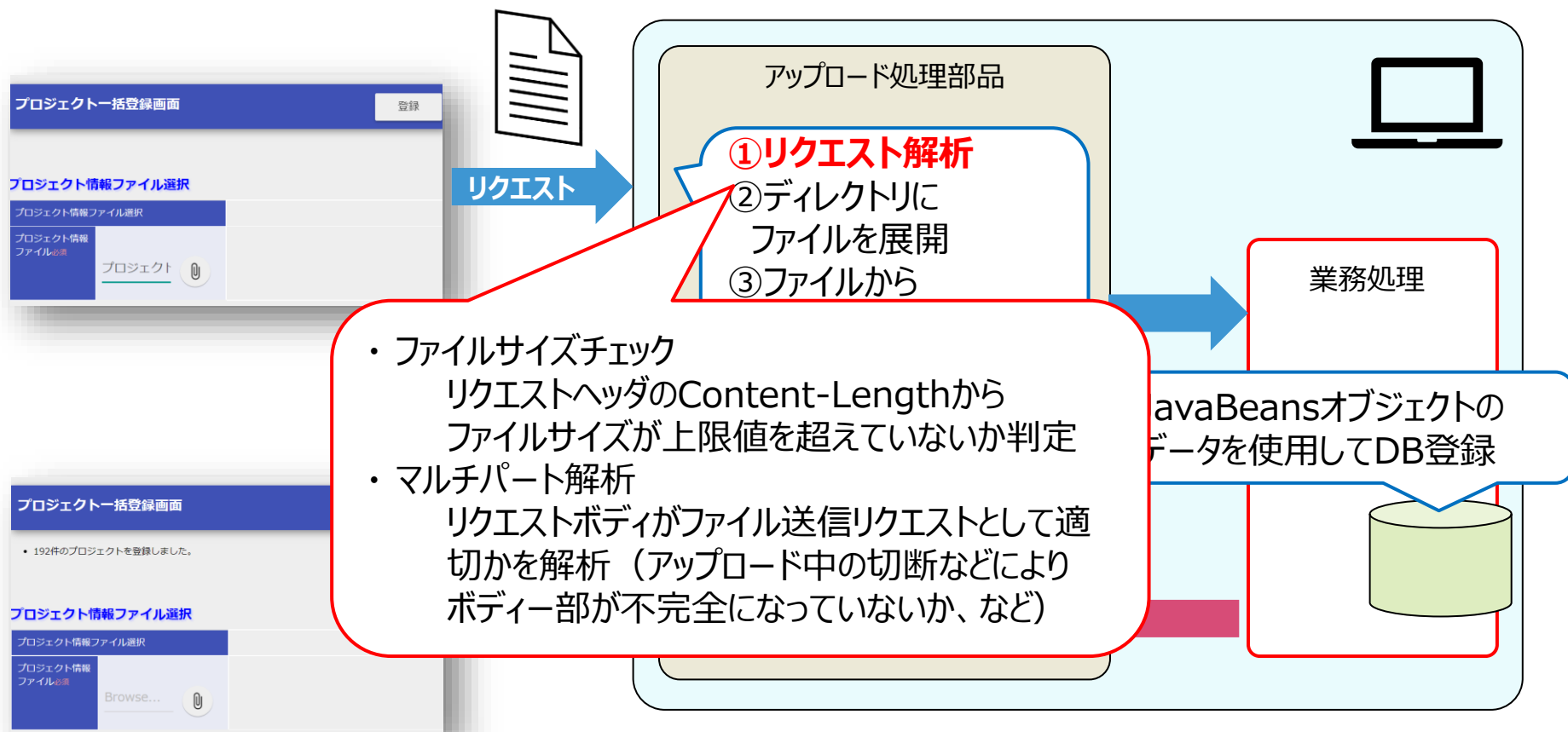
## CSVファイルをアップロードしてデータベースに登録



次ページから各処理を説明します

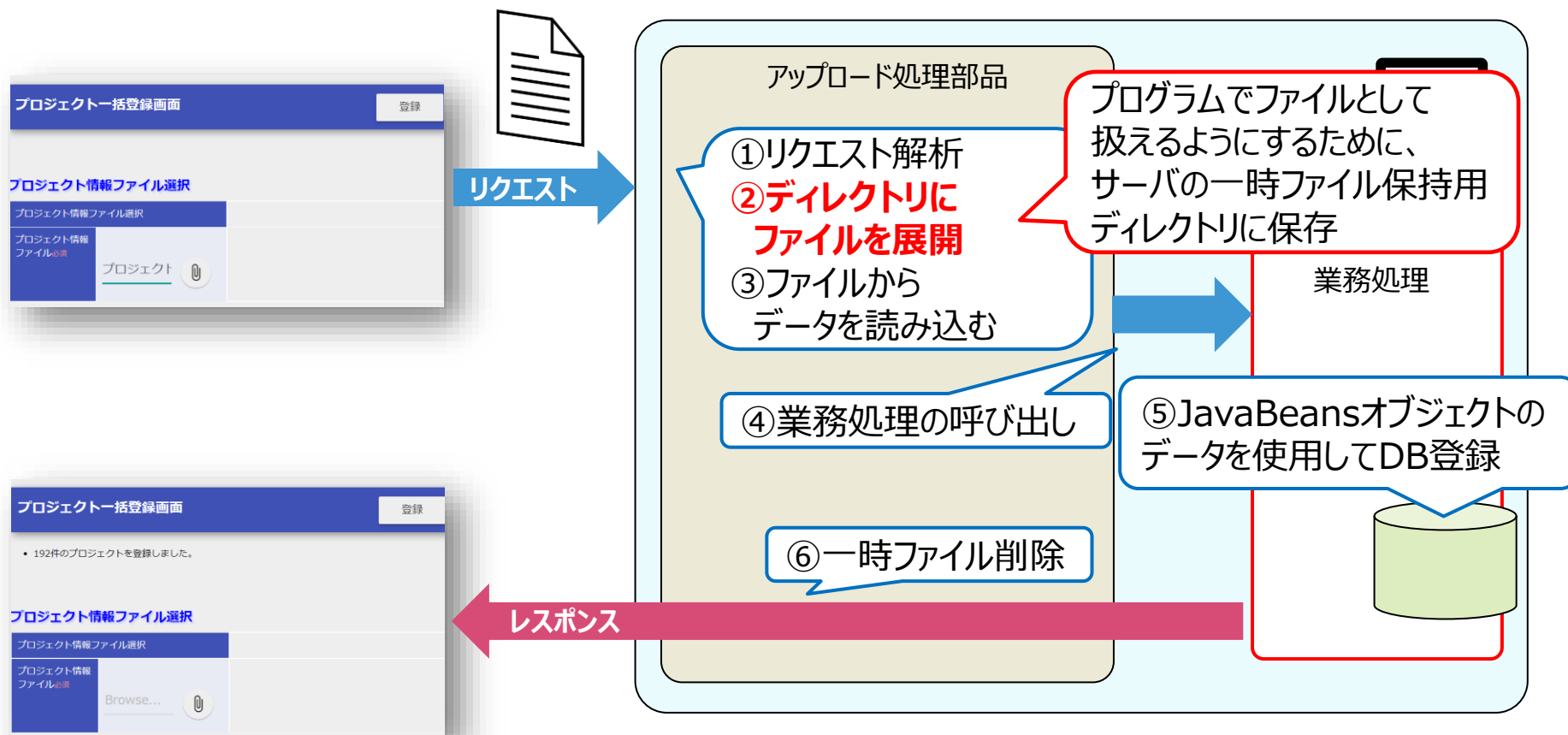
# Nablarchでの処理例 (2/5)

## CSVファイルをアップロードしてデータベースに登録



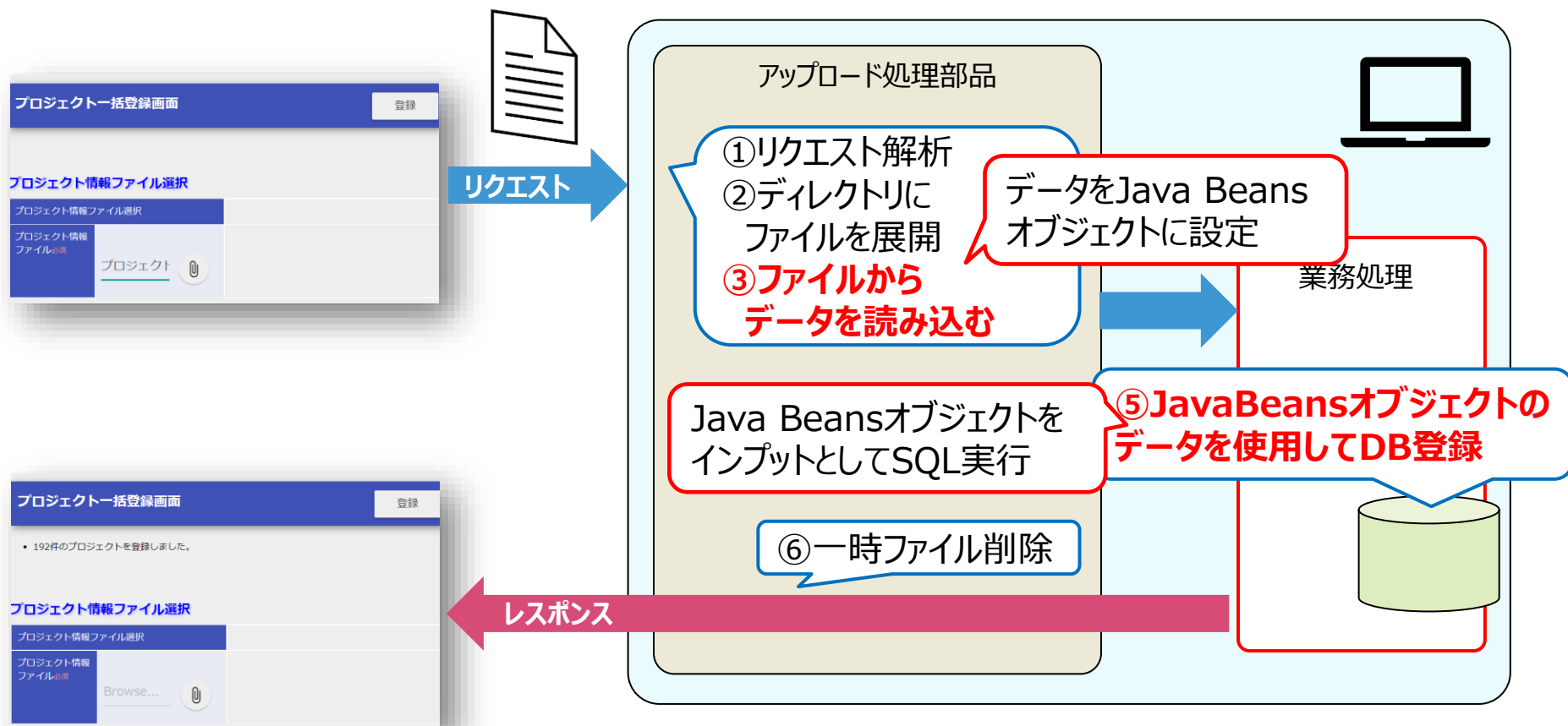


# CSVファイルをアップロードしてデータベースに登録



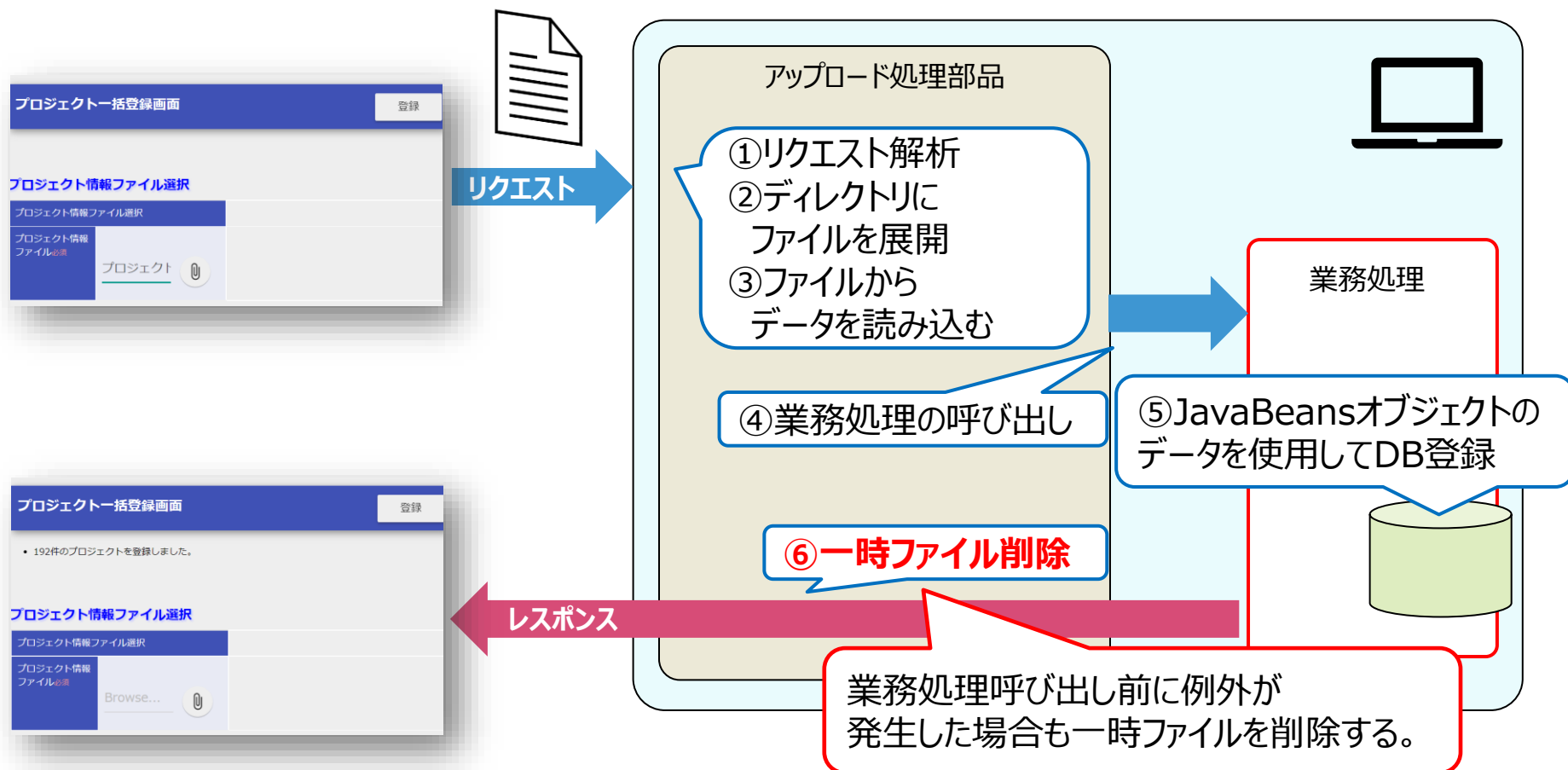
# Nablarchでの処理例（4/5）

## CSVファイルをアップロードしてデータベースに登録



# Nablarchでの処理例（5/5）

## CSVファイルをアップロードしてデータベースに登録



# これだけでは十分ではありません

- タイムアウトする程、ファイルサイズが大きい
  - ファイルにエラーがあった時の対応
- など、考慮すべき点はほかにもあります。

※以下の投稿記事も参考になります。

Qiita「至高のファイルアップロード」

投稿者：@kawasima

<https://qiita.com/kawasima/items/f80bc54efb12d5509c0b>

- ファイルアップロード
- ファイルダウンロード
  - ファイルダウンロードの流れ
  - ファイルダウンロード処理で注意すべき点

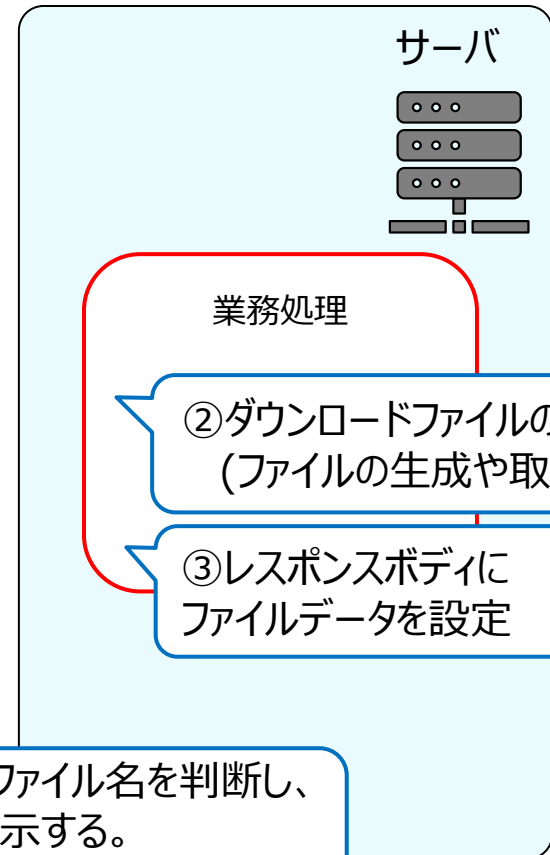
# ファイルダウンロード

クライアント



①ダウンロードボタン押下

リクエスト



レスポンス

④レスポンスヘッダからファイル形式やファイル名を判断し、ブラウザがダウンロードダイアログを表示する。  
※ブラウザにより挙動は異なる

# ファイルダウンロード時に送信されるレスポンスの例

## ヘッダ部

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Set-Cookie: NABLARCH_SID=8d920de2-e2c4-4b4b-8f18-dc6e49ba5745; Path=/; HttpOnly
X-Frame-Options: SAMEORIGIN
X-Content-Type-Options: nosniff
Content-Disposition: attachment;
filename*=UTF-8"%E3%83%95%E3%82%A1%E3%82%A4%E3%83%AB.csv";
filename="?UTF-8?B?44OV44Kh44Kk44OrLmNzdg==?="
X-XSS-Protection: 1; mode=block
Content-Type: text/csv
Transfer-Encoding: chunked
Date: Mon, 27 Aug 2018 03:32:11 GMT
```

Content-Typeでボディ部の形式を指定。  
(ブラウザ側ではContent-Typeでボディ部の  
ファイル形式を判断している)

【補足】  
通常の画面遷移(Webページの表示)では、  
Content-Typeは"text/html"。

## ボディ部

```
"projectName","projectType","projectClass"
"プロジェクト 1","development","a"
"プロジェクト 2","development","a"
```

受信ファイルの中身

- ファイルアップロード
- ファイルダウンロード
  - ファイルダウンロードの流れ
  - ファイルダウンロード処理で注意すべき点



日本語(全角文字)を含むファイル名はエンコードしないとダウンロード時に文字化けしてしまう。

➡ サーバ側でレスポンスヘッダのContent-Dispositionに「filename\*」を設定する (RFC6266で規定)

## 記述例 (レスポンスヘッダ抜粋)

Content-Disposition:  
attachment;  
filename="研修資料";

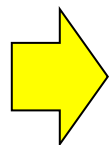
「filename\*」に対応していない古いブラウザを考慮するなら、「filename」も設定。

**filename\*=utf-8' %E7%A0%94%E4%BF%AE%E8%B3%87%E6%96%99**

「filename\*」に対応しているブラウザでは「filename」部が無視される。

# ダウンロード時の画面遷移

ファイルダウンロード時、同時に画面遷移はしない。  
リクエストに対するレスポンスとして受け取れるのは、ダウンロード対象のファイルまたはWebページ(html)のどちらかであるため。



ファイルダウンロードと画面遷移を両方行わせたい場合、  
そのための仕組みを実装する必要がある。

- 認証・認可
- 入力値のバリデーション
- 二重サブミット対策
- エラーハンドリング
- 戻る遷移
- ファイルアップロード・ダウンロード

## Web全般の基礎知識を習得

Webを支える技術  
技術評論社  
山本陽平 著  
ISBN 978-4-7741-4204-3



<https://gihyo.jp/book/2010/978-4-7741-4204-3>  
2021年11月25日17時の最新情報を取得

ITで、社会の願い叶えよう。



**TIS INTEC**  
Group