EDI 用途における Web API 利用ガイドライン

V1.0.0

インターネット EDI 普及推進協議会
Japan internet EDI Association (JiEDIA)

目　次

1.要旨

1.1.はじめに

NTT 東日本株式会社ならびに NTT 西日本株式会社は、2024 年から 2025 年にかけて従来の固定電話網（PSTN：Public Switched Telephone Network）を IP 網（Internet Protocol）へ段階的に移行する取り組みを進めてきた。

このネットワーク基盤の刷新は、音声通話サービスのみならず、多様なデジタル通信サービスに広範な影響を及ぼすことになった。特に、企業間電子データ交換（EDI：Electronic Data Interchange）の分野においては、その安定性と信頼性から長年にわたり広範に利用されてきた「INS ネットディジタル通信モード（ISDN）」が IP 網への移行に伴いサービス提供が終了するため、代替手段への移行が急務となっていた。

こうした背景を踏まえ、多くの企業や業界団体が新たな EDI 通信基盤としてインターネット EDI を選択し、移行が完了しつつある。インターネット EDI は、IP 網の速度や柔軟性、コスト効率性を活用し、従来の固定電話網に比べてより多様な通信プロトコルやセキュリティ技術の適用が可能となるため、EDI の主流となりつつある。

インターネット EDI 普及推進協議会（JiEDIA）では「通信回線の EDI 利用」という視点から移行期における課題を整理、検討し、IP 網に対応したインターネット EDI への安全な移行推進を目的としてガイドラインの策定や技術支援、情報提供を行っている。

1.2.ガイドライン作成の背景と目的

近年のクラウドサービスの急速な発展と普及に伴い、企業内外の多種多様なシステムやアプリケーション間で、リアルタイムかつ効率的にデータを連携させるニーズが急速に高まっている。従来、企業間におけるデータ連携の手段としては、標準化されたフォーマットと通信プロトコルを用いた EDI が広く利用されてきたが、システムの構築や運用に比較的大きなコストがかかることや、柔軟性に乏しいことなどが課題として指摘されてきた。

こうした状況を背景に、近年ではインターネット技術を基盤とする Web API を用いたデータ連携が注目されている。Web API はリアルタイム性の高い連携やより柔軟なデータ形式での情報交換を可能とする一方で、明確な仕様が定まっておらず、各企業が個別に設計・運用を行っているのが現状であり、互換性や再利用性の面での課題が顕在化しつつある。かつて、簡易的な企業間データ連携手段として利用が拡大した Web-EDI においても、導入の手軽さから各企業が独自仕様でシステムを構築した結果クライアント側の対応負荷が増大し、業務効率の低下を招くといった問題が生じた。

このような過去の経験を踏まえ、インターネット EDI 普及推進協議会（JiEDIA）では、API を活用したデータ交換の健全な普及・発展を目指し、一定の基準を設けるべきとの考えのもと、「EDI 用途における Web API 利用ガイドライン」を策定する。本ガイドラインは、API の開発および運用に関する最低限必要な指針を示し、企業間データ連携の効率化と安定的な運用を支援するものである。

1.3. ガイドライン化の範囲

本ガイドラインは、EDI（ファイル交換型）の枠組みにおいて、Web API を補完的に活用するケースを想定している。そのうえで、インターネット EDI において広く用いられている通信プロトコルと同等レベルの運用形態を前提とし、Web API を企業間データ交換に適用する際の、汎用的かつ実践的な技術的・運用的指針を体系的に整理することを目的とする。なお、本ガイドラインは特定業界の業務プロセスや上位メッセージ仕様の標準化を目的とするものではなく、異なる業種・業態間でも適用可能な、Web API を EDI 用途で安全かつ信頼性高く活用するための基盤構築を目的とした枠組みを提示するものである。

1.4. 本ガイドラインが対象とする組織と想定する読者

本ガイドラインは、企業内外で異なるシステムやアプリケーション間のデータ連携に API を利用する、あらゆる組織を対象とする。具体的には、以下のような組織が想定される。

事業会社	サプライチェーン管理、顧客関係管理、マーケティング、IoT データ活用など、様々な業務で API を利用する企業。
システム開発会社	企業向けに API を設計、開発する SIer やソフトウェアベンダー。
サービス事業者	データ連携用途としての API を提供し、様々なシステムとの連携を可能にする事業者。
業界団体	業界内でのデータ連携標準を策定するために、API 利用ガイドラインを必要とする団体。
政府機関・自治体	行政サービスのデジタル化や、企業・市民とのデータ連携のために API を利用する組織。
システム開発者	API の設計、開発、実装に携わるエンジニア。
システム管理者	API の運用、保守、セキュリティ管理を担当する担当者。
システム企画者	API を利用したシステム連携を企画する担当者。
業務担当者	API を介して提供されるデータを業務で利用するユーザー。

本ガイドラインが、これらの組織や読者にとって API を効果的に活用し、安全かつ信頼性の高いデータ連携を実現するための一助となれば幸いである。

1.5. 本ガイドラインの利用上の注意事項

本ガイドラインに記載の内容は、Web API を EDI 用途で利用する際の一般的な考え方および設計上の留意点を整理したものである。実際のシステム構成や要件は、利用する通信環境・取引形態・運用体制などによって異なるため、各社の実情に応じて適宜修正・補足して活用することが望ましい。

技術情報については、2025 年 12 月時点の技術動向、通信仕様、セキュリティ標準に基づいて作成している。今後の技術動向や規格改定、関連ソフトウェアの更新により、内容が変更となる可能性がある点に留意し、標準化団体等から発信される最新情報を確認のうえ、必要に応じて見直しを検討されたい。特にセキュリティ関連情報は更新頻度が高いため、常に最新の情報を参照することが重要である。

また、第 4 章に掲載しているサンプルコードは、概念や処理の流れを理解しやすくするための例示であり、実際のシステム環境や要件によって実装内容は異なる場合がある。

特に、サンプルコードでは視認性の観点から設定値などをソースコード内に直接記載しているが、実際の運用においてはパラメータファイルや環境設定ファイルなどに外部化し、適切に管理することを推奨する。したがって、掲載コードはあくまでも参考例として取り扱われたい。

2.Web API とは

2.1.Web API 活用にあたって

Web API を EDI 用途で活用するにあたり、はじめに注意すべき点を整理する。
Web API は、リアルタイムかつ直接的なデータ連携を前提に設計されており、システム間の即時通信や疎結合な連携に適した技術である。一方で、EDI はファイルベースのバッチ処理を中心とした運用が一般的であり、特に企業間取引においては長年の運用実績を持つ通信プロトコルを活用することで、高い信頼性・安定性・セキュリティが確保されてきた経緯がある。

このような背景から、現在の業務要件や既存の運用体制を維持したまま EDI を行う場合には、Web API ではなく従来型の通信プロトコルを継続して利用する方が、現実的かつ合理的であるケースが多い。実際、EDI を支える多くのシステムでは、ファイル形式やフォーマット、通信時間帯、送受信順序、到達確認の方式などに関して厳密なルールが定義されており、これらを Web API に適合させるためには、追加の設計や実装対応が不可欠となる。
さらに、Web API は即時性を前提としていることから、エラー発生時のリトライ処理、セッション管理、通信結果の検証といった運用上の設計要素も従来以上に複雑化する傾向がある。したがって、EDI と比較すると、Web API を活用して EDI を実現するためには設計・運用両面において考慮すべき事項が多く、導入・維持の負担が増す可能性がある点に留意が必要である。
しかしながら、昨今のビジネス環境は急速に変化しており、取引のスピードや柔軟性が重視される傾向にある。こうした中、バッチ処理に適した EDI に加えて、リアルタイム性を備えたデータ連携のニーズも徐々に高まりつつあり、将来的には業務システムがリアルタイム処理へとシフトしていくことも想定される。

このような状況を踏まえ、EDI を補完する形で Web API を活用する過渡的利用モデルを想定し、Web API を EDI の通信手段として活用する場合の考慮点や、具体的な実装モデルについて述べていく。

2.2.Web API の特徴

Web API（Web Application Programming Interface）とは、異なるソフトウェア間で通信やデータのやり取りを可能にするインターフェースである。通常、Web API は HTTP（Hypertext Transfer Protocol）を利用して通信を行い、リクエストとレスポンスの形式でデータを交換する。リクエストには HTTP メソッドが使用され、レスポンスは一般的に JSON（JavaScript Object Notation）や XML（eXtensible Markup Language）といった軽量なデータ形式が用いられる。これにより、インターネットを介したシステム連携が容易になるといった特徴がある。また、認証方式には API キーや OAuth、JWT（JSON Web Token）などが使用される。

2.3. 技術的要素

一般的な Web API で使用される技術的要素を以下に示す。

①HTTP/HTTPS プロトコルの使用

多くの Web API は、HTTP または HTTPS プロトコルを使用して通信を行っている。
これにより、インターネット経由でのデータ送受信が容易になり、さまざまなクライアントからのアクセスが可能となる。

②リクエストとレスポンスのモデル

クライアントがサーバーにリクエストを送ると、サーバーはレスポンスを返す。
リクエストは通常、HTTP メソッドを使って送信される。

③データ形式の標準化

一般的に、Web API では JSON や XML などのデータ形式が用いられる。

④認証とセキュリティ

API にアクセスするためには認証を必要とする場合が多い。
例えば、API キー、OAuth、JWT（JSON Web Token）などの方法がある。

⑤アーキテクチャ

Web API で一般的に用いられるアーキテクチャとその特徴は以下の通り。

- SOAP（Simple Object Access Protocol）
  XML をベースとしたメッセージングプロトコルであり、WSDL（Web Services Description Language）によってインターフェースが定義される。
  堅牢で標準化されたプロトコルを必要とする場面に適している。
- REST（Representational State Transfer）
  リソースという概念を中心に設計されたアーキテクチャスタイルであり、HTTP メソッドを利用してリソースの状態を操作する。
  ステートレスな通信を基本とし、軽量な JSON や XML 形式のデータ形式を扱うことが多いため、高いスケーラビリティと柔軟性を持つ。軽量で開発しやすいことから、Web API の主流となっている。
- GraphQL
  API のためのクエリ言語で、クライアントが必要なデータだけを効率的に取得可能。
  単一のエンドポイントに対して、取得したいデータの構造やフィールドをクライアント側で柔軟に指定できるのが特徴。

⑥URI（Uniform Resource Identifier）

Web 上のリソースを一意に識別するための文字列である。RESTful API では、URI がリソースを表現し、その URI に対して HTTP メソッドで操作を行う。

例: /users（ユーザーの一覧）, /users/{id}（特定のユーザー）

⑦HTTP メソッド

HTTP メソッドは、クライアントがサーバーに対して、指定されたリソースに対して

どのような操作を行いたいかを伝えるためのものである。
主要なメソッドは以下の通り。

GET: リソースの取得
POST: 新しいリソースの作成
PUT: リソースの完全な更新（置き換え）
DELETE: リソースの削除

⑧ステータスコード
API リクエストの結果を表す 3 桁の数値であり、一般的な HTTP ステータスコードに準拠している。
代表的なステータスコードを以下に示す。

コード	ステータス	意味	備考
1xx:情報レスポンス			
100	Continue	リクエストの初期部分は問題なし。 クライアントは処理を継続可能。	実運用ではあまり使用されない。
2xx:成功			
200	OK	リクエストが成功し、期待したレスポンスを返した。	最も一般的。GET や POST などで使用。
201	Created	新しいリソースが作成された。	POST での作成時に使う。
202	Accepted	リクエストは受理されたが、処理は完了していない。	非同期処理など。
204	No Content	正常終了したが、返す内容がない。	DELETE などで使用。
3xx:リダイレクト			
301	Moved Permanently	リソースの URL が恒久的に移動した。	
302	Found	一時的に別の場所へ移動した。	
304	Not Modified	前回と内容が変わっていない。 キャッシュ利用を促す。	
4xx:クライアントエラー			
400	Bad Request	不正なリクエスト（構文エラーなど）。	
401	Unauthorized	認証が必要。認証情報が不足または無効。	
403	Forbidden	アクセス禁止。認証済みでも権限がない。	
404	Not Found	指定リソースが存在しない。	

405	Method Not Allowed	許可されていない HTTP メソッド。	
409	Conflict	リクエストが現在の状態と矛盾。 例：重複登録。	
422	Unprocessable Entity	内容は正しいが、意味的に処理できない。 バリデーションエラー等。	
5xx:サーバエラー			
500	Internal Server Error	サーバ内で予期しないエラーが発生。	
502	Bad Gateway	ゲートウェイやプロキシからの無効な応答。	
503	Service Unavailable	サーバが一時的に利用不能 （過負荷、メンテ中など）。	
504	Gateway Timeout	上流サーバが応答しなかった（タイムアウト）。	

⑨レートリミット

レートリミットとは、一定期間内に API を呼び出せる回数を制限する機能であり、サーバーの過負荷を防ぎつつ、すべての利用者に対して公平な利用を促すことを目的としている。

主な制限の例は以下のとおりである。

・時間あたりのリクエスト数制限
一定時間（例：1 分、1 時間）あたりの API リクエスト数に上限を設ける。
・特定の操作に対する制限
API の中でも、負荷の高い処理や頻繁な更新を伴う操作に対して個別に制限を設ける。レートリミットを超過した場合、HTTP ステータスコード「429 Too Many Requests」が返されるのが一般的である。

⑩バージョン管理

API の仕様が変更された際に、既存のクライアントへの影響を最小限に抑えるための仕組みである。一般的な手法としては、以下のようなものがある。

・URI にバージョンを含める方法
例：https://api.example.com/v1/users
メリット：最も一般的で理解しやすい。
バージョンが URL に明示されるため、API ドキュメントとの整合性がとりやすい。
ブラウザで直接アクセス・検証しやすい。
デメリット：URI が長くなりやすい。
URL 構造がバージョンごとに変わるため、ルーティングやバージョ管理が複雑になる。

・HTTP ヘッダーを使用する方法
例：Accept: application/vnd.example.v1+json
メリット：URL が変わらないため、リソースの一貫性が保たれる。
デメリット：サーバ側でメディアタイプをハンドリングする必要があり、実装がやや複雑になる。
バージョン情報が外部から見えにくいため、開発・テスト・デバッグ時の可視性が低い。

2.4.Web API の利用例

Web API の利用例は多岐にわたるが、いくつか代表的なものを以下に示す。

・ソーシャルメディア API
X（旧 Twitter）や Facebook の API を使って、外部アプリケーションから投稿したり、ユーザー情報を取得したりする。
・地図 API
Google Maps API を利用して、アプリに地図を埋め込んだり、位置情報を提供したりする。
・決済 API
Stripe や Amazon Pay の API を使って、オンラインで決済を処理する。
・天気 API
OpenWeatherMap や Weatherstack などを使って、天気情報を取得する。
・ファイルストレージ／クラウド API
Google Drive や AWS S3 が提供する API を使って、ファイルをクラウドストレージに保存したり、既存のファイルにアクセスする。

3.Web API を EDI で利用する場合の留意点

3.1.EDI と Web API の違い

現在、企業間における商取引データ（受発注、出荷、請求など）の交換手段として、EDI（Electronic Data Interchange）が広く活用されている。
EDI は、かつて紙ベースで行われていた企業間取引を電子化することを目的に発展してきた経緯があり、導入・運用にはいくつかの前提条件がある。たとえば、データ送受信のための「通信回線」や「通信プロトコル」、データ形式を変換する「トランスレーター」といった仕組みが必要であり、さらに業界団体や取引先企業間であらかじめ合意されたデータフォーマットやコード体系に準拠する必要がある。
また、EDI は業務プロセスのデジタル化の流れの中で発展してきたため、複数のデータを一括で処理するバッチ型の運用に最適化されており、リアルタイム性の確保という点では一定の課題が残されている。

これに対し、Web API はシステム間のデータ連携ニーズを起点として設計された技術であり、基本的に HTTP/HTTPS など汎用的なインターネットプロトコルを用いて通信を行う。そのため、専用の通信回線やプロトコルを必要とせず、標準的なネットワーク環境で容易に導入・運用することが可能である。
また、Web API はリクエスト／レスポンス型の通信方式を採用しており、バッチ処理に依

存せずにリアルタイム性の高い処理を実現できる。この特性により、在庫情報の照会やステータス確認など、即時性が求められる業務にも柔軟に対応できるほか、運用面でもきめ細やかな制御や拡張が可能となっている。

近年、EDI と一部の Web API において、それぞれの利用環境や用途が重なりつつあり、開発・運用現場では両者の使い分けや位置づけに関して混乱が生じているケースも見受けられる。本章では、EDI と Web API の本質的な違いを踏まえたうえで、両者の適切な役割分担と運用方法を整理することを目的とするが、特に EDI（ファイル交換型）の枠組みの中で Web API を補完的に活用するケースに焦点を当てて記述する。

なお、Web API の活用によって将来的にリアルタイム EDI の実現が期待されているものの、本ドキュメントではその検討は現時点では対象外とする。
以下に、EDI と Web API の主な特徴をまとめる。

特徴	Web API	EDI
主な目的	アプリケーション間で機能やデータをリアルタイムに連携・利用。 柔軟なデータ連携、新たなサービス連携の実現。	企業間の定型的な商取引文書の電子的交換。 業務プロセスの効率化、手作業の削減。
技術基盤	インターネット HTTP/HTTPS	専用回線またはインターネット 通信プロトコル （JX,全銀 TLS, EDIINT-AS2 など）
柔軟性	比較的高い （多様な連携、機能公開）	標準化されており限定的 （定型的な取引向け）
データ形式	JSON, XML など	EDIFACT, ANSI X.12, 業界標準形式など
データサイズ	小量データ	小～大量データに対応
文字コード	UTF-8 など	EBCDIC,ASCII など
主な利用者	不特定多数の開発者、システム	取引関係にある企業間

3.2. 通信プロトコルとの機能比較

Web API は、HTTP などの Web 技術を使い、アプリケーションやサービス同士が機能やデータをやり取りするための取り決めや仕組みである。Web API は通信プロトコルそのものではなく、HTTP 等のプロトコルの上に構築されたアプリケーション層の設計仕様である。そのため、通信プロトコルと同列に比較することは適切ではないが、両者の役割や位置づけの違いを理解することは、Web API の本質を捉えるうえで有用と考える。
以下に、Web API と主要な通信プロトコルとの機能を比較する。

	Web API	JX 手順	全銀 TCP/IP （広域 IP 網）	ebXML MS V2.0	EDIINT-AS2
通信回線	インターネット	インターネット	インターネット、専用回線など	インターネット	インターネット
メッセージ形式	JSON/XML など	XML など	固定長など	XML	XML など
セキュリティ	SSL/TLS による通信暗号化 API キー、OAuth、JWT など	SSL/TLS による通信暗号化 ベーシック認証、クライアント証明書 EDI 標準に基づくメッセージセキュリティ、認証	SSL/TLS による通信暗号化 全銀手順が持つ ID、パスワードなど	SSL/TLS による通信暗号化 クライアント証明書 EDI 標準に基づくメッセージセキュリティ、認証	SSL/TLS による通信暗号化 クライアント証明書 EDI 標準に基づくメッセージセキュリティ、認証

3.3. API 通信型の選定方針

Web API には、用途や通信特性に応じて複数の通信型が存在する。
EDI 用途において Web API を利用する場合は、リアルタイム性、信頼性、運用性などの観点から、目的に適した通信方式を選定することが重要である。
以下に代表的な通信型の特徴を示す。

通信型	概要	特徴・用途	EDI 用途での適合性
REST（同期型）	クライアントがリクエストを送信し、サーバが即時にレスポンスを返す方式であり、HTTP ベースの標準的な通信モデル。	実装が容易であり、ステートレスなためスケーラビリティを確保しやすく、業務システムとの親和性が高い。	◎ 取引データ送受信など主要な処理方式として最適。
Pub/Sub（非同期通知型）	サーバ側でイベントが発生した際、メッセージキューや Webhook 等を介してクライアントに通知す	クライアントが定期ポーリングを行わずとも通知を受け取ることが	○ 処理完了通知やステータス更新

	る方式。	できるため、非同期イベント通知に適する。	などに有効。
Streaming（常時接続型）	クライアントとサーバが常時接続を維持し、リアルタイムにデータを送受信する方式。	リアルタイム性が高いが、接続維持コストが大きく、監視・再接続制御の設計が複雑となる。	△ リアルタイム監視やイベント配信など限定的な用途に適する。

3.4. セキュリティ

前述のとおり、EDI は企業間取引を背景に発展してきた経緯があり、その歴史的経緯から「データの確実な送達」と「信頼性」に重きが置かれてきた。そのため、EDI で使用される通信プロトコルの多くは、セキュリティ機能を通信層に組み込んでいる。特に、インターネットを利用した EDI では、より強固なセキュリティ対策が講じられている。

一方で、Web API においては、セキュリティ機能を通信プロトコル自体が標準で備えているとは限らず、別途セキュリティ対策を実装・運用する必要がある。以下に Web API と EDI のセキュリティ機能を比較する。

セキュリティ項目	Web API	EDI
通信経路の暗号化	HTTPS	HTTPS、専用回線など
クライアント認証	OAuth2, API キー, JWT, Basic 認証など	ベーシック認証、デジタル証明書、IP アドレス制限など
認可 （アクセス制御）	スコープ制御やロールベース認可など	基本的には取引先ごとの接続単位で制御マスタ等で管理するため柔軟な制御は難しい
改ざん検知	HMAC 署名や JWT 署名など	電子署名
レート制限・DoS 対策	API Gateway 等で可能	通常はサービス提供者側の制限のみで、柔軟性は低い
可用性・冗長性	クラウドベースで高可用性設計が可能	サービス提供事業者に依存（高い信頼性だが拡張性は限定的）
セキュリティ更新	開発側が随時更新可能（モダン設計）	更新頻度は低く、規格変更は慎重（レガシー基盤のことが多い）

①通信経路の暗号化

EDI システムにおいてインターネットを利用する場合、通信経路の暗号化は情報漏えいや改ざんを防ぐために不可欠な要素である。SSL/TLS を利用して通信経路を暗号化することで、ネットワーク上の盗聴や不正な改ざんを防止することができる。

＜EDI＞
インターネットEDIが普及する以前は通信経路として閉域網や専用回線を利用することが主流であったため、外部からの盗聴リスクは比較的低かった。そのため、暗号化は主にインターネット通信手順を利用する場合に導入されることが多い。通信方式や接続クライアントが固定されていることが多く、暗号化方式の変更や証明書更新も計画的に実施可能であり、運用上の影響を最小限に抑えられる構成となっている。

＜Web API＞
Web APIはインターネットを介して多様なクライアントからアクセスされることを前提としているため、暗号化は必須要件であると同時に、以下のような点に考慮する必要がある。

・証明書管理
SSL/TLS証明書の有効期限や中間証明書、ルート証明書の管理を徹底し、期限切れによる通信障害や接続トラブルを防止する。

・暗号化方式の選定
利用するTLSバージョンや暗号スイートの選定に注意し、既知の脆弱性に対応する必要がある。互換性の確保と安全性の両立が求められる。

②クライアント認証
EDIシステムにおいて、通信相手の正当性を確認するクライアント認証は、セキュリティ確保の重要な要素である。EDIとWeb APIでは認証方式や運用方法に違いがあるため、設計段階で留意する必要がある。

＜EDI＞
インターネットEDIが普及する以前は通信経路として閉域網や専用回線を利用することを前提としていたため、外部からの不正アクセスリスクは比較的低かった。そのため、クライアント認証は主に接続IDやパスワードなどの運用ルールによって管理されることが多い。また、一部インターネット通信手順において、プロトコル固有のクライアント認証機能が利用されている。

＜Web API＞
Web APIはインターネットを介して多様なクライアントからアクセスされることを前提としているため、クライアント認証は単なるID/パスワード管理だけでは不十分であり、技術的な仕組みとして組み込むことが求められる。
具体的には、以下のような方式が利用される。

・OAuth 2.0
・APIキー
・JWT（JSON Web Token）

以下にそれぞれの特徴を述べるが、最終的な採用方式については、利用者側のシステム構成、セキュリティ要件、運用方針に基づき判断いただきたい。
各方式には特性および適用範囲が異なるため、自社のリスク許容度および運用負荷を

考慮した上で選定することが望ましい。

項目	OAuth 2.0	API キー	JWT (JSON Web Token)
目的	認可	主に簡易認証	認証とトークンベースの情報伝達
特徴	高度な認可に向く。 外部サービス連携やユーザーごとのアクセス制御に適している。	最も簡単だが、セキュリティ要件が高い環境には適さない。	サーバーレス API やマイクロサービスでのトークン型認証に適しており、処理が高速。
使われ方	サードパーティアプリに限定的なアクセスを許可	アクセス制限のない基本的な認証	認証後に発行し、トークンでユーザー情報をやりとり
特徴	認可とアクセストークンによる権限管理	シンプルなトークンによる認証	トークンに署名を含み改ざん防止できる
トークン形式	アクセストークン＋ リフレッシュトークン	単一のキー（文字列）	ペイロード付きの JWT 文字列（Base64 エンコード）
セキュリティ	高 （設計次第で強固）	低 （漏洩すると完全アクセス可能）	中～高 （署名付き・期限付き）
認可の柔軟性	高 （スコープで操作制限可）	低 （操作制限なし）	中 （トークン内容に依存）
有効期限	アクセストークンに期限あり（短期）	任意 （一般的には無期限）	有効期限内であれば有効
メリット	・細かなアクセス制御が可能 ・安全な外部連携	・実装が簡単 ・早く動く	・ステートレスで高速 ・自己完結で情報を持てる
デメリット	・実装が複雑 ・セキュリティ設計が必須	・漏洩リスクが高い ・制限しづらい	・トークンサイズが大きい ・漏洩時のリスク高

③認可

アクセスしてきたクライアントが正当なものであることを確認（認証）した後、「何に対して、どのような操作が許可されているか」を決定する。

＜EDI＞
EDI における認可について、アクセス権限の制御はサービス提供者側の内部業務システムに集約されており、利用者ごとに細かく設定する仕組みはほとんど存在しなかった。例えば、ある取引先は「受発注データの送受信が可能」といった大枠の権限を与えられるのみであり、ユーザーやアプリケーション単位で「この種類の取引だけ」「この処理は参照のみ」といった粒度での制御は想定されていなかった。
結果として、運用はシンプルな反面、柔軟性に欠けるという特徴があった。

＜Web API＞
Web API では、クライアントの認証後に、ユーザーやアプリケーションごとに「許可された操作」や「アクセス可能なリソース」を細かく制御することが可能である。
特に、Web API では以下のようなアクセス制御手法により、よりきめ細かく柔軟な運用が可能となっている。

・スコープ制御
・ロールベースアクセス制御
・属性ベースアクセス制御
・アクセスコントロールリスト

このように、従来の EDI では実現が難しかった柔軟なアクセス制御を可能とするため、Web API の認可機能を EDI 用途で活用する際には、アクセス権限の設計について十分な検討が不可欠である。

④（補足）OAuth 2.0 認可方式の取扱い
本ガイドラインにおける認証・認可は、OAuth 2.0 を利用することを前提としている。OAuth 2.0 には複数の認可方式が存在するが、近年のセキュリティ動向や標準仕様の更新により、利用が推奨される方式と、非推奨または廃止が検討されている方式がある。以下に、2025 年 12 月時点における代表的な認可方式の概要と推奨状況を提示する。

(1) 推奨されるフロー

フロー名	概要	推奨度
Authorization Code Flow	認可コードを介してアクセストークンを取得する方式。サーバ側で安全にトークンを処理できるため、Web アプリなどに適している。	◎推奨
Authorization Code Flow + PKCE	PKCE（Proof Key for Code Exchange）を併用し、クライアントシークレットを持たない環境でも安全に利用できる方式。	◎強く推奨
Client Credentials Flow	クライアント自身の認証情報のみでトークンを取得する方式。ユーザー操作を伴わない機械間通信に適している。	◎推奨

Refresh Token Flow	有効期限が切れたアクセストークンを再取得するための方式。長期稼働システムや再認証を避けたい環境で利用する。	○利用可
Device Authorization Flow	入力装置を持たないデバイス（IoT 機器など）での認可に対応する方式。	○利用可 （用途限定）

（2）非推奨または廃止が検討されているフロー

フロー名	概要	非推奨理由
Implicit Grant	認可コードを介さず、トークンをブラウザ経由で直接受け取る方式。	アクセストークンが URL 上に露出し、漏洩リスクが高い。
Resource Owner Password Credentials Grant（ROPC）	ユーザーID とパスワードを直接クライアントに入力させる方式。	アプリがパスワードを扱うため、漏洩リスクが高い。
Plain PKCE（code_challenge_method=plain）	PKCE のハッシュ化（S256）を行わず、平文で送信する方式。	保護効果がなく、中間者攻撃に弱い。
SAML 2.0 Bearer Assertion Flow	SAML を使って OAuth トークンを取得する旧来方式。	OIDC（OpenID Connect）への移行が推奨されている。

※今後の仕様改訂について

タスクフォースで標準化が進められている OAuth 2.1 では、Authorization Code Flow（PKCE 対応）を標準方式として明示し、Implicit Flow と ROPC Flow は正式に削除される予定となっている。そのため、今後の実装や運用にあたっては、最新の情報を調査した上でこれらの仕様変更を見据えた設計・更新計画を検討することが望ましい。

⑤改ざん検知

データが送受信の過程で第三者により改ざんされていないかを検知する。これはデータの完全性を保証するために不可欠な要素であり、誤発注や不正取引を防ぐ上でも重要である。

＜EDI＞

インターネット EDI が普及する以前は通信経路として閉域網や専用回線を利用することが主流であり、通信経路の安全性は比較的高かった。そのため、改ざん検知は主にファイル単位やジョブ単位でのチェックサムやハッシュ値によって行われることが多い。一部のインターネット EDI プロトコルでは SSL/TLS を基本とし、さらに高度な改ざん検知機能を備えている場合もある。

＜Web API＞
Web API では SSL/TLS による改ざん検知を基本としつつ、データ本体にデジタル署名や HMAC 署名を付与することで、より柔軟かつ強固な改ざん検知を実現できる。
以下に、2025 年 12 月時点における代表的な改ざん検知方式の概要を提示する。

手法	内容	特徴
HTTPS/TLS	通信内容全体を暗号化して改ざんを防ぐ	標準的。盗聴・中間者攻撃にも有効
HMAC 署名	秘密鍵とリクエスト内容からハッシュを作成。サーバ側で検証	内容の一部でも改ざんされると検知可能
JWT の署名検証	JWT トークンの署名をサーバで検証（公開鍵 or 秘密鍵）	アクセス制御情報を含み、改ざん検知と認可を両立
コンテンツ・ダイジェストヘッダ（Digest）	リクエスト/レスポンスの本文にハッシュを付与し整合性を確認	HTTP 仕様（RFC3230/7616）にも準拠
リクエストのタイムスタンプ＋Nonce	一定時間内のリクエストのみ有効にし、リプレイ攻撃を防ぐ	改ざん＋再送信の両方に対応可能
API Gateway/WAF のシグネチャ検証	API Gateway が署名・ヘッダ・ハッシュ等を一括検証	環境側で集中管理が可能

⑥レート制限/DoS 対策
外部からの過剰なアクセスや攻撃的なリクエストを制御し、サービス全体の安定性を確保する。特に EDI のように取引業務を扱うシステムでは、短時間のリソース逼迫であっても業務停止に直結するため、十分な対策が求められる。

＜EDI＞
専用回線や閉域網を利用した EDI の場合、同時接続数を制限することで、自然にレート制御や DoS 対策が実現されていた。送信されたデータはセンター側で管理されるため、処理の過負荷やサービス停止のリスクは低く、また契約単位での ID・パスワード管理や IP 制限によって、不正アクセスも運用ルールの中で制御されている。しかし、近年ではインターネット EDI が主流となりつつあり、従来型の運用的な制御だけでは十分でなく、URL を非公開にしたり、送信元 IP アドレスを制限するなどの追加的な対策が必要となっている。

＜Web API＞
Web API はインターネットに公開されることを前提とするため、不特定多数からのアクセスが可能である。そのため、EDI のように「接続環境の制約による防御」に依存することはできず、明示的に技術的な制御を実装する必要がある。
具体的な対策としては以下が挙げられる。

・レート制限
一定時間内に許容されるリクエスト数を制御し、過剰アクセスを抑制する。

例えば、クライアントごとに秒間・分間・日間のリクエスト上限を設定し、上限を超えた場合は「HTTP 429（Too Many Requests）」を返却する、必要に応じてRetry-After ヘッダで再試行可能時刻を通知する、といった対策を行う。

・IP 制御
不正な送信元からのリクエストを遮断するために、送信元 IP アドレスの制限やブラックリスト/ホワイトリストを活用する。

・バースト吸収
一時的なアクセス集中（バースト）に対して、トークンバケット方式などを用いて吸収し、全体の処理能力を超えないように制御する。

・DoS／DDoS 対策
API Gateway や WAF（Web Application Firewall）によって、異常なアクセスや不正トラフィックを検知・遮断する。異常検知は監視システムと連携し、自動でアクセス制御ルールを強化できるように設計する。

・監査・可視化
取引先ごとのリクエスト数やエラー状況をログとして収集し、ダッシュボードで可視化することで、異常傾向を早期に把握する。これにより、正規業務と不正アクセスの切り分けが容易になる。

⑦可用性・冗長性
サービスが停止すると、ユーザー企業の業務そのものに大きな影響を与えるため、可用性と冗長性を確保しておくことは極めて重要である。特に EDI のように取引データを扱うシステムでは、一時的な停止が発注や配送、決済に直結するため、その影響範囲は広い。

＜EDI＞
専用回線や閉域網を利用した EDI では、通信経路自体の安定性が比較的高く、センター側では EDI エンジンやバッチ処理サーバを複数台構成することで一部の処理サーバが停止しても残りのサーバで処理を継続することが可能であるため、外部要因によるサービス停止のリスクは限定的である。
さらに、ジョブやファイル単位で再送処理が可能な設計になっており、障害発生時にも運用ルールとしてリカバリできる仕組みが整備されている。つまり、EDI の可用性・冗長性は、物理的冗長化と運用プロセスの組み合わせによって自然に担保されていたと言える。

＜Web API＞
Web API はインターネットを介して公開され、多様な利用環境からアクセスされることを前提としている。そのため、従来型の運用依存の冗長化だけでは十分でなく、技術的な冗長性を設計段階で組み込むことが不可欠である。
具体的には、以下のような設計要素を考慮する必要がある。

・冗長構成の設計

Web サーバ、アプリケーションサーバ、データベース、API Gateway などの主要コンポーネントを冗長化し、単一障害点（SPOF）を排除する。

・自動フェイルオーバー
障害検知後、トラフィックを正常系サーバへ自動切替する仕組みを導入し、サービス停止時間を最小化する。

・負荷分散
複数のサーバ間でアクセスを分散するロードバランサを利用し、処理能力の平準化と冗長性を両立する。

・監視・アラート
サーバやサービスの稼働状況を監視し、異常発生時には即座に通知。障害の早期検知と対応を可能にする。

・データの冗長化・バックアップ
データベースやファイルストレージは複数拠点で冗長化し、障害やデータ損失に備える。

⑧セキュリティ更新
EDI システムを安全に運用するためには、セキュリティ更新を計画的に実施することが不可欠である。EDI と Web API では、更新の対象や手順、影響範囲が異なるため、設計段階から留意しておく必要がある。

＜EDI＞
専用回線や閉域網を利用した EDI では、インターネット経由の脅威は限定的であった。インターネット EDI が普及した現在においても、SSL/TLS 証明書や各種ミドルウェア・ライブラリの更新、脆弱性対応などは、定期的な運用作業として管理されることが多く、更新による業務影響も比較的抑えやすい環境である。

また、通信方式や接続クライアントが固定されているため、証明書更新や暗号化方式の変更も事前に調整して計画的に行うことが可能である。運用担当者は、更新手順をマニュアル化し、障害発生時のリカバリ手順を整備することで、サービス停止や接続トラブルを最小限に抑えることができる。

＜Web API＞
Web API はインターネット公開が前提であり、多様なクライアント環境からのアクセスが想定される。そのため、セキュリティ更新の影響範囲は EDI に比べてはるかに広く、更新作業自体を慎重に設計することが求められる。
具体的には、以下のポイントが重要である。

・証明書管理
SSL/TLS 証明書の有効期限管理を徹底し、期限切れによる通信障害を防止する。更新時には、クライアント側で中間証明書やルート証明書の更新が必要な場合もあるため、事前通知や手順整備が重要である。

・ライブラリ・フレームワーク更新
利用するライブラリや API フレームワークの脆弱性対応を定期的に実施する。更新時には互換性や API 仕様への影響を検証することが必要である。

・セキュリティパッチの適用
OS やミドルウェア、データベースのセキュリティパッチも対象である。自動適用と手動適用のポリシーを明確化し、更新時の影響を最小化する設計が望ましい。

・事前通知・運用調整
クライアントに影響が出る更新は事前通知を行い、更新作業時間を調整することで業務への影響を抑えることが重要である。

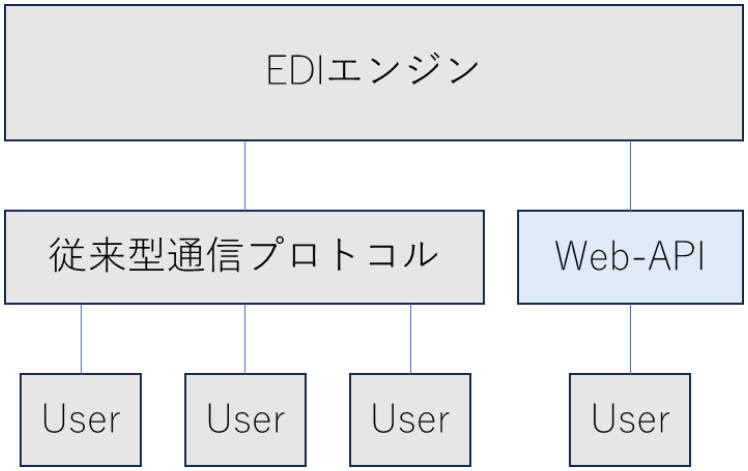
3.5. データサイズ

Web API はリアルタイムでのシステム間連携を前提として設計されており、小規模かつ高速な通信に適している。そのため、EDI のようにファイル単位で大量のデータを一括送受信する用途には必ずしも最適ではない。EDI 用途で Web API を利用する場合、大量データの送受信はパフォーマンスやレスポンスに大きく影響する。そのため、データサイズに応じた適切な設計と実装が必要である。
本ガイドラインでは、データサイズに応じた実装イメージとして、以下の 2 つの方式を参考として提示する。

	直接送受信	間接送受信
概要	EDI のように、Web API を利用して直接トランザクションファイルを送受信する。	大量データ送受信時の負荷を分散するため、トランザクションファイルは外部ストレージなどに保管し、Web API では送受信に必要な最小限の情報のみをやり取りする。
処理対象	トランザクションデータ本体	設定ファイルまたはメタデータ
データ量	小～中程度に適する	大容量に適する
API 負荷	高め（分割・圧縮などが必要）	軽量（通知・状態管理中心）
実装・運用の 複雑性	中 API 内で全処理を担う必要がある	高 外部ストレージの管理や通知フローの設計など、追加の管理項目が発生することで運用全体が複雑化する可能性がある

■直接送受信方式

Web API を利用して、トランザクションデータそのものを直接送受信する方式。
この方式は少量データの送受信に適しているが、大量データを扱う場合には以下の点に注意が必要である。



①データ分割

トランザクションデータを複数ファイルに分割し、1 件あたりのデータ量や件数に上限を設けてシステム負荷を軽減する。分割方式（件数単位、サイズ単位、伝票単位など）は事前に取り決めておく。

②専用エンドポイントの設置

大量データ送受信時のシステム負荷を軽減するために、通常のエンドポイントとは別に、バルクデータ専用のエンドポイントを設け、処理の分散と効率化を図る。

③ペイロードの圧縮

通信帯域の効率化を目的にトランザクションデータを圧縮する。クライアント／サーバー双方が対応している必要がある。圧縮方式（gzip など）は事前に取り決めておく。

④タイムアウト設定

通信の種類や処理内容に応じた適切なタイムアウト値を設定する。

⑤再送処理（リトライ）

通信エラー時のアクション、最大リトライ回数、待機間隔などを検討、定義する。

⑥セッション管理

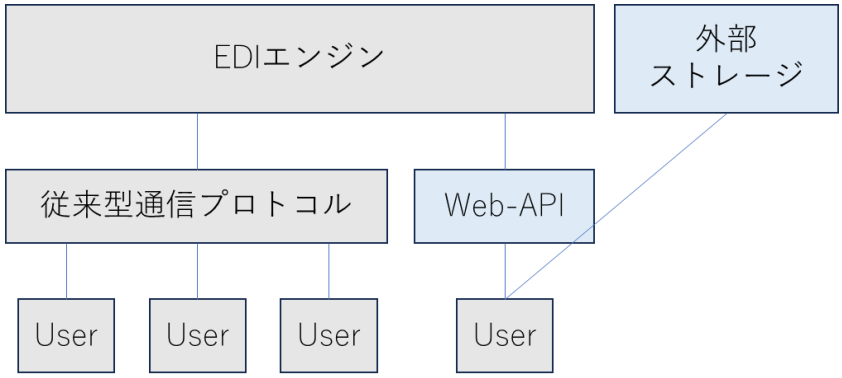
セッションタイムアウトの発生がトラブルの原因となりやすいため、通信経路上のセッション有効期間や制御方式について事前に確認・共有しておく。

⑦ストリーミング対応（可能であれば）

メモリに全データを保持せずに逐次処理を行うことで、パフォーマンスと安定性を向上させる。

■間接送受信方式

トランザクションデータを外部ストレージに格納し、Web API ではその参照や管理のみを行う方式。この方式は、大量データを扱う際に通信負荷を抑えつつ、効率的なデータ処理を実現できる点が特長である。ただし、直接送受信方式における注意点に加えて、以下の点にも留意する必要がある。



①ファイル管理とメタ情報の分離

トランザクションデータはファイルとして保管し、API ではファイル名・種別・送信日時などのメタデータをやり取りすることで、API の負荷を抑える。

②安全なファイル転送手段の選定

SFTP、HTTPS 経由の署名付き URL（例：AWS S3 の Pre-signed URL）など、安全性の高い手段で転送を行う。

③ファイル保存期間・責任分界の明確化

ファイルを一時的に保管する場合、保存期間や管理責任範囲を明示することで、トラブル時の対応を明確にする。

3.6. ログ取得・トラッキング

EDI において、ログ取得やトラッキングは運用上重要な機能であり、特に取引履歴の証跡管理や障害発生時の原因追跡、監査対応に不可欠な要素である。Web API はリアルタイム通信が前提であり、多様なクライアントや多地点からのアクセスが可能であるため、ログの粒度や種類などにおいて EDI と同等以上の情報を取得し、ユーザーに適切に提供・可視化する設計が求められる。

API ゲートウェイやトレースツール等を活用することで、アクセスログ・操作ログ・エラー情報・リクエスト ID 等を取得、統合管理することが望ましい。また、ユーザー視点から、EDI と同環境でワンストップで情報を把握できることが理想的である。

以下に、一般的に取得されるログ項目を比較する。

項目	Web API	EDI
送信元/宛先情報	クライアント ID、IP アドレス、 ユーザーID、API キーなど	送信元 ID、受信先 ID など （プロトコルに応じる）

送受信日時	タイムスタンプ （秒・ミリ秒単位）	通信確立日時・ファイル送受時間
通信ステータス	HTTP ステータスコード	HTTP ステータスコード、通信プロトコル等が持つ正常終了/エラーコード、業務アプリケーションからのステータスコードなど
取引 ID	UUID、リクエスト ID、 X-Correlation-ID など	メッセージ ID・電文識別子など
エラー	API レスポンスメッセージ、 例外ログ、スタックトレース	回線断、フォーマット異常など
データ件数	ヘッダ情報、件数カウントが必要	ヘッダ情報、件数カウントが必要
操作ユーザー情報	JWT トークンや OAuth 認可情報から 特定可	基本的になし （システム単位の認証）
署名・認証記録	HMAC 署名、JWT 署名、 OAuth 認可ログあり	電子証明書
トラッキング ID	API Gateway や Tracing システムで 一元管理	専用ミドルウェアや VAN など
ログ保存先	クラウドログ （例：CloudWatch, ELK, Datadog など）	専用ミドルウェアや VAN など
トレーサビリティ	相関 ID によりサービス横断の 処理可視化	専用ミドルウェアや VAN など

3.7. 信頼性

EDI は業務の中核を担うシステム間連携手段として広く利用されており、取り扱うデータの重要性からも、高い信頼性の確保が必須とされている。そのため、Web API を EDI の通信手段として活用する場合においても、「通信が可能であること」だけでなく、「データが確実に届き、正しく処理されること」を前提とした設計および運用が求められる。以下に、その実現に向けて留意すべき点を整理する。

①到達確認

確実に送受信が行われたことを確認するためのレスポンス設計や、ログを記録する仕組みが必要である。HTTP ステータスコードに加えて、アプリケーションレベルでの応答を活用し、システム間で明確に通信成功を判断できるようにする。

②重複検知

ネットワークやアプリケーション等の障害によって、同一データが誤って複数回送信される可能性がある。受信側がそのまま処理を行ってしまうと、二重発注や二重請求とい

った業務上の不具合につながるため、システム側で重複リクエストを検知・排除する仕組みを設ける必要がある。

(1)重複検知の目的

・一意性の確保
　各メッセージに固有の識別子を持たせ、他のメッセージと区別する
・再送時の安全性
　送信側が再送した場合でも、受信側が同一メッセージと判断して二重処理を回避できる
・整合性の担保
　送信・受信の状態管理と組み合わせることで、処理の不整合を防止できる
・追跡性
　メッセージごとに識別できるため、ログ追跡が容易になる

(2)メッセージ ID の利用

重複検知のキー情報としては、メッセージ ID を利用するケースが一般的である。送信側（クライアント）は、各 EDI メッセージに対して一意な ID（MessageID や TransactionID など）を生成し、必要に応じてタイムスタンプ等と組み合わせてデータに付与する。これにより、受信側は「すでに処理済みのメッセージかどうか」を判断できる。

③再送制御（リトライ制御）

通信エラーやタイムアウトなどによる回線断が発生した際に、同一データを再送する仕組みを検討する。

④改ざん検知

通信経路上でデータが第三者により不正に変更されていないことを検証するための仕組みを検討する。具体的な手法としては、「3.4.セキュリティ」の「●改ざん検知」を参照されたい。

⑤通信制限

EDI は特定の取引先との通信を前提としており、不特定多数からのアクセス集中は想定されていない。そのため、大量データの送受信に対しては負荷分散が施されているものの、過負荷対策自体は重視されてこなかった。一方、Web API はインターネット経由での接続が前提であるため、DoS 攻撃や過剰なリクエストによるシステム負荷を防ぐ必要がある。そのため、リクエスト数を制限するレートリミットの導入が重要となる。具体的な手法としては、「3.4.セキュリティ」の「●レート制限/DoS 対策」を参照されたい。

3.8.その他

システム面以外において、Web API を EDI 用途で運用する上で考慮が必要な点をその他として整理する。

①障害時対応

エラーハンドリングの明確化やユーザーへの障害通知、ユーザー対応など、現在の EDI

で行っている対応を踏襲することが望ましい。

②変換処理
フォーマットや文字コードの変換が必要な場合には、専用のユーティリティを実装することを検討する。例えば、固定長フォーマットから JSON/XML 形式への変換や、EBCDIC から UTF-8 への文字コード変換など、EDI での実装は負荷が大きいケースが多い。
これらの処理を Web API の周辺機能として実装することは、Web API の柔軟性を活かした有効な手段といえる。

③エラー処理
API でエラーが発生した場合に、適切なエラーメッセージをユーザーに返送する。
エラーメッセージやエラーコードについては、EDI で利用されているものを踏襲しつつ、HTTP ステータスコードを適切に活用し、必要に応じて詳細情報をレスポンスボディで返却することが望ましい。なお、エラーメッセージやエラーコードの標準化は、業界固有のルールや業務内容に依存する場合が多く、統一的な標準化は困難である。このため、本ガイドラインではエラーメッセージに関する標準化は対象外とする。

④ログ管理
EDI では問題発生時の切り分けのために多様なログが取得されている。
Web API においても、アクセスログや処理ログなどを可能な限り記録し、トラブルシューティングや監査に活用できるようにすることが望ましい。
特に「データを受信したかどうか」に関する確認は頻繁に発生するため、受信状況を確実に把握できるよう、該当ログを確実に記録・保管しておくことが重要である。

⑤バックアップと復旧
データロストに備え、責任の所在や保存期間などについて、事前に関係者間で明確に取り決めておくことが望ましい。また、システム全体のバックアップ体制および復旧方針についても、SLA（サービスレベル合意）を踏まえて事前に合意を得ておく必要がある。

⑥ドキュメント
API の利用方法や仕様を明確に記述したドキュメントを提供し、利用者がスムーズに API を利用できるようにする。

⑦API 管理
API のライフサイクル全体を管理するための仕組みを検討する。API ゲートウェイなどを活用して、API の公開、バージョン管理、利用状況の把握などを行う。

⑧コスト管理
API の運用コストを把握し、必要に応じて最適化を行う。クラウドサービスを利用する場合など、料金体系の把握と請求金額をモニタリングしておくことが望ましい。必要な場合は為替レートにも注意する。

⑨変更管理:
API に変更を加える際には、影響範囲を十分に考慮し、適切なテストやリリース手順を

踏んで実施する。API のバージョン管理を適切に行い、下位互換性を可能な限り維持する。

⑩接続テスト環境の準備
クライアントが自由に接続テストを実施できるよう、本番環境とは別に接続テスト環境を用意しておくことが望ましい。

3.9.API 公開に関する考え方

Web API を外部（取引先企業など）に公開する場合、通信の安全性を確保するだけでなく、「誰が・どのように・何を」利用できるのかといったアクセス管理と運用ルールの整備が不可欠である。以下に API 公開にあたっての主な留意点を整理する。

①公開範囲の明確化
API を「完全公開（パブリック）」するのか、「限定公開（取引先限定）」とするのか、明確に定義する。EDI 用途の場合、基本的には取引先限定のプライベート API として運用するケースが一般的であると考える。

②API 仕様の整備と提供
利用者が正しく API を実装できるように、以下の情報を含んだ「API 仕様書」を提供する。

＜概要＞
エンドポイント一覧
認証・認可方式
パラメータ定義
レスポンス例・エラーコード定義
レート制限・注意点など

＜セキュリティ対策＞
API 認証方式の導入
IP 制限やアクセス元制御
TLS による暗号化
利用ログの取得と監査

③ライフサイクル管理
API のバージョン管理（例：v1, v2）を行い、互換性や移行パスを明確にする。
廃止予定の API に対しては事前告知期間を設け、ユーザーに影響が出ないよう段階的に対応する。

④公開プロセスと承認フロー
API の新規公開・変更・廃止には、システム部門・セキュリティ部門・業務部門などによる承認プロセスを通すことで、品質と統制を担保する。

⑤利用者管理・契約管理
API 利用者（クライアント）の登録・認可・識別（クライアント管理）を厳密に行う。

特に EDI 用途では、API 利用は業務契約に基づくことが多いため、契約と API 利用権限の紐づけを明確にする仕組みが必要となる。

⑥API 管理基盤の活用

API Gateway を活用することで、上記のようなアクセス制御・認証・監視・レート制限などの管理を一元的に実施することが可能である。EDI 用途ではプライベートなネットワーク内での運用が一般的であるため、API Gateway を導入するケースは少ないが、特定の機能をうまく取り入れることでセキュリティ向上や運用負荷の軽減につながる可能性があるため、考慮されたい。

4. システム構築例

本章では、システム構築例をセンター側およびクライアント側のシステム設計者ならびに運用担当者向けに提示する。ここで示す内容は、あくまで参考であり、実際の環境・要件に応じて適宜調整することを前提とする。

各設定は以下の通りとする。

①想定するプレイヤーとストーリー

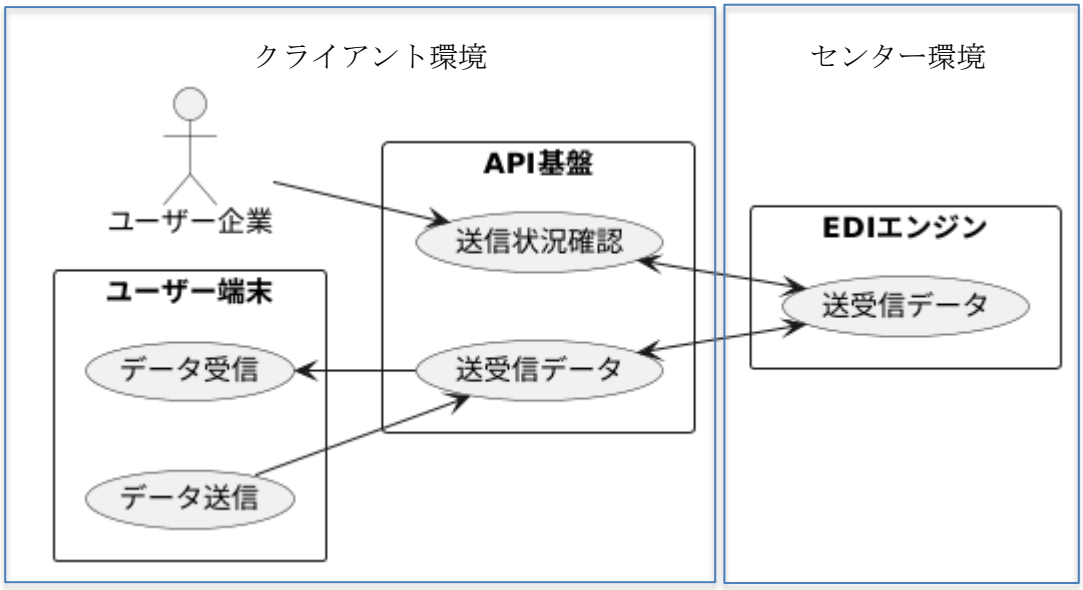
(1)センター

EDI 環境を継続運用しつつ、新たに Web API 環境を併設して提供する。
セキュリティ強度および運用レベルについては、EDI 環境と同等以上を維持することを前提とする。

(2)クライアント

センターが提供する Web API 環境に接続し、データ交換を実施する。

②ユースケース



③設定

設定項目	設定値	理由・補足
交換方式	直接送受信	構成がシンプルで、広く利用されている直接送受信方式を対象とする。
API 通信型	REST （同期型）	Web API で標準的に利用されているため。
通信経路の暗号化	HTTPS （TLS 1.2 以上）	セキュリティ確保のため TLS1.2 以上を必須とする。
クライアント認証	OAuth 2.0	セキュリティ強度の観点から OAuth を利用する。
認可	サービス提供者側管理に従う	EDI 用途において複雑な認可は不要と考え、従来のサービス事業者側での管理方式に従う。
タイムアウト時間	30～60 秒	ファイルサイズに起因する処理遅延を想定した時間を設定する。 タイムアウトの判断基準はネットワーク機器を含めた全体のスループットではなく、サーバからの応答時間に基づいて設定する。
自動リトライ	なし	データ重複を避けるため、自動リトライは行わない。
データサイズ	～10MB 程度	大量データは想定しない。
データ圧縮	なし	大量データは想定しない。
ファイル形式	従来形式（固定長、XML など）	従来型の EDI ファイル形式に従う。
ファイル送信形式	バイナリデータ送信 （Content-Type: application/octet-stream）	ファイルデータを API で扱いやすくする。
署名/ハッシュ検証	SHA-256	改ざん検知、内容完全性を確認する。
レスポンス設計	ステータスコード	成否判定とエラー内容を明確化する。
エンドポイント URL	クライアント＞データ種ごとに用意	例： /api/v1.x/customer1/order/yyyymmddhhmmss /api/v1.x/customer1/invoice/yyyymmddhhmmss

4.1.センター側

4.1.1.システム開発者向け

センター側環境は提供事業者によって大きく異なるため、具体例の提示は困難である。そこで、従来の EDI 基盤に Web API 機能を追加する場合において一般的に考慮すべき要素を上記設定を基に記載する。

■A-1.機能

①通信セキュリティ

(1)暗号化通信

・HTTPS（TLS1.2 以上）による暗号化通信の実装

・将来の標準化動向に備え、TLS1.3 への対応を推奨

(2)証明書管理

・サーバ証明書の導入および適切な管理

・有効期限切れによる通信断を防止するための更新プロセス整備

・自動更新機構（例：ACME プロトコル等）の導入を検討することが望ましい

(3)アクセス制御

・ファイアウォールによる不正アクセス防止

・WAF によるアプリケーション層の攻撃対策（SQL インジェクション、XSS 等）

(4)監査・トレーサビリティ

・通信ログの取得と長期保存

・認証・認可操作を含む監査ログの確保

・障害解析・不正調査に活用可能なトレーサビリティの確保

②認証・認可

(1)利用者認証

・OAuth2.0 に基づく認証方式を採用し、クライアントごとのアクセストークンを発行・管理する

・必要に応じてクライアント証明書や ID プロバイダ連携（OpenID Connect 等）を組み合わせ、信頼性を高める

(2)認可モデル

・利用者の権限管理は、EDI での管理手法に準拠

(3)多要素認証

・セキュリティリスクの高い操作に対しては、多要素認証の利用を検討する

・ワンタイムパスワード、端末証明書、生体認証などを組み合わせることで、不正利用リスクを低減

(4)運用考慮

・アクセストークンの有効期限・失効管理（強制失効、期限切れ時のリフレッシュ処理）

・認証・認可のログ取得と監査証跡の保持

③API 設計・実装

(1)インタフェース設計

・REST（同期型）を採用し、REST/JSON を基本としたリクエスト／レスポンス

定義
・必要に応じて XML 形式や SOAP との互換性を検討
（EDI との親和性確保のため）
・リクエスト ID の付与により、トランザクション単位での追跡性を確保
(2)ファイル送受信 API
・アップロード／ダウンロード機能を提供
・10MB 程度までのファイルサイズに対応（multipart/form-data 等を利用）
・offset や limit といったパラメータによるオーバーフロー防止
(3)バージョン管理
・URI でのバージョン付与（例：/v1/, /v2/）
・旧バージョンの並行稼働期間を定め、利用者への移行を計画的に実施
・メジャーバージョンに対する対応方法
→ URI に含まれるメジャーバージョンを更新する
→ 後方互換を保つため、一定期間バージョンの並行運用を行う
・マイナーバージョンに対する対応方法
→ 既存パラメータはそのまま、新たなパラメータを追加することで対応する
(4)エラーハンドリング
・HTTP ステータスコードに基づく統一的なエラーレスポンス
・アプリケーションエラーコードを付与し、業務上の意味を明示
・冪等性確保（同一リクエストの再送に対して安全に処理できる設計）
(5)URI 設計と後方互換性
・既存クライアントとの後方互換性を確保する
・URI にメジャーバージョンと API 名を含める
・リソース指向の設計とし、パスを短くする
・URI にサーバ側アーキテクチャ情報を反映させない
・スペース、エンコードを必要とする文字を利用しない

④信頼性・可用性
(1)再送・重複検出
・再送による二重処理を防ぐため、メッセージ ID や取引番号を用いた重複検出の仕組みを実装する
・EDI で一般的だった「送信済／受信済ステータス管理」の考え方を API 設計にも反映させることが重要。
(2)スケーラビリティ確保
・高負荷時にも安定稼働できるよう、API サーバの水平スケールアウトを前提とした設計とする。
・レート制限やスロットリングを組み合わせ、サービス全体のリソースを保護する。
(3)フェイルオーバー／冗長化構成
・システム障害やネットワーク障害に備え、冗長構成（アクティブ-アクティブ／アクティブ-スタンバイ）を導入する。
・フェイルオーバー時に取引が途切れたり、二重処理が発生したりしないよう、取引状態を共有できる仕組みを準備する。
・障害時の切替時間（RTO）やデータ損失許容範囲（RPO）を明確化し、ユーザー

に周知しておくことが望ましい。

⑤使用性

(1)使用方法の公開

・提供する API に関連する業務内容および API の使い方をドキュメントとして公開し、利用者が仕様や利用手順を容易に理解できるようにする

(2)API シミュレータ

・API シミュレータを用意し、サンプルデータを用いたリクエスト／レスポンスを実際に試行できる環境を提供することで、接続試験や動作確認を容易にする

(3)情報提供

・開発者向けサイト等を通じて、API キーの発行、サンプルコード、FAQ などの補助情報を一元的に提供する

⑥運用管理

(1)利用者別ログ管理

・利用者単位でのアクセスログ・処理ログの取得

・ユーザー自身による照会を可能とする証跡管理機能の提供

(2)サービス利用制御

・契約に基づく SLA（応答時間、可用性など）の監視と管理

・利用可能時間帯の制御（例：深夜バッチ処理との調整）

(3)障害通知機能

・エラー発生時のレスポンスコードや詳細メッセージの返却

・障害や異常を検知した際のメール／アラート通知

(4)運用監視

・死活監視（稼働確認、ヘルスチェック API など）

・リソース監視（CPU、メモリ、ネットワーク帯域、ストレージ使用量など）

(5)運用補助機能

・データの再投入（リトライ、リカバリ処理）

・障害発生時における代替手段の提供（代替 API、バックアップルートなど）

⑦セキュリティ対策

(1)入力値検証

・クライアントから送信されるリクエストパラメータやファイル内容について、インジェクション攻撃（SQL、コマンド、スクリプト等）を防ぐための検証を行う。

(2)受信ファイルのウイルスチェック

・クライアントから受信するファイルに対してウイルススキャンやマルウェア検知を行うことが望ましい。

(3)レート制限／DoS 対策

・過剰なリクエストや攻撃的なアクセスからサービスを保護するため、レート制限やスロットリングを適用する。

(4)API Gateway／リバースプロキシによる公開制御

・外部公開される API に対して、ゲートウェイやリバースプロキシを活用し、アクセス制御、認証・認可の集中管理、リクエストのフィルタリングを行う。

⑧移行対応（EDI 通信手順 → Web API）

(1)概念のマッピング

・「接続先 ID」や「送受信 BOX」といった EDI 固有の概念を、API のエンドポイントやリソース設計に対応させる。これにより、クライアント側の設定変更や業務フローへの影響を抑えつつ、API 環境に移行する。

(2)接続試験環境の提供

・クライアントが本番切替前に API を検証できるよう、専用の接続試験環境を用意する。テスト用データやシナリオを整備し、実際の運用に近い形で動作確認が行えることが望ましい。

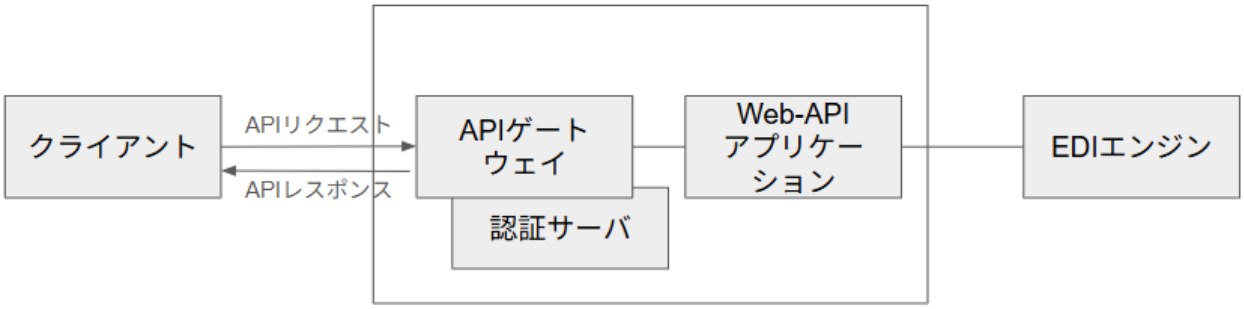
(3)EDI 通信手順との並行稼働

・移行期間中は、必要に応じて EDI と Web API の並行稼働を許容する。
これにより、移行中のトラブルや業務停止リスクを低減できる。並行稼働に伴うデータ整合性や重複処理に対する設計も併せて検討することが望ましい。

(4)移行スケジュールと切替手順の明確化

・移行計画、段階的切替の手順、関係者への周知を明確化する。切替時の停止時間やデータ移行方法、障害発生時のリカバリ手順もあらかじめ設計する。

■A-2.システム構成



＜項目説明＞

API ゲートウェイ：認証・レート制限・ルーティングを担う。

Web API アプリケーション：REST エンドポイントの実装本体。
ファイル受信・登録・変換・応答処理などを行う。

認証サーバ：アクセストークンの発行など、API セキュリティ全般を担う。

EDI エンジン：業務データを管理する。

■A-3.API 仕様設計

①リクエスト例

```
PUT /api/v1.x/files/sample HTTP/1.1
Host: edi-center.com
Authorization: Bearer abcdef123456
Content-Type: application/octet-stream
X-Filename: order_20250801.csv
X-File-Type: text/csv
```

```
X-Message-Id: MSG-20250801-000123
Content-Length: 1234
---バイナリデータ---
```

②レスポンス例

```
HTTP/1.1 200 OK
Content-Type: application/json
X-Message-Id: MSG-20250801-000123
{
  "id": "abc123",
  "fileName": "order_20250801.csv",
  "fileType": "text/csv",
  "size": 1234,
  "uploadedAt": "2025-08-01T08:21:00Z"
}
```

■A-4.メッセージ仕様

メッセージは、EDI におけるフォーマット仕様、ファイル命名ルール、文字コードおよび改行コードの取り扱いに準拠するものとする。

■A-5.認証・認可方式

①認証方式

Web API を EDI 用途で利用する場合、ユーザーによるログイン操作を伴わないマシン間連携が主となるため、OAuth2 の中でも「クライアント ID／クライアントシークレット」によってトークンを発行するクライアントクレデンシャル認証方式を採用する。

②認証フロー

・クライアントは、センターが発行した「client_id」と「client_secret」を使って、認証サーバに「access_token」を要求する。
・認証サーバは資格情報を検証し、短時間有効な「access_token」を返却する。
・クライアントは、そのトークンを使って API を呼び出す

③トークン取得 API

(1)エンドポイント仕様

メソッド：POST
パス：/oauth2/token
Content-Type：application/x-www-form-urlencoded

(2)リクエストパラメータ

grant_type：client_credentials
client_id：発行された ID（センターが事前払い出し）
client_secret：発行されたシークレット（センターが事前払い出し）

(3)リクエスト例

```
POST /oauth2/token HTTP/1.1
Host: auth.edi-center.com
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials&
client_id=client-001&
client_secret=XYZsecret987654321
```

(4)レスポンス例

```
{
  "access_token": "eyJhbGciOiJIUzI1NiIsInR...",
  "token_type": "Bearer",
  "expires_in": 3600
}
```

④認可方式

認可については、複雑なアクセス制御は実施せず、従来の EDI で採用されているセンター側による一元的な管理方式に準拠する。

■A-6.エラー設計

API はエラー発生時に、HTTP ステータスコードとともに以下の形式で JSON レスポンスを返却する。HTTP ステータスコードについては「2.3.技術的要素」の「⑧ステータスコード」を参照されたい。

①エラーレスポンス例

```
{
  "error": {
  "code": "E4001",
  "message": "ファイル ID が指定されていません。",
   "detail": "fileId is required in the path.",
  "retryable": false,
  "timestamp": "2025-08-01T10:15:00+09:00"
  "trace_id": "ab12cd34ef56"
  }
}
```

＜項目説明＞

code：センターで定義された独自エラーコード
message：エラー概要（運用担当者向け）
detail：英語の詳細情報（必要であれば）
retryable：再送可能なエラーかどうか（true/false）
timestamp：発生日時
trace_id：トレース用 ID

※「code」には EDI におけるエラーコードをセットされたい。

■A-7.ログ設計
ログについては、EDI サービスと統合的に管理できるよう、取得および連携を行う。一般的なログの種類は以下の通り。

アクセスログ：API 単位、HTTP リクエストなど
アプリケーションログ：業務処理、ファイル検証など
認証ログ：トークン取得・失敗など
エラーログ：例外発生時に詳細記録

■A-8.テスト計画
一般的なテストの種類は以下の通り。

単体テスト：各モジュールの正当性を検証
結合テスト：モジュール連携の整合性
API テスト：REST API 仕様に対する動作検証
業務シナリオテスト：ファイル送受信のビジネス整合性
異常系テスト：異常入力や失敗時の耐性評価
負荷テスト：同時接続／大容量ファイル等の耐性検証
セキュリティテスト：トークン漏洩、スコープ逸脱などの確認
運用・監視テスト：ログ・監視・通知設計の実働確認

■A-9.API ツールの活用
Web API を安定的かつ効率的に運用するためには、API ゲートウェイに加えて、設計・開発・運用・監視の各段階を支援するツール群を適切に導入することが望ましい。
Web API を内製するにあたってこれらのツールを理解、活用することで、API の品質を維持しつつ、ライフサイクル全体を通じた管理を容易にすることができる。
主なカテゴリと役割を以下に示す。

区分	主なツール・機能	概要・活用例
API ゲートウェイ	認証・認可、トラフィック制限、アクセス制御、ログ取得、バージョン管理等を行うツール。AWS API Gateway など。	クライアントからのリクエストを一元的に制御し、API 公開を安全に実現する中核コンポーネント。
API 設計・ドキュメント化ツール	API をデプロイする前にデザイン、文書化、テストするための支援を行うツール。 OpenAPI（Swagger）、Postman など。	API 仕様を定義・共有し、設計書と実装を整合させる。クライアント向けドキュメントの自動生成にも利用できる。
テストツール	Postman、Newman、Jmeter など。 ※API 設計・ドキュメント化ツールでカバー可能	API の機能テスト・回帰テストを自動化し、変更時の影響確認を容易にする。
API ライフサイクル管理ツール	API の設計から廃止までのライフサイクル全体を管理するツール。MuleSoft Anypoint	設計、公開、バージョン管理、廃止までの API ライフサイクルを統合的

	Platform、Apigee、WSO2 API Manager など。	に管理する。
API 監視・分析ツール	API のパフォーマンス、使用状況、エラーレポートなどを監視し、分析するためのツール。Datadog、Prometheus、ELK Stack など。	稼働状況や応答性能、利用傾向を可視化し、障害検知や改善活動に活用する。

4.1.2.運用担当者向け

Web API を EDI 用途で提供する際の運用にあたっては、EDI 同様に、日常的な監視や障害対応、データ保全、定期運用作業を計画的に実施することが望ましい。サービスレベルは現在提供中の EDI を踏襲しつつ、Web API の特性を考慮する必要がある。

■B-1.モニタリング・監視

EDI に加えて Web API 基盤も監視対象となるため、監視範囲は従来より広範になることに注意が必要である。システムの稼働状況、リソース使用状況、API 応答時間などを常時監視し、死活監視やジョブ監視に加え、異常値検知や閾値超過時のアラート通知を設定することが重要である。

■B-2.障害対応

システム環境が広範になるため、障害発生時には原因の特定や影響範囲の評価が複雑になる点に注意が必要である。復旧手順をマニュアル化し、再送や代替ルートの利用、関係者への通知など、事前に定義された手順に沿った運用を行うことが重要である。

■B-3.データ保全とリカバリ

通信エラーなどによる処理中断やデータ損失に備え、バックアップと復元手順を整備することが望ましい。データ再投入（リトライ）をどの段階で実施するか？（EDI エンジンで行うか、API 基盤で行うか）を明確にし、障害発生時の代替処理を含むリカバリ手順を設計段階で検討しておくことが重要である。リカバリ手順は複雑化を避け、可能な限りシンプルな仕組みにしておくことが望ましい。

■B-4.定期運用業務

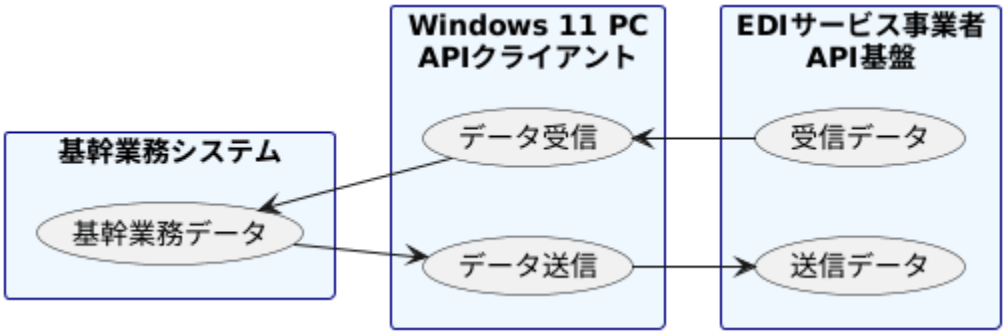
日常的な運用業務として、ユーザーからの問い合わせ対応やファイル再送処理を含む運用補助作業を計画的に実施することが求められる。あわせて、ログ確認や証跡管理、システムのメンテナンス作業も定期運用業務の一環として位置付けるとよい。さらに、ユーザー自身による照会を可能とする証跡管理機能を提供し、センター側担当者の運用負荷を低減する仕組みを整備しておくことが望ましい。

4.2.クライアント側

4.2.1.システム開発者向け

クライアント側環境は、センター側が提供する Web API を利用するための実行環境や運用機能を準備することが求められる。ここでは、一般的に入手が容易な環境を想定し、構成例を提示する。また、EDI 通信プロトコルが備えている機能をクライアント自身で実装・運用する必要がある点を考慮する。

以下に、一般的に必要となる主な要素を示す。
（実際の環境や要件に応じて追加・変更が必要な場合がある）



＜想定環境＞

OS	Windows 11 Professional
ブラウザ	Google Chrome
開発言語	Python
その他補足情報	API クライアントは基幹業務システムと切り離す、連携する。

■C-1.機能

①開発環境・実行環境

- Windows 11 上に Python 環境を構築し、API 実行に必要なライブラリ（例：requests、httpx）を導入する。
- ライブラリのバージョン管理を行い、API 仕様変更やセキュリティ更新に備える。
- API クライアント処理（リクエスト／レスポンス処理、エラー処理）を実装する。

②API クライアント処理

- リクエスト／レスポンス構造のバリデーション（JSON スキーマチェック等）を行う。
- HTTP ステータスや業務エラーコードに応じたエラーハンドリングを設計する。
- 通信エラーやタイムアウト発生時のリトライ方針を明確化する。

・タイムアウト設定や接続プール管理を行い、高負荷時の安定性を確保する。

③認証・認可
・OAuth 2.0 による認証処理を実装し、トークン取得および自動更新を行う。
・アクセストークンのキャッシュや期限管理を適切に行う。
・センター側が提供する EDI における権限管理に従い、必要に応じて適切な情報を提供する。

④通信・セキュリティ対応
・HTTPS（TLS1.2 以上）による暗号化通信を行い、サーバ証明書の正当性を検証する。
・証明書や認証情報を安全に保管し、期限切れや漏洩に備える。
・入力値検証やインジェクション対策、受信ファイルのウイルスチェックなどを実施する。

⑤トランスレータ
・自社業務システムへのデータ連携を考慮し、フォーマット変換や各種コード変換を行うトランスレータの導入を検討する。

⑥データ保全・ログ取得
・送受信したデータは適切なエリアに必要な期間保存する。あわせてハッシュ値による改ざん検知を行う。
・送信ログ、取引番号、リクエスト ID などを記録し、センター側のログと突き合わせて証跡を確保する。
・ログは利用者自身が参照可能な形で管理し、トラブルシューティング等に活用する。

■C-2.リクエスト仕様
以下に、シンプルな送受信サンプルコードを提示する。

①ポイント
(1)XML ファイル添付
・multipart/form-data を使用してファイルを送信する。
(2)ログ管理
・成功・失敗・エラーをログに記録し、MessageID で追跡可能とする。
(3)リトライなし
・単回送信のみを前提とする。
(4)安全な通信
・HTTPS と OAuth2.0 を利用する。
(5)運用補助
・ファイル存在チェックを行い、将来的にハッシュ検証や改ざんチェックを追加可能な設計とする。

あくまでサンプルであり、実際の環境や要件に応じて適宜改変して利用すること。

---

②Python によるデータ受信サンプルコード

```
import requests
from datetime import datetime
import uuid
import os

# ==========================
# 設定値（事前に EDI サービス事業者から提供される情報を設定）
# ==========================
API_BASE_URL = "https://api.ediservice.example.com/v1"
CLIENT_ID = "client_id"
CLIENT_SECRET = "client_secret"
SAVE_DIR = "received_orders"
LOG_FILE = "order_receive.log"

os.makedirs(SAVE_DIR, exist_ok=True)

# ==========================
# アクセストークン取得
# ==========================
def get_access_token():
    payload = {
        "grant_type": "client_credentials",
        "client_id": CLIENT_ID,
        "client_secret": CLIENT_SECRET,
    }
    response = requests.post(f"{API_BASE_URL}/oauth2/token", data=payload)
    response.raise_for_status()
    return response.json()["access_token"]

# ==========================
# ログ出力
# ==========================
def log_result(message_id, status, info):
    timestamp = datetime.now().isoformat()
    with open(LOG_FILE, "a", encoding="utf-8") as f:
        f.write(f"{timestamp} | {message_id} | {status} | {info}¥n")

# ==========================
# ファイル受信
# ==========================
def receive_order_file(access_token):
    message_id = str(uuid.uuid4())  # トラッキング用 ID
```

```
    url = f"{API_BASE_URL}/orders/file/latest"  # ファイル取得エンドポイント
    headers = {
        "Authorization": f"Bearer {access_token}",
        "X-Message-ID": message_id
    }

    try:
        # ファイルをダウンロード
        response = requests.get(url, headers=headers, timeout=30, stream=True)
        if response.status_code == 200:
            file_name =
            f"order_{datetime.now().strftime('%Y%m%d_%H%M%S')}_{message_id}.xml"
            file_path = os.path.join(SAVE_DIR, file_name)
            with open(file_path, "wb") as f:
                for chunk in response.iter_content(chunk_size=8192):
                    f.write(chunk)
            print(f"受信成功: {file_path}")
            log_result(message_id, "SUCCESS", file_path)
            return file_path
        else:
            print(f"受信失敗 (Status={response.status_code})")
            log_result(message_id, "FAIL", response.text)
            return None
        except requests.RequestException as e:
            print(f"受信エラー: {e}")
            log_result(message_id, "ERROR", str(e))
            return None

# ==========================
# メイン処理
# ==========================
if __name__ == "__main__":
    token = get_access_token()
    print("Access Token 取得成功")

    received_file = receive_order_file(token)
    if received_file:
        # 必要に応じて、受信したファイルを解析・処理
        print(f"ファイルを保存しました: {received_file}")
```

---

③Python によるデータ送信サンプルコード

```
import requests
from datetime import datetime
import uuid
import os

# ==========================
#設定値（事前に EDI サービス事業者から提供される情報を設定）
# ==========================
API_BASE_URL = "https://api.ediservice.example.com/v1"
CLIENT_ID = "client_id"
CLIENT_SECRET = "client_secret"
ORDER_FILE = "order.xml"  # 送信する XML ファイルパス
LOG_FILE = "order_send.log"

# ==========================
# アクセストークン取得
# ==========================
def get_access_token():
    payload = {
        "grant_type": "client_credentials",
        "client_id": CLIENT_ID,
        "client_secret": CLIENT_SECRET,
    }
    response = requests.post(f"{API_BASE_URL}/oauth2/token", data=payload)
    response.raise_for_status()
    return response.json()["access_token"]

# ==========================
# ログ出力
# ==========================
def log_result(message_id, status, info):
    timestamp = datetime.now().isoformat()
    with open(LOG_FILE, "a", encoding="utf-8") as f:
        f.write(f"{timestamp} | {message_id} | {status} | {info}¥n")

# ==========================
# ファイル送信
# ==========================
def send_order_file(access_token, file_path):
    message_id = str(uuid.uuid4())  # トラッキング用 ID
    url = f"{API_BASE_URL}/orders/file/upload"  # ファイル送信エンドポイント
    headers = {
```

```
        "Authorization": f"Bearer {access_token}",
        "X-Message-ID": message_id
    }

    if not os.path.exists(file_path):
        print(f"ファイルが存在しません: {file_path}")
        log_result(message_id, "FAIL", "File not found")
        return False

    with open(file_path, "rb") as f:
        files = {
            "file": (os.path.basename(file_path), f, "application/octet-stream")
        }
        try:
            response = requests.post(url, headers=headers, files=files, timeout=30)
            if response.status_code == 200:
                print(f"送信成功: {file_path}")
                log_result(message_id, "SUCCESS", response.text)
                return True
            else:
                print(f"送信失敗 (Status={response.status_code})")
                log_result(message_id, "FAIL", response.text)
                return False
        except requests.RequestException as e:
            print(f"送信エラー: {e}")
            log_result(message_id, "ERROR", str(e))
            return False

# ==========================
# メイン処理
# ==========================
if __name__ == "__main__":
    token = get_access_token()
    print("Access Token 取得成功")

    success = send_order_file(token, ORDER_FILE)
    if success:
        print(f"ファイルを送信しました: {ORDER_FILE}")
```

4.2.2.運用担当者向け

EDI と同様に、Web API を利用した EDI においても、日常的な監視や障害対応、データ保全、定期運用作業が必須である。センター側が提供する Web API 環境の仕様や運用補助機能を活用しつつ、EDI で行われていた運用業務を拡張する形で運用業務を設計することが望ましい。
以下に、考慮すべきポイントを示す。

■D-1.モニタリング・監視
- API 通信状態、応答時間、エラー発生状況を日常的に監視する。
- 通信エラー検知や閾値超過時のアラート通知を設定し、早期対応が可能な仕組みを整備する。
- 各種ログを取得し、原因特定や再送対応に役立てる。

■D-2.定期運用業務
- データ送受信はジョブスケジュール機能を活用し、自動実行環境を検討する。
- バックエンドの基幹業務システムとの連携を考慮する。
- ユーザーからの問い合わせ対応やファイル再送処理など、日常的な運用補助業務を計画的に実施する。
- ログ確認や証跡管理、アプリケーションや依存ライブラリの更新作業を定期運用業務に組み込む。
- ユーザー自身が照会可能な証跡管理機能を提供し、運用担当者の負荷を低減する仕組みを整備する。

■D-3.障害対応・リカバリ
- 障害発生時の原因特定や影響範囲の評価を迅速に行える手順を準備する。
- 通信エラーや認証エラーに対するリトライ、代替手段の利用などを事前に検討し、決められた手順に従って対応する。
- センター側サポート窓口への問い合わせや関係者通知の手順を整備する。

■D-4.運用作業・定期更新
- Windows Update やライブラリの更新を計画的に実施し、脆弱性を放置しない。
- センター側から提供される証明書更新や API バージョン変更に合わせて、事前に検証・対応を行う。

4.3.テスト項目

Web API を利用した EDI 環境を安定稼働させるために、本番運用前に多面的なテストを実施することが望ましい。
以下に、開発段階で考慮すべき主なテスト項目例を整理する。

①通信レベルテスト
- API エンドポイントに HTTPS 接続できるかを確認する。
- HTTP ステータスコード（200、400、401、500 など）が仕様通り返却されるかを検証する。
- リクエスト／レスポンスのヘッダ情報（Content-Type、Authorization）が正しく設定・取得できるかを確認する。

②セキュリティテスト
・OAuth2.0 のアクセストークン取得／更新が正しく動作するかを確認する。
・有効期限切れや不正なトークン使用時に適切に拒否されるかを検証する。
・サーバ証明書の正当性（有効期限、中間証明書の連鎖）を確認する。
・通信データの盗聴・改ざん防止が TLS で担保されているかをテストする。

③機能テスト
・正常なファイル送信で、サーバ側がデータを受理することを確認する。
・受信 API から取得したデータが正しく格納・利用できるかを検証する。
・取引番号や MessageID による重複検出が機能するかをテストする。
・ファイル添付（multipart/form-data）の形式が仕様通りであるかを確認する。

④異常系テスト
・ネットワーク切断やタイムアウト時にエラーが検知され、ログに記録されるかを確認する。
・不正な形式（タグ欠落、文字コード不一致など）でのファイル送信時にエラー応答が返るかを検証する。
・誤った認証情報でアクセスした場合、アクセス拒否されるかを確認する。
・サーバ側障害時（HTTP 500）にクライアント側が想定通りのエラーハンドリングを行うかを確認する。

⑤性能・負荷テスト
・一定サイズ以上のファイルの送受信時間を計測する。
・大量データを連続送受信した際に応答遅延や処理落ちが発生しないかを検証する。
・同時接続（例：50 クライアント同時リクエスト）時のサーバ応答を確認する。
・長時間稼働させた際にメモリリークやリソース枯渇がないかを検証する。

5.Appendix

5.1.導入時チェックシート

Web API を EDI 用途で導入・運用する際に、設計・実装・運用の各フェーズで確認すべき事項の一例を、参考としてチェックシート形式で整理、掲載する。
センター側・クライアント側の双方が共通の観点で準備状況を確認することで、導入プロジェクト全体の品質および運用安定性の確保につながれば幸いである。

なお、Web API 技術やセキュリティ要件は将来的に変更される可能性があるため、定期的に内容を見直し、最新の運用実態に即した形に更新することが望ましい。

1. 目的・要件定義			
	1.1. 導入目的の明確化	従来の EDI システムからの移行か？	
	1.2. 対象業務範囲	対象業務は？	
	1.3. 対象取引先	全ての取引先 or 特定の取引先？	
		取引先の Web API 対応状況はどうか？	
2. 技術要件・アーキテクチャ			
	2.1. API 提供方式	RESTful API での提供に適しているか？	
		ファイル送受信の具体的なエンドポイントは？（例: /upload, /download）	
	2.2. 通信プロトコル	HTTPS で要件を満たしているか？	
	2.3. データフォーマット	ファイル形式は何か？（EDI ファイル形式など）	
		ファイルサイズ、送受信の頻度、データ量は？	
		文字コードは？	
	2.4. 内部連携	Web API が受け取ったデータと既存の基幹システムや EDI システムとの連携方法は？	
		EDI システムとの変換（マッピング）ツールや機能は必要か？	
	2.5. パフォーマンス要件	ピーク時のリクエスト数、データ転送速度の目標値は？	
		応答時間の目標値は？	
3. セキュリティ			
	3.1. 認証方式	OAuth による認証で要件を満たしているか？	
	3.2. 認可方式	ロールベースアクセス制御（RBAC）か？	
		API キーやクライアント ID ごとにアクセス可能な範囲を限定するか？	
	3.3. 通信の暗号化	HTTPS を必須として問題ないか？	
		TLS バージョン 1.2 以上で要件を満たしているか？	
	3.4. データ改ざん検知・防止	入力値検証のルールは明確か？（必須）	
		ハッシュ値やデジタル署名によるデータの完全性検証を導入するか？	
	3.5. 脆弱性対策	SQL インジェクション、XSS などの一般的な Web 脆弱性対策は適切か？	

		定期的な脆弱性診断は実施するか？	
	3.6. アクセス制御・ネットワークセキュリティ	ファイアウォール、WAF の導入は考慮されているか？	
		IP アドレス制限は行うか？	
		DDoS 攻撃対策は考慮されているか？	
4. 運用・保守			
	4.1. ロギング	API のアクセスログ、エラーログ、処理ログは適切に取得されているか？	
		ログの保存期間、保存場所、アクセス権限は明確か？	
	4.2. モニタリング	API の稼働状況（稼働率、応答時間など）は監視されているか？	
		エラー発生時やパフォーマンス低下時のアラート通知体制は確立されているか？	
	4.3. エラーハンドリング	エラー時のレスポンスコード（HTTP ステータスコード）とメッセージは明確か？	
		リトライ処理やエラー通知の仕組みはどうか？	
	4.4. バージョン管理	API のバージョンアップ計画は考慮されているか？	
		旧バージョンのサポート期間は？	
	4.5. 障害対応	障害発生時の連絡体制、復旧手順、取引先への通知方法は明確か？	
		BCP/DR（事業継続計画/災害復旧）は考慮されているか？	
	4.6. 運用体制	API を運用・保守する部門は決まっているか？	
		SLA（サービス品質保証）は考慮されているか？	
5. 取引先対応			
	5.1. ドキュメント提供	API 仕様書、利用マニュアルなどを整備・提供できるか？	
		開発者向けポータルサイトの提供は考慮されているか？	
	5.2. テスト環境	取引先が接続テストを行えるステージング環境を提供できるか？	
	5.3. 移行計画	従来の EDI から Web API 型 EDI への移行スケジュールと、取引先との調整方法は？	
		並行稼働期間は設けるか？	
	5.4. サポート体制	取引先からの問い合わせ対応、技術サポート体制は構築	

		されているか？	
6. 費用			
	6.1. 開発費用	API サーバー、連携機能の開発費用	
		ドキュメント、テスト環境整備費用	
	6.2. インフラ費用	サーバー（クラウド/オンプレミスか）、ネットワーク、ストレージ費用	
		セキュリティ関連製品（WAF など）の費用	
	6.3. 運用・保守費用	モニタリングツールなどの費用	
		人件費	
	6.4. ライセンス費用	ミドルウェアやソフトウェアを利用する場合のライセンス費用	
	6.5. 通信費用	インターネット回線費用	
7. 法規制・コンプライアンス			
	7.1. 電子帳簿保存法対応	電子取引データの保存は必要か？	
	7.2. 個人情報保護法対応	個人情報を取り扱う場合、適切なセキュリティ対策が講じられているか？	
	7.3. 業界固有の規制	業界団体等が定める EDI に関するガイドラインや要件に適合するか？ 改定は必要か？	
	7.4. 契約	取引先との間で EDI 利用に関する契約や覚書を締結するか？ 締結済みの場合、改定は必要か？	

EDI 用途における Web API 利用ガイドライン

2026 年 1 月　発行

インターネット EDI 普及推進協議会
Japan internet EDI Association (JiEDIA)

本資料に関する問い合わせは、下記までお願いします。

JiEDIA 事務局：一般社団法人 情報サービス産業協会
https://www.jisa.or.jp/tabid/2821/Default.aspx

〒101-0047
東京都千代田区内神田 2-3-4
S-GATE 大手町北 6F
TEL：03-5289-7651（代表）
FAX：03-5289-7653