API Versioning Strategies

1. The Real Problem This Exists to Solve

When you evolve an API, existing clients break if you make incompatible changes. API versioning allows you to introduce new features, fix design mistakes, and improve your API without breaking existing integrations. Without versioning, every change risks breaking production systems that depend on your API.

Real production scenario:

- Payment API used by 500 mobile apps (takes months for users to update)
- Current API: `POST /payments { amount: 100 }`
- You need to add currency support: `POST /payments { amount: 100, currency: "USD" }`
- **Without versioning:**
  - Deploy breaking change
  - 500 apps break immediately
  - Customers can't make payments
  - Emergency rollback at 2am
  - Lost revenue, angry partners
  - Can't improve API without breaking clients
- **With versioning:**
  - Deploy v2: `POST /v2/payments { amount: 100, currency: "USD" }`
  - v1 still works: `POST /v1/payments { amount: 100 }` (defaults currency to USD)
  - Old apps continue working
  - New apps adopt v2 over 6 months
  - Eventually deprecate v1 after migration complete
  - Smooth evolution, zero downtime

The fundamental problem: APIs are contracts between producer (server) and consumer (clients). Changing contracts breaks integrations. Versioning allows multiple contract versions to coexist, enabling gradual migration and backward compatibility.

Without API versioning:

- Breaking changes break all clients
- Can't evolve API without coordination
- Emergency rollbacks
- Lost business
- Technical debt accumulates (afraid to change)

With API versioning:

- Multiple versions coexist
- Gradual client migration
- Backward compatibility
- Controlled deprecation
- Increased complexity and maintenance

2. The Naive / Incorrect Approaches (IMPORTANT)

❌ Incorrect Approach #1: No Versioning (Hope Nobody Breaks)

```
// Incorrect: Just change the API and hope for the best
// Old API (before)
app.post('/api/payments', async (req, res) => {
  const { amount } = req.body;

  // Process payment in USD (assumed)
  await processPayment(amount, 'USD');

  res.json({ success: true });
});

// New API (after) - BREAKING CHANGE
app.post('/api/payments', async (req, res) => {
  const { amount, currency } = req.body; // currency now required!

  if (!currency) {
    return res.status(400).json({ error: 'currency is required' });
  }

  await processPayment(amount, currency);

  res.json({ success: true });
});
```

Why it seems reasonable:

- Simple, no versioning complexity
- Forces everyone to latest version
- Less code to maintain

How it breaks:

```
Mobile App v1.0 (in production):
  POST /api/payments
  Body: { amount: 100 }

Server response after deployment:
  400 Bad Request: { error: 'currency is required' }

Result:
  - All payments fail
  - 500 apps broken simultaneously
  - Customer complaints flood in
  - Emergency rollback required
```

Production symptoms:

- Spike in 400 errors after deployment
- Support tickets: "Payments not working"
- Revenue drops to zero
- Partner integrations break
- Emergency hotfix at 3am

❌ Incorrect Approach #2: Optional Parameters Everywhere (Ambiguous Behavior)

```
// Incorrect: Make everything optional to avoid breaking changes
app.post('/api/payments', async (req, res) => {
  const {
    amount,           // optional? required?
    currency,         // optional (defaults to USD)
    paymentMethod,    // optional (defaults to card)
    description,      // optional
    metadata          // optional
  } = req.body;

  // Unclear what's required vs optional
  // Defaults scattered throughout code
  const finalCurrency = currency || 'USD';
  const finalMethod = paymentMethod || 'card';

  await processPayment(amount || 0, finalCurrency, finalMethod);

  res.json({ success: true });
});
```

Why it seems reasonable:

- Backward compatible (old clients work)
- No versioning needed
- Flexible

How it breaks:

```
// Ambiguous behavior
POST /api/payments { amount: 100 }
  → What currency? USD assumed, but not documented

POST /api/payments { amount: 100, currency: null }
  → Does null mean USD or error?

POST /api/payments {}
  → amount: 0? Or error? Unclear!

// Client confusion
const client = new PaymentClient();
await client.createPayment({ amount: 100 }); // Works, but what currency?
```

Production symptoms:

- Unclear API contract
- Clients guess defaults incorrectly
- Payments in wrong currency
- Support: "What fields are required?"
- Documentation out of sync with code

- Technical debt accumulates

❌ Incorrect Approach #3: Date-Based Versioning in URL (Maintenance Nightmare)

```
// Incorrect: Version by date
app.post('/api/2024-01-15/payments', async (req, res) => {
  // Implementation for 2024-01-15 version
});

app.post('/api/2024-06-20/payments', async (req, res) => {
  // Implementation for 2024-06-20 version
});

app.post('/api/2024-09-10/payments', async (req, res) => {
  // Implementation for 2024-09-10 version
});

// 50+ date versions later...
app.post('/api/2026-02-03/payments', async (req, res) => {
  // Latest version
});
```

Why it seems reasonable:

- Clear when version was released
- Prevents conflicts (unique dates)
- Easy to track changes over time

How it breaks:

- 50+ routes for same endpoint
- Code duplication everywhere
- Bug fix must be backported to all versions
- Impossible to deprecate old versions
- Clients confused which version to use

Production symptoms:

```
// Chaos
GET /api/2024-01-15/payments  → works
GET /api/2024-02-20/payments  → 404 (no changes that day)
GET /api/2024-06-20/payments  → works
GET /api/2024-07-01/payments  → 404

// Which version should I use?
// What changed between versions?
// How to migrate from 2024-01-15 to 2024-09-10?
```

❌ Incorrect Approach #4: Breaking Changes in Patch Versions

```typescript
// Incorrect: Semantic versioning but breaking changes in patches
// v1.0.0 (initial)
interface PaymentRequest {
  amount: number;
}

// v1.0.1 (patch) - BREAKING CHANGE IN PATCH VERSION
interface PaymentRequest {
  amount: number;
  currency: string; // Required! Breaking change in patch version
}

// v1.1.0 (minor) - Another breaking change
interface PaymentRequest {
  amount: number;
  currency: string;
  paymentMethod: 'card' | 'bank'; // Required! Breaking change in minor version
}
```

Why it seems reasonable:

- Uses semantic versioning (looks professional)
- Incremental version numbers
- Clear progression

How it breaks:

- Semantic versioning promises:
  - Patch (1.0.x): Bug fixes only, no breaking changes
  - Minor (1.x.0): New features, backward compatible
  - Major (x.0.0): Breaking changes allowed
- Adding required field in patch/minor is BREAKING
- Clients auto-update patches → break

Production symptoms:

```
Client config: "api-version": "^1.0.0" (any 1.x version)
Deploy v1.0.1 → client auto-updates → breaks
Client: "You said backward compatible!"
```

❌ Incorrect Approach #5: Version in Request Body (Client Must Know)

```typescript
// Incorrect: Version specified in request body
app.post('/api/payments', async (req, res) => {
  const { version, amount, currency } = req.body;

  if (version === 1) {
    // v1 behavior (no currency)
    await processPayment(amount, 'USD');
  } else if (version === 2) {
    // v2 behavior (currency required)
```

```
    await processPayment(amount, currency);
  } else {
    return res.status(400).json({ error: 'Invalid version' });
  }

  res.json({ success: true });
});

// Client usage
POST /api/payments
Body: { version: 2, amount: 100, currency: "USD" }
```

Why it seems reasonable:

- Single endpoint
- Client controls version
- Simple routing

How it breaks:

- Version is data, not metadata (shouldn't be in body)
- Client can accidentally omit version
- No URL-based caching
- Can't use HTTP headers for versioning
- Awkward API design

Production symptoms:

```
// Client forgets version
POST /api/payments { amount: 100, currency: "USD" }
// What version? Default to v1? Error?

// Middleware can't route by version (body not parsed yet)
// Logging/monitoring harder (all requests go to same endpoint)
```

3. Correct Mental Model (How It Actually Works)

API versioning creates parallel implementations of endpoints, allowing old and new versions to coexist. Clients specify which version they want via URL, header, or content negotiation. Old versions are maintained until all clients migrate, then deprecated.

The Version Lifecycle

```
1. v1 released (initial version)
2. v2 developed (breaking changes needed)
3. v2 released, v1 maintained (both coexist)
4. v1 deprecated (6-12 month warning)
5. v1 sunset (remove after migration period)
```

Version Selection Strategies

URL versioning: `/v1/payments`, `/v2/payments` **Header versioning:** `Accept: application/vnd.myapi.v2+json` **Query parameter:** `/payments?version=2` **Content negotiation:** `Accept: application/json; version=2`

Backward Compatibility

```
Breaking changes (require new version):
  - Remove field
  - Rename field
  - Change field type
  - Add required field
  - Change behavior

Non-breaking changes (can add to existing version):
  - Add optional field
  - Add new endpoint
  - Deprecate field (but keep working)
```

4. Correct Design & Algorithm

Strategy 1: URL Path Versioning (Most Common)

```
GET  /v1/users
POST /v1/payments
GET  /v2/users      ← New version
POST /v2/payments   ← New version
```

Strategy 2: Header Versioning (RESTful)

```
GET /users
Header: API-Version: 1

GET /users
Header: API-Version: 2
```

Strategy 3: Content Negotiation (True REST)

```
GET /users
Accept: application/vnd.myapi.v1+json

GET /users
Accept: application/vnd.myapi.v2+json
```

5. Full Production-Grade Implementation

```
import express, { Request, Response, NextFunction } from 'express';

/**
```

```
 * Version-aware request
 */
interface VersionedRequest extends Request {
  apiVersion: number;
}

/**
 * Middleware: Extract API version from URL
 */
function extractVersionFromURL(req: VersionedRequest, res: Response, next:
NextFunction) {
  const match = req.path.match(/^\/v(\d+)\//);

  if (match) {
    req.apiVersion = parseInt(match[1]);
  } else {
    // Default to latest version if no version specified
    req.apiVersion = 2;
  }

  next();
}

/**
 * Middleware: Extract API version from header
 */
function extractVersionFromHeader(req: VersionedRequest, res: Response, next:
NextFunction) {
  const versionHeader = req.headers['api-version'] as string;

  if (versionHeader) {
    req.apiVersion = parseInt(versionHeader);
  } else {
    req.apiVersion = 1; // Default to v1
  }

  next();
}

/**
 * Version deprecation warnings
 */
const DEPRECATED_VERSIONS = [1];
const SUNSET_DATE: Record<number, string> = {
  1: '2026-06-01', // v1 will be removed on this date
};

function addDeprecationHeaders(req: VersionedRequest, res: Response, next:
NextFunction) {
  const version = req.apiVersion;

  if (DEPRECATED_VERSIONS.includes(version)) {
```

```
    res.setHeader('Deprecation', 'true');
    res.setHeader('Sunset', SUNSET_DATE[version]);
    res.setHeader('Link', '</docs/migration>; rel="deprecation"');
  }

  next();
}

/**
 * Payment request types (versioned)
 */
namespace PaymentV1 {
  export interface CreateRequest {
    amount: number;
  }

  export interface PaymentResponse {
    id: string;
    amount: number;
    status: string;
  }
}

namespace PaymentV2 {
  export interface CreateRequest {
    amount: number;
    currency: string;
    paymentMethod?: 'card' | 'bank' | 'wallet';
  }

  export interface PaymentResponse {
    id: string;
    amount: number;
    currency: string;
    paymentMethod: string;
    status: string;
    createdAt: string;
  }
}

/**
 * Payment service with versioned logic
 */
class PaymentService {
  /**
   * Create payment v1 (legacy)
   */
  async createPaymentV1(request: PaymentV1.CreateRequest):
Promise<PaymentV1.PaymentResponse> {
    const { amount } = request;

    // v1 behavior: assume USD
```

```
    const payment = await this.processPayment(amount, 'USD', 'card');

    return {
      id: payment.id,
      amount: payment.amount,
      status: payment.status,
    };
  }

  /**
   * Create payment v2 (current)
   */
  async createPaymentV2(request: PaymentV2.CreateRequest):
Promise<PaymentV2.PaymentResponse> {
    const { amount, currency, paymentMethod = 'card' } = request;

    // v2 behavior: explicit currency
    const payment = await this.processPayment(amount, currency, paymentMethod);

    return {
      id: payment.id,
      amount: payment.amount,
      currency: payment.currency,
      paymentMethod: payment.paymentMethod,
      status: payment.status,
      createdAt: payment.createdAt.toISOString(),
    };
  }

  /**
   * Internal implementation
   */
  private async processPayment(amount: number, currency: string, method: string) {
    // Actual payment processing
    return {
      id: `pay_${Date.now()}`,
      amount,
      currency,
      paymentMethod: method,
      status: 'completed',
      createdAt: new Date(),
    };
  }
}

const paymentService = new PaymentService();

/**
 * URL-based versioning (Strategy 1)
 */
const app = express();
```

```
// v1 endpoints (deprecated)
app.post('/v1/payments', async (req: VersionedRequest, res) => {
  try {
    const payment = await paymentService.createPaymentV1(req.body);

    // Add deprecation warning
    res.setHeader('Deprecation', 'true');
    res.setHeader('Sunset', '2026-06-01');
    res.setHeader('Link', '</docs/v2-migration>; rel="deprecation"');

    res.json(payment);
  } catch (error: any) {
    res.status(500).json({ error: error.message });
  }
});

// v2 endpoints (current)
app.post('/v2/payments', async (req, res) => {
  try {
    const { amount, currency, paymentMethod } = req.body;

    if (!currency) {
      return res.status(400).json({ error: 'currency is required' });
    }

    const payment = await paymentService.createPaymentV2({
      amount,
      currency,
      paymentMethod,
    });

    res.json(payment);
  } catch (error: any) {
    res.status(500).json({ error: error.message });
  }
});

/**
 * Header-based versioning (Strategy 2)
 */
const app2 = express();
app2.use(extractVersionFromHeader);
app2.use(addDeprecationHeaders);

app2.post('/payments', async (req: VersionedRequest, res) => {
  try {
    const version = req.apiVersion;

    if (version === 1) {
      const payment = await paymentService.createPaymentV1(req.body);
      res.json(payment);
    } else if (version === 2) {
```

```
      const { amount, currency, paymentMethod } = req.body;

      if (!currency) {
        return res.status(400).json({ error: 'currency is required in v2' });
      }

      const payment = await paymentService.createPaymentV2({
        amount,
        currency,
        paymentMethod,
      });

      res.json(payment);
    } else {
      res.status(400).json({ error: `Unsupported API version: ${version}` });
    }
  } catch (error: any) {
    res.status(500).json({ error: error.message });
  }
});

/**
 * Routing abstraction for cleaner code
 */
class VersionedRouter {
  private handlers: Map<number, express.RequestHandler> = new Map();

  version(version: number, handler: express.RequestHandler) {
    this.handlers.set(version, handler);
    return this;
  }

  build(): express.RequestHandler {
    return async (req: VersionedRequest, res, next) => {
      const handler = this.handlers.get(req.apiVersion);

      if (!handler) {
        return res.status(400).json({
          error: `API version ${req.apiVersion} not supported`,
          supportedVersions: Array.from(this.handlers.keys()),
        });
      }

      return handler(req, res, next);
    };
  }
}

// Usage
app2.use(extractVersionFromHeader);

app2.post('/payments',
```

```
  new VersionedRouter()
    .version(1, async (req, res) => {
      const payment = await paymentService.createPaymentV1(req.body);
      res.json(payment);
    })
    .version(2, async (req, res) => {
      const { amount, currency, paymentMethod } = req.body;

      if (!currency) {
        return res.status(400).json({ error: 'currency is required' });
      }

      const payment = await paymentService.createPaymentV2({
        amount,
        currency,
        paymentMethod,
      });

      res.json(payment);
    })
    .build()
);

/**
 * TypeScript SDK with versioning
 */
class PaymentClientV1 {
  constructor(private apiKey: string, private baseURL: string) {}

  async createPayment(amount: number): Promise<PaymentV1.PaymentResponse> {
    const response = await fetch(`${this.baseURL}/v1/payments`, {
      method: 'POST',
      headers: {
        'Authorization': `Bearer ${this.apiKey}`,
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({ amount }),
    });

    return response.json();
  }
}

class PaymentClientV2 {
  constructor(private apiKey: string, private baseURL: string) {}

  async createPayment(
    amount: number,
    currency: string,
    paymentMethod?: 'card' | 'bank' | 'wallet'
  ): Promise<PaymentV2.PaymentResponse> {
    const response = await fetch(`${this.baseURL}/v2/payments`, {
```

```
      method: 'POST',
      headers: {
        'Authorization': `Bearer ${this.apiKey}`,
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({ amount, currency, paymentMethod }),
    });

    // Check for deprecation warnings
    const deprecation = response.headers.get('Deprecation');
    if (deprecation) {
      console.warn(`[DEPRECATION] This API version is deprecated. Sunset date:
${response.headers.get('Sunset')}`);
    }

    return response.json();
  }
}

/**
 * Adapter pattern for backward compatibility
 */
class PaymentAdapter {
  /**
   * Convert v1 request to v2 format
   */
  static v1ToV2Request(v1Request: PaymentV1.CreateRequest): PaymentV2.CreateRequest
{
    return {
      amount: v1Request.amount,
      currency: 'USD', // Default for v1
      paymentMethod: 'card', // Default for v1
    };
  }

  /**
   * Convert v2 response to v1 format
   */
  static v2ToV1Response(v2Response: PaymentV2.PaymentResponse):
PaymentV1.PaymentResponse {
    return {
      id: v2Response.id,
      amount: v2Response.amount,
      status: v2Response.status,
      // Strip v2-only fields
    };
  }
}

// Implementation: v1 endpoint delegates to v2 logic
app.post('/v1/payments-delegated', async (req, res) => {
  // Convert v1 request to v2
```

```
  const v2Request = PaymentAdapter.v1ToV2Request(req.body);

  // Use v2 logic
  const v2Response = await paymentService.createPaymentV2(v2Request);

  // Convert v2 response back to v1
  const v1Response = PaymentAdapter.v2ToV1Response(v2Response);

  res.json(v1Response);
});

/**
 * GraphQL versioning (field-level)
 */
const typeDefs = `
  type Payment {
    id: ID!
    amount: Float!
    currency: String!

    # Deprecated field (v1)
    amountUSD: Float @deprecated(reason: "Use amount + currency instead")
  }

  input CreatePaymentInput {
    amount: Float!
    currency: String!
  }

  type Mutation {
    createPayment(input: CreatePaymentInput!): Payment!
  }
`;

/**
 * Migration guide generator
 */
function generateMigrationGuide(fromVersion: number, toVersion: number): string {
  if (fromVersion === 1 && toVersion === 2) {
    return `
# Migration Guide: v1 → v2

## Breaking Changes

### 1. Currency is now required
- **Before (v1):** POST /v1/payments { amount: 100 }
- **After (v2):** POST /v2/payments { amount: 100, currency: "USD" }

### 2. Response includes additional fields
- **Before (v1):** { id, amount, status }
- **After (v2):** { id, amount, currency, paymentMethod, status, createdAt }
```

```
## Migration Steps

1. Update your code to include \`currency\` in all payment requests
2. Update response parsing to handle new fields
3. Test with v2 endpoint: POST /v2/payments
4. Deploy to production
5. Monitor for errors

## Timeline

- **Today:** v1 still works (deprecated)
- **2026-06-01:** v1 will be removed
- **Action required:** Migrate by 2026-05-31
    `;
  }

  return '';
}
```

6. Correct Usage Patterns (Where This Shines)

Pattern 1: Mobile App Compatibility

```
// iOS app v1.0 (released 6 months ago)
const client = new PaymentClientV1(apiKey, baseURL);
await client.createPayment(100); // Works with /v1/payments

// iOS app v2.0 (new release)
const client = new PaymentClientV2(apiKey, baseURL);
await client.createPayment(100, 'EUR', 'bank'); // Uses /v2/payments
```

Pattern 2: Partner API Integration

```
// Partner A (uses v1, won't update)
POST /v1/payments { amount: 100 }

// Partner B (migrated to v2)
POST /v2/payments { amount: 100, currency: "USD" }
```

Pattern 3: Gradual Rollout

```
Week 1: Release v2, v1 still works
Week 4: 10% of clients migrated to v2
Week 12: 50% of clients migrated
Week 24: 90% of clients migrated
Week 26: Deprecate v1 (sunset warning)
Week 52: Remove v1 (all clients migrated)
```

7. Failure Modes & Edge Cases

Forgotten Version

Problem: Client doesn't specify version, gets unexpected behavior.

Mitigation: Require version (error if missing) or default to latest with warning.

Version Explosion

Problem: 10+ versions maintained simultaneously.

Mitigation: Aggressive deprecation policy (max 2-3 versions at once).

Inconsistent Versioning

Problem: Some endpoints v2, others v1 (mixed versions).

Mitigation: Version entire API surface together.

8. Performance Characteristics & Tradeoffs

Code Complexity

- **Single version:** 100 LOC
- **2 versions:** 180 LOC (routing + adapters)
- **3+ versions:** Complexity explodes

Maintenance Cost

- **Per version:** Additional testing, bug fixes, documentation
- **Recommendation:** Support max 2-3 versions concurrently

Migration Window

- **Too short:** Clients can't migrate in time
- **Too long:** Maintaining old versions forever
- **Sweet spot:** 6-12 months

9. Foot-Guns & Common Mistakes (DO NOT SKIP)

Mistake 1: Breaking Changes in Minor Versions

Fix: Follow semantic versioning strictly (major = breaking).

Mistake 2: No Deprecation Warnings

Fix: Add `Deprecation: true` and `Sunset: 2026-06-01` headers.

Mistake 3: Supporting Too Many Versions

Fix: Max 2-3 versions, aggressive deprecation.

Mistake 4: No Migration Guide

Fix: Publish detailed migration guide with examples.

Mistake 5: Version in Request Body

Fix: Use URL path or headers (metadata, not data).

10. When NOT to Use This (Anti-Patterns)

Internal APIs

If all clients are controlled by you, coordinate breaking changes directly.

Prototypes

Don't version early-stage APIs (iterate fast, break often).

Event Streams

Use schema evolution (Avro, Protobuf) instead of API versioning.

11. Related Concepts (With Contrast)

Schema Evolution

Difference: API versioning duplicates endpoints, schema evolution evolves single schema.

Feature Flags

Difference: Feature flags toggle behavior, versioning creates parallel implementations.

Contract Testing

Related: Ensures versions don't break accidentally (Pact, Spring Cloud Contract).

12. Production Readiness Checklist

Versioning Strategy

- [ ] Choose versioning scheme (URL, header, or content negotiation)
- [ ] Document version in all responses
- [ ] Add deprecation headers (Deprecation, Sunset, Link)
- [ ] Publish API changelog

Version Management

- [ ] Max 2-3 concurrent versions
- [ ] Deprecation policy (6-12 month window)
- [ ] Sunset dates communicated 6+ months ahead
- [ ] Migration guide for each version upgrade

Monitoring

- [ ] Track usage by version (% of requests per version)
- [ ] Alert when deprecated version usage >5%
- [ ] Dashboard showing adoption of new versions

- [ ] Log when clients hit deprecated endpoints

Testing

- [ ] Integration tests for each version
- [ ] Contract tests (Pact) for version compatibility
- [ ] Test migration from v1 → v2
- [ ] Test that v1 still works after v2 release