API Versioning & Changes

Introduction

This guide outlines our approach to API versioning, managing changes, and supporting clients through API evolution. We use semantic versioning (major.minor.patch) to communicate the scope and impact of changes while ensuring a smooth transition between versions.

Version Control Strategy

Our versioning strategy ensures backward compatibility while enabling API evolution. We maintain clear upgrade paths and support multiple versions simultaneously to accommodate different client needs.

Versioning Schema

Defines how we structure and identify different versions of the API using semantic versioning.

```
interface APIVersion {
  major: number;    // Breaking changes
  minor: number;    // Backward-compatible features
  patch: number;    // Bug fixes
  status: 'alpha' | 'beta' | 'stable' | 'deprecated';
}

// Example: v2.1.3
const VERSION_PATTERN = /^v(\d+)\.(\d+)\.(\d+)$/;
```

URI Versioning

Implements version routing in API endpoints through URL-based versioning.

```
// Base URL structure
const API_URL = 'https://api.example.com/v{major}/{resource}';

// Version routing middleware
const versionRouter = (req: Request, res: Response, next: NextFunction) => {
  const version = req.path.split('/')[1]; // e.g., 'v2'
  const [major] = version.match(VERSION_PATTERN) || [];

  req.apiVersion = {
    major: parseInt(major),
    implementation: getVersionImplementation(major)
  };
```

```
  next();
};
```

Change Classifications

We classify API changes based on their impact on existing clients, which determines version increments and communication strategies.

1. Breaking Changes

Changes that require client updates and trigger a major version increment.

```
interface BreakingChange {
  type: 'BREAKING';
  changes: {
    removedFields?: string[];
    modifiedFields?: {
      field: string;
      oldType: string;
      newType: string;
    }[];
    removedEndpoints?: string[];
    authenticationChanges?: string[];
  };
  migrationGuide: string;
  effectiveDate: Date;
}
```

2. Non-Breaking Changes

Backward-compatible additions resulting in minor version increments.

```
interface NonBreakingChange {
  type: 'NON_BREAKING';
  changes: {
    addedFields?: string[];
    addedEndpoints?: string[];
    optionalParameters?: string[];
    enhancedResponses?: string[];
  };
  announcementDate: Date;
```

```
}
```

3. Bug Fixes

Corrections that don't affect the API contract, resulting in patch version increments.

```
interface BugFix {
  type: 'BUG_FIX';
  fixes: {
    description: string;
    affectedEndpoints: string[];
    resolution: string;
  }[];
  deploymentDate: Date;
}
```

Version Implementation

Manages different API versions within the codebase while maintaining clean separation between versions.

Version Manager

```
class APIVersionManager {
  private versions: Map<number, Implementation>;

  constructor() {
    this.versions = new Map();
    this.initializeVersions();
  }

  private initializeVersions() {
    this.versions.set(1, new V1Implementation());
    this.versions.set(2, new V2Implementation());
  }

  public getImplementation(version: number): Implementation {
    const impl = this.versions.get(version);
    if (!impl) {
      throw new Error(`Version ${version} not supported`);
    }
    return impl;
```

```
  }
}
```

Change Management Process

1. Planning Phase

Structured approach to planning and assessing API changes.

```
interface ChangeProposal {
  type: 'BREAKING' | 'NON_BREAKING' | 'BUG_FIX';
  description: string;
  justification: string;
  impact: {
    clients: string[];
    endpoints: string[];
    estimatedEffort: string;
  };
  timeline: {
    developmentStart: Date;
    betaRelease: Date;
    stableRelease: Date;
    deprecationDate?: Date;
  };
}
```

2. Communication Strategy

Clear communication of changes to API clients.

```
interface VersionAnnouncement {
  version: string;
  type: 'NEW_VERSION' | 'DEPRECATION' | 'END_OF_LIFE';
  details: {
    summary: string;
    changes: string[];
    migrationGuide?: string;
    timeline: {
      announcementDate: Date;
      effectiveDate: Date;
      endOfSupportDate?: Date;
```

```
    };
  };
  distributionChannels: string[];
}
```

Deprecation Strategy

1. Deprecation Timeline

```
interface DeprecationSchedule {
  version: string;
  announceDate: Date;
  deprecationDate: Date;
  endOfLifeDate: Date;
  alternativeVersion: string;
  migrationDeadline: Date;
}

const DEPRECATION_POLICY = {
  minimumNoticePeroid: 180, // days
  supportWindow: 365, // days
  maxVersionsSupported: 2
};
```

2. Deprecation Notices

Implementation of standardized deprecation warnings.

```
app.use((req, res, next) => {
  const version = req.apiVersion;

  if (isDeprecated(version)) {
    res.setHeader('Deprecation', 'true');
    res.setHeader('Sunset', getEndOfLifeDate(version));
    res.setHeader('Link', `<${getUpgradeGuideUrl(version)}>; rel="deprecation"`);
  }

  next();
});
```

Monitoring and Analytics

Track API version usage to inform deprecation decisions.

```
interface VersionMetrics {
  version: string;
  requests: number;
  uniqueClients: number;
  errorRate: number;
  avgResponseTime: number;
  deprecationStatus: 'active' | 'deprecated' | 'sunset';
}

const trackVersionUsage = async (req: Request, res: Response, next: NextFunction) => {
  const startTime = Date.now();
  const version = req.apiVersion;

  res.on('finish', () => {
    const duration = Date.now() - startTime;
    metrics.recordVersionUsage({
      version,
      duration,
      status: res.statusCode,
      clientId: req.auth?.clientId
    });
  });

  next();
};
```

Best Practices

1. **Version Selection**
   - Use semantic versioning consistently
   - Make breaking changes only in major versions
   - Maintain at least one previous major version

2. **Change Management**
   - Plan changes well in advance
   - Provide detailed migration guides
   - Use feature flags for gradual rollouts

3. **Communication**
   - Announce changes early
   - Provide clear upgrade paths

- Maintain comprehensive documentation

4. **Support**
   - Offer migration tools
   - Provide support during transitions
   - Monitor version usage patterns

5. **Maintenance**
   - Regular deprecation reviews
   - Clean up deprecated features
   - Archive old documentation

Implementation Checklist

- ☐ Define versioning strategy
- ☐ Set up version routing
- ☐ Implement version manager
- ☐ Create deprecation policy
- ☐ Configure monitoring
- ☐ Prepare communication templates
- ☐ Document upgrade paths
- ☐ Set up automated testing

Revision #3
Created 26 November 2024 01:15:06 by Aric Fedida
Updated 26 November 2024 01:25:20 by Aric Fedida