Versioning in Software Development

Part 1: Semantic Versioning

Introduction to Semantic Versioning

Semantic Versioning (SemVer) is a versioning scheme designed to communicate the nature of changes in software releases. It provides a standardized way to convey meaning about the underlying changes through version numbers.

The basic format of SemVer is: **MAJOR.MINOR.PATCH** (e.g., 2.1.4)

Each component has a specific meaning:

- **MAJOR**: Incremented when making incompatible API changes
- **MINOR**: Incremented when adding functionality in a backward-compatible manner
- **PATCH**: Incremented when making backward-compatible bug fixes

The Importance of Semantic Versioning

Semantic versioning addresses several critical needs in software development:

1. **Clear communication**: Developers and users immediately understand the impact of a version change
2. **Dependency management**: Package managers can intelligently handle version constraints
3. **Risk assessment**: Teams can evaluate the potential impact of upgrades
4. **Upgrade planning**: Organizations can plan resources needed for updates

SemVer Rules in Detail

The SemVer specification includes several rules:

1. Version numbers must increase numerically: 1.9.0 → 1.10.0 (not 1.91)
2. Once a version is released, its content MUST NOT be modified. Any changes require a new version
3. Pre-release versions use a hyphen and identifier: 1.0.0-alpha, 1.0.0-beta.2
4. Build metadata uses a plus sign: 1.0.0+20130313144700

Examples of SemVer Changes

Let's examine real-world scenarios where each version component would change:

PATCH version increment (1.0.0 → 1.0.1):

- Fixed a bug where timestamps were incorrectly formatted
- Corrected a typo in error messages
- Optimized database query performance without changing functionality

MINOR version increment (1.0.1 → 1.1.0):

- Added a new endpoint to the API
- Introduced a new optional parameter
- Added support for a new file format

MAJOR version increment (1.1.0 → 2.0.0):

- Changed required parameters for an API endpoint
- Removed a deprecated method
- Changed the response format of essential endpoints

Practical Implementation

When implementing SemVer in your project, consider:

1. Documenting version policy in your README
2. Using CHANGELOG.md to track changes
3. Automating version bumps when possible
4. Clearly communicating breaking changes to users

Part 2: API Versioning

Introduction to API Versioning

API versioning is the practice of managing changes to an API while maintaining backward compatibility for existing clients. Versioning allows API providers to evolve their services without breaking client applications.

Why Version APIs?

APIs need versioning because:

1. **Evolution is inevitable**: Requirements change, bugs are fixed, and features are added
2. **Multiple consumers**: Different clients may adopt API changes at different rates
3. **Client stability**: Consumers depend on consistent behavior for their applications
4. **Transition periods**: Organizations often need time to move from old to new versions

Common API Versioning Strategies

1. URI Path Versioning

Including the version in the URL path:

```
https://api.example.com/v1/users
https://api.example.com/v2/users
```

Advantages:

- Highly visible and explicit
- Easy to route to different code bases
- Straightforward for clients to understand

Disadvantages:

- Resources have different URIs across versions
- Can violate REST principles of resource identification
- URLs change with version, making hypermedia navigation complex

2. Query Parameter Versioning

Using a query parameter to specify the version:

```
https://api.example.com/users?version=1
https://api.example.com/users?version=2
```

Advantages:

- Maintains consistent resource URIs
- Easy to implement
- Optional (can default to latest)

Disadvantages:

- Easy to miss when copying URLs
- May be stripped in some contexts
- Less explicit than URI versioning

3. Header-Based Versioning

Using HTTP headers to specify the version:

```
GET /users HTTP/1.1
Host: api.example.com
Accept-version: v1

GET /users HTTP/1.1
Host: api.example.com
Accept-version: v2
```

Advantages:

- Cleaner URLs
- Follows HTTP content negotiation principles
- Separates versioning from resource identification

Disadvantages:

- Less visible
- Harder to test (requires header manipulation)
- More complex for API consumers

4. Media Type Versioning (Content Negotiation)

Using the Accept header with custom media types:

```
GET /users HTTP/1.1
Host: api.example.com
Accept: application/vnd.example.v1+json

GET /users HTTP/1.1
Host: api.example.com
Accept: application/vnd.example.v2+json
```

Advantages:

- Follows HTTP content negotiation standards
- Maintains resource URI consistency
- Can version media types rather than entire APIs

Disadvantages:

- Most complex to implement and consume
- Less transparent than other methods
- Requires deeper HTTP knowledge

Best Practices for API Versioning

1. **Plan for versioning from the start**, even if you only have one version initially
2. **Document changes thoroughly** between versions
3. **Support multiple versions simultaneously** during transition periods
4. **Communicate deprecation timelines** clearly
5. **Consider compatibility headers** to test client readiness
6. **Use semantic versioning principles** for API versions

Part 3: Versioning in FastAPI

Introduction to FastAPI Versioning

FastAPI is a modern, high-performance web framework for building APIs with Python. It includes built-in support for API versioning through several mechanisms.

Versioning Options in FastAPI

FastAPI supports multiple versioning approaches:

1. Path Versioning in FastAPI

Creating different route paths for different versions:

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/v1/users/")
async def get_users_v1():
    return {"version": "v1", "data": ["user1", "user2"]}

@app.get("/v2/users/")
async def get_users_v2():
    return {"version": "v2", "data": {"users": ["user1", "user2"]}}
```

This approach is straightforward but requires duplication of route declarations.

2. Using APIRouter with Prefixes

A more organized approach using APIRouter with version prefixes:

```python
from fastapi import FastAPI, APIRouter

app = FastAPI()
```

```
# Create routers for different versions
router_v1 = APIRouter(prefix="/v1")
router_v2 = APIRouter(prefix="/v2")

# Define routes on each router
@router_v1.get("/users/")
async def get_users_v1():
    return {"version": "v1", "data": ["user1", "user2"]}

@router_v2.get("/users/")
async def get_users_v2():
    return {"version": "v2", "data": {"users": ["user1", "user2"]}}

# Include routers in the main app
app.include_router(router_v1)
app.include_router(router_v2)
```

This approach keeps code more organized and maintainable.

3. Header-Based Versioning

Implementing header-based versioning using dependencies:

```
from fastapi import FastAPI, Header, HTTPException, Depends

app = FastAPI()

async def get_api_version(api_version: str = Header(None)):
    if api_version is None:
        return "1.0"  # Default version
    return api_version

@app.get("/users/")
async def get_users(version: str = Depends(get_api_version)):
    if version == "1.0":
        return {"version": "1.0", "data": ["user1", "user2"]}
    elif version == "2.0":
        return {"version": "2.0", "data": {"users": ["user1", "user2"]}}
    else:
        raise HTTPException(status_code=400, detail="API version not supported")
```

This approach checks the API version from headers and returns appropriate responses.

4. Media Type Versioning in FastAPI

Using content negotiation with dependencies:

```
from fastapi import FastAPI, HTTPException, Depends
from fastapi.responses import JSONResponse
from typing import Optional

app = FastAPI()

async def get_accept_version(accept: Optional[str] = Header(None)):
    if not accept:
        return "1.0"
    if "application/vnd.company.v1+json" in accept:
        return "1.0"
    elif "application/vnd.company.v2+json" in accept:
        return "2.0"
    else:
        return "1.0"  # Default to v1

@app.get("/users/")
async def get_users(version: str = Depends(get_accept_version)):
    if version == "1.0":
        content = {"version": "1.0", "data": ["user1", "user2"]}
        return JSONResponse(
            content=content,
            media_type="application/vnd.company.v1+json"
        )
    elif version == "2.0":
        content = {"version": "2.0", "data": {"users": ["user1", "user2"]}}
        return JSONResponse(
            content=content,
            media_type="application/vnd.company.v2+json"
        )
```

1. What is "Accept"?

The `Accept` header is part of the HTTP protocol and is used by clients (e.g., browsers, APIs, or other services) to specify the media types (or formats) that they can understand or are willing to accept in the response from the server.

For example:

- `Accept: application/json` means the client expects a JSON response.
- `Accept: text/html` means the client expects an HTML response.

In the context of API versioning, the `Accept` header can also be used to indicate the version of the API that the client wants to interact with. This is often done using custom media types, which brings us to the next part.

---

2. What is "vnd"?

`vnd` stands for **"vendor"** and is a prefix used in custom media types to indicate that the media type is specific to a particular vendor or organization. It is part of the Internet Assigned Numbers Authority (IANA) standards for defining custom MIME types.

For example:

- `application/vnd.company.v1+json` is a custom media type where:
    - `application` indicates the general type (e.g., structured data).
    - `vnd.company` specifies that this is a vendor-specific format owned by "company."
    - `v1` indicates the version of the API.
    - `+json` specifies that the format is JSON.

This approach allows APIs to use the `Accept` header for versioning, enabling clients to request specific versions of the API without changing the URL or query parameters.

Key Points:

1. **Purpose of the Function**:
   - The `get_accept_version` function extracts the API version from the `Accept` header.
   - If the `Accept` header is not provided, it defaults to version `1.0`.
2. **Custom Media Types**:
   - The function checks for specific custom media types:
     - `application/vnd.company.v1+json` corresponds to version `1.0`.
     - `application/vnd.company.v2+json` corresponds to version `2.0`.
3. **Default Behavior**:
   - If the `Accept` header does not match any of the specified media types, the function defaults to version `1.0`.
4. **Usage of `Header`**:
   - The `Header` dependency in FastAPI is used to extract the value of the `Accept` header from the incoming HTTP request.

Example Usage

Request 1:

```
GET /example HTTP/1.1
Host: api.example.com
Accept: application/vnd.company.v1+json
```

- The `Accept` header specifies version `1.0`.
- The function will return `"1.0"`.

Request 2:

```
GET /example HTTP/1.1
Host: api.example.com
Accept: application/vnd.company.v2+json
```

- The `Accept` header specifies version `2.0`.
- The function will return `"2.0"`.

Request 3:

```
GET /example HTTP/1.1
Host: api.example.com
Accept: application/json
```

- The `Accept` header does not match any of the custom media types.
- The function will default to `"1.0"`.

Implementing a Comprehensive Version Strategy

For a production API, you might want to combine approaches:

```
from fastapi import FastAPI, APIRouter, Header, HTTPException, Depends
from typing import Optional, Dict, List, Union

app = FastAPI(title="My Versioned API")

# Version-specific routers
v1_router = APIRouter(prefix="/v1")
v2_router = APIRouter(prefix="/v2")

# Version-specific data models
class UserResponseV1:
    def __init__(self, username: str, email: str):
        self.username = username
        self.email = email

class UserResponseV2:
    def __init__(self, username: str, email: str, role: str = "user"):
        self.username = username
        self.email = email
        self.role = role

# User database (in memory for example)
users_db = [
    {"username": "johndoe", "email": "john@example.com", "role": "admin"},
    {"username": "janedoe", "email": "jane@example.com", "role": "user"}
]

# V1 endpoints
@v1_router.get("/users/", response_model=List[Dict[str, str]])
async def get_users_v1():
    # V1 returns a simple list of user data
    return [{"username": user["username"], "email": user["email"]} for user in users_db]

@v1_router.get("/users/{username}")
async def get_user_v1(username: str):
    for user in users_db:
        if user["username"] == username:
            return {"username": user["username"], "email": user["email"]}
    raise HTTPException(status_code=404, detail="User not found")

# V2 endpoints with enhanced features
@v2_router.get("/users/")
async def get_users_v2():
    # V2 returns a structured response with metadata
    return {
        "version": "2.0",
        "total": len(users_db),
        "users": users_db
    }

@v2_router.get("/users/{username}")
async def get_user_v2(username: str):
    for user in users_db:
        if user["username"] == username:
            return user
    raise HTTPException(status_code=404, detail="User not found")

# Header-based version selection for a shared endpoint
async def get_api_version(api_version: Optional[str] = Header(None, alias="X-API-
Version")):
    if api_version is None:
        return "1.0"  # Default to v1
    if api_version not in ["1.0", "2.0"]:
        raise HTTPException(status_code=400, detail="API version not supported")
    return api_version

@app.get("/shared/users/")
async def get_users_shared(version: str = Depends(get_api_version)):
    if version == "1.0":
        return [{"username": user["username"], "email": user["email"]} for user in
users_db]
    else:  # version == "2.0"
        return {
            "version": "2.0",
            "total": len(users_db),
            "users": users_db
        }

# Include both routers
app.include_router(v1_router)
app.include_router(v2_router)
```

Handling Versioning With Dependency Injection

FastAPI's dependency injection system is powerful for versioning:

```
from fastapi import FastAPI, Depends, HTTPException
from typing import Callable, Dict, List, Type, TypeVar

app = FastAPI()

# Define type for user models
T = TypeVar('T')

# Different user repositories for different versions
class UserRepositoryV1:
    async def get_users(self) -> List[Dict[str, str]]:
        # V1 implementation
        return [
            {"username": "user1", "email": "user1@example.com"},
            {"username": "user2", "email": "user2@example.com"}
        ]

class UserRepositoryV2:
    async def get_users(self) -> Dict[str, object]:
        # V2 implementation with more data and structure
        return {
            "count": 2,
            "users": [
                {"username": "user1", "email": "user1@example.com", "role": "admin"},
                {"username": "user2", "email": "user2@example.com", "role": "user"}
            ]
        }

# Factory function to get the appropriate repository based on version
def get_repository(version: str) -> Callable[[], Type[T]]:
    if version == "1":
        return UserRepositoryV1
    elif version == "2":
        return UserRepositoryV2
    else:
        raise HTTPException(status_code=400, detail="Unsupported version")

# Routes using path versioning and dependencies
@app.get("/v1/users/")
async def get_users_v1(repo: UserRepositoryV1 = Depends(lambda: get_repository("1")())):
    return await repo.get_users()

@app.get("/v2/users/")
async def get_users_v2(repo: UserRepositoryV2 = Depends(lambda: get_repository("2")())):
    return await repo.get_users()
```

Documenting API Versions

FastAPI automatically generates documentation using OpenAPI. You can enhance this by:

1. Adding version-specific tags
2. Using clear descriptions for each version
3. Marking deprecated endpoints

```
from fastapi import FastAPI, APIRouter

app = FastAPI(title="My API", description="API with versioning")

# Create version-specific routers with tags
v1_router = APIRouter(
    prefix="/v1",
    tags=["v1"],
    deprecated=False,
    responses={404: {"description": "Not found"}}
)

v2_router = APIRouter(
    prefix="/v2",
    tags=["v2"],
    responses={404: {"description": "Not found"}}
)

# Deprecated v1 endpoint
@v1_router.get(
    "/legacy/",
    summary="Legacy endpoint",
    description="This endpoint will be removed in v3",
    deprecated=True
)
async def legacy_endpoint():
    return {"message": "This is deprecated"}
```

```python
# Current v1 endpoint
@v1_router.get(
    "/users/",
    summary="Get users (v1)",
    description="Returns a basic list of users"
)
async def get_users_v1():
    return ["user1", "user2"]

# Enhanced v2 endpoint
@v2_router.get(
    "/users/",
    summary="Get users (v2)",
    description="Returns enhanced user data with additional fields"
)
async def get_users_v2():
    return {
        "total": 2,
        "users": [
            {"username": "user1", "email": "user1@example.com", "role": "admin"},
            {"username": "user2", "email": "user2@example.com", "role": "user"}
        ]
    }

app.include_router(v1_router)
app.include_router(v2_router)
```

Part 4: Putting It All Together - Version Management Best Practices

Comprehensive Version Management Strategy

A comprehensive versioning strategy should include:

1. **Use semantic versioning for your package/application version**
    - Follow MAJOR.MINOR.PATCH principles
    - Document changes in a CHANGELOG.md
2. **Choose an appropriate API versioning approach**
    - Path versioning is often simplest
    - Header versioning may be cleaner for RESTful resources
3. **Maintain version compatibility periods**
    - Support at least one previous version
    - Communicate deprecation schedules
4. **Document breaking changes clearly**
    - Provide migration guides
    - Explain the benefits of upgrading

Version Management Implementation Example

Here's a comprehensive implementation combining all these approaches:

```python
from fastapi import FastAPI, APIRouter, Header, Depends, HTTPException
from typing import Optional, List, Dict, Union
```

```python
import logging

# Set up application with metadata
app = FastAPI(
    title="Sample API",
    description="API demonstrating versioning best practices",
    version="2.1.0",  # Semantic version of the application
    docs_url="/docs",
    redoc_url="/redoc",
)

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Version-specific routers
v1_router = APIRouter(prefix="/v1", tags=["v1"])
v2_router = APIRouter(prefix="/v2", tags=["v2"])

# In-memory database for demonstration
products_db = [
    {"id": 1, "name": "Laptop", "price": 999.99, "category": "Electronics"},
    {"id": 2, "name": "Smartphone", "price": 499.99, "category": "Electronics"},
    {"id": 3, "name": "Headphones", "price": 99.99, "category": "Audio"}
]

# Version determination from headers
async def get_api_version(api_version: Optional[str] = Header(None, alias="X-API-
Version")):
    if api_version is None:
        return "1.0"  # Default version if not specified

    if api_version not in ["1.0", "2.0"]:
        logger.warning(f"Unsupported API version requested: {api_version}")
        raise HTTPException(
            status_code=400,
            detail=f"API version {api_version} not supported. Supported versions: 1.0, 2.0"
        )

    return api_version

# V1 endpoints
@v1_router.get("/products/", summary="List all products (v1)")
async def get_products_v1():
    """
    Returns a simple list of products with basic information.

    **V1 Implementation**: Returns a flat list of product data.
    """
    return [{"id": p["id"], "name": p["name"], "price": p["price"]} for p in products_db]

@v1_router.get("/products/{product_id}", summary="Get a specific product (v1)")
async def get_product_v1(product_id: int):
    """
    Returns details about a specific product.

    **V1 Implementation**: Returns basic product information.
    """
    for product in products_db:
        if product["id"] == product_id:
            return {"id": product["id"], "name": product["name"], "price":
product["price"]}

    raise HTTPException(status_code=404, detail="Product not found")

# V2 endpoints with enhanced features
@v2_router.get("/products/", summary="List all products (v2)")
async def get_products_v2(category: Optional[str] = None):
    """
    Returns an enhanced list of products with additional information.

    **V2 Implementation**:
    - Returns structured response with metadata
    - Supports filtering by category
    - Includes additional product details
    """
    filtered_products = products_db
    if category:
        filtered_products = [p for p in products_db if p["category"].lower() ==
category.lower()]

    return {
        "metadata": {
            "version": "2.0",
            "total": len(filtered_products),
            "api_version": "2.1.0"  # Application's semantic version
        },
        "products": filtered_products
    }

@v2_router.get("/products/{product_id}", summary="Get a specific product (v2)")
async def get_product_v2(product_id: int):
    """
    Returns enhanced details about a specific product.

    **V2 Implementation**: Returns complete product information.
    """
    for product in products_db:
        if product["id"] == product_id:
            return {
                "metadata": {
                    "version": "2.0",
                    "api_version": "2.1.0"  # Application's semantic version
                },
                "product": product
            }

    raise HTTPException(status_code=404, detail="Product not found")

# Shared endpoint with header-based versioning
@app.get("/products/", summary="List products (header-versioned)")
async def get_products_shared(version: str = Depends(get_api_version)):
    """
    Returns a list of products with version-specific implementations.

    Specify version using the X-API-Version header.
    """
    if version == "1.0":
        products = [{"id": p["id"], "name": p["name"], "price": p["price"]} for p in 
products_db]
        return products
    else:  # version == "2.0"
        return {
            "metadata": {
                "version": "2.0",
                "total": len(products_db),
                "api_version": "2.1.0"  # Application's semantic version
            },
            "products": products_db
        }

# Health check endpoint with version information
@app.get("/health", tags=["system"])
async def health_check():
    """System health check endpoint that returns version information."""
    return {
        "status": "healthy",
        "api_version": "2.1.0",  # Application's semantic version
        "supported_api_versions": ["1.0", "2.0"]
    }

# Include version-specific routers
app.include_router(v1_router)
app.include_router(v2_router)

# Startup event to log version information
@app.on_event("startup")
async def startup_event():
    logger.info(f"Starting API version 2.1.0")
    logger.info("Supporting API versions: 1.0, 2.0")
```

Final Recommendations

1. **Start with versioning in mind** - It's easier to implement from the beginning than retrofit later
2. **Choose the right versioning strategy for your needs**:
    - Path versioning for simplicity and visibility
    - Header versioning for cleaner URLs
    - Consider combining strategies for flexibility
3. **Follow semantic versioning principles** to communicate the impact of changes
4. **Document your versioning strategy** for both internal developers and API consumers
5. **Plan for version transitions**:
    - Support multiple versions concurrently during transition periods
    - Provide migration guides for major changes
    - Set clear deprecation schedules
6. **Use FastAPI's features** like dependency injection, response models, and automatic documentation to support your versioning strategy
7. **Monitor version usage** to make informed decisions about deprecation timelines

By implementing a thoughtful versioning strategy, you can evolve your software while maintaining compatibility with existing clients, ensuring a smoother experience for your users and developers alike.