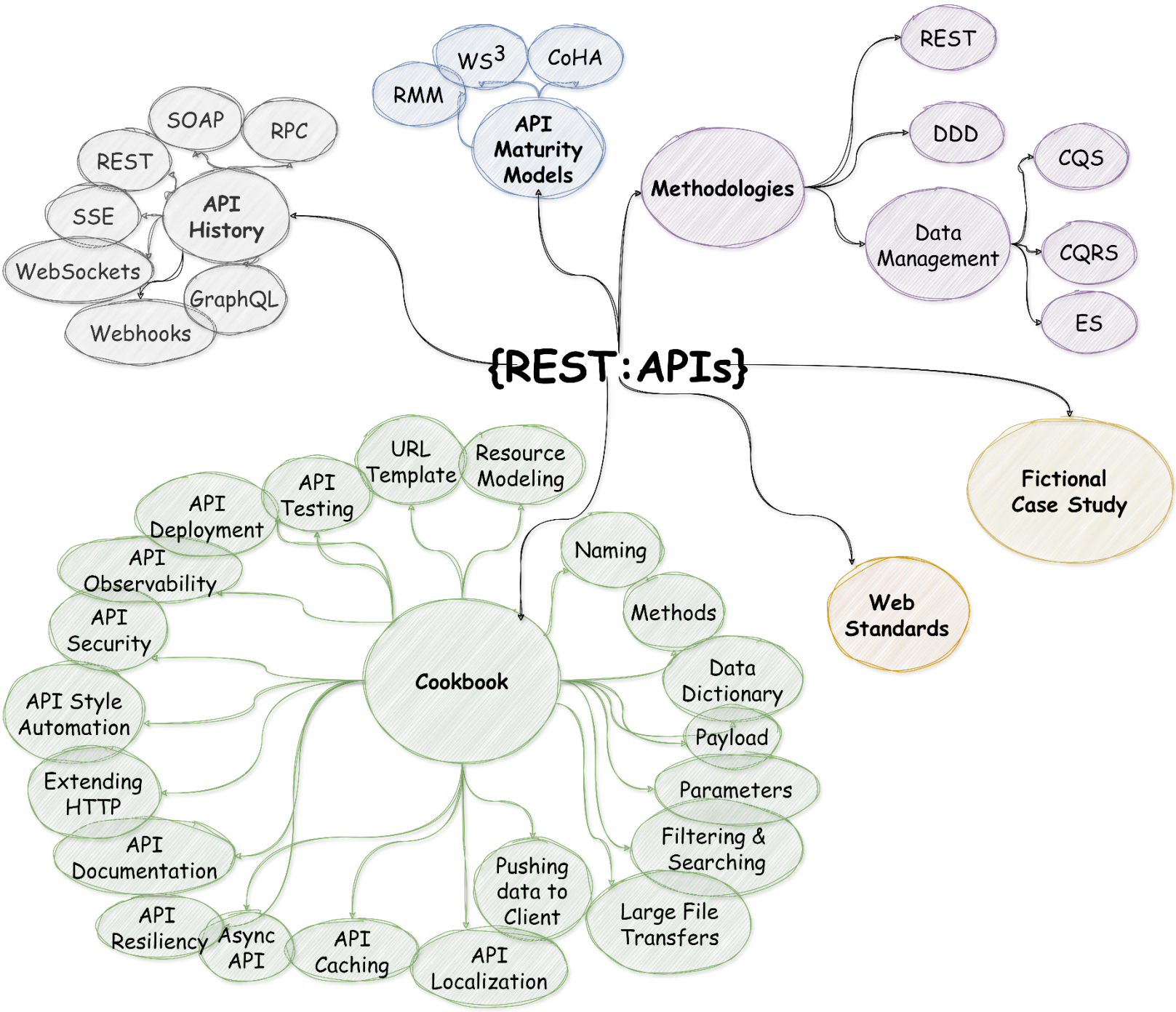


Simple RESTful API design guidelines that make sense

[that Developers will (want to) use]

Michalis ARGYRIOU

s

Table of Contents

Table of Figures

Table of Tables

Chapter 1. Introduction

1.1. Purpose of this book

This book provides a small cookbook of simple guidelines for designing RESTful APIs. They have been validated:

- by different points-of-view, for example, backend developer, frontend developer, enterprise architect, system architect, solution architect, etc.
- on various frameworks, such as Spring Boot, Angular, React, etc.

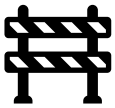
Each guideline is duly justified in its own section later in the book.

1.2. About the author

Michalis ARGYRIOU[1] has been involved as developer or architect in numerous projects for the European Commission, EU Agencies, and International organizations for the last 15 years. Currently, M. ARGYRIOU is serving as Head of Development and Architecture in Sword Services Greece S.A. M. ARGYRIOU is the author of the WIPO Standard ST.90[2] and the original author of ST.97[3] for the standardization of the exchange of IP data using APIs. Therefore, M. ARGYRIOU is familiar with various standards and industry practices on designing APIs.

1.3. Acknowledgements

The people that have reviewed this book – in no particular order:



1.4. Why is this book different from the other “REST API” books?

This book is **purely technical**. Its intention is to be practical and simple. It targets RESTful APIs for typical System (not large-scale systems that have extreme peculiar needs).

Practical because it follows a top-down approach by **providing first a cookbook of guidelines in a few pages** and if someone wants to better understand why each guideline is provide may delve into detail into the respective section that justifies each guideline.

Simple because it prioritizes simplicity to **simplify the guidelines**. For example, CRUD APIs and Intent APIs are described but it is recommended Intent APIs to be converted to CRUD APIs.

This book is published on leanpub.com following a lean-based approach. As a result, it will be evolving, and **its content will always be improving**.

[1] https://www.linkedin.com/in/micharg/
[2] https://www.wipo.int/export/sites/www/standards/en/pdf/03-90-01.pdf
[3] https://www.ompi.int/export/sites/www/standards/en/pdf/03-97-01.pdf

1.5. Notation

The following notation is used in the book:

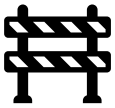
	This paragraph provides an important information. Make sure you do not miss it!
	This paragraph lists pros.
	This paragraph lists cons.
	This paragraph provides a guideline.
	This paragraph describes information how to realize a guideline.
	Case Study from the IT Industry
	Alternative
	Paragraph is in progress.

Table 1 Notation

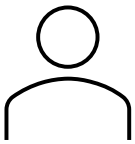
For all the examples provided in JSON, the naming convention is lowerCamelCase (for example, for Java implementation of JSON serialization and deserialization).

1.6. Errata

For any typo or mistake identified or any improvement proposed, please send an email to micharg@gmail.com.

1.7. Book Target Audience

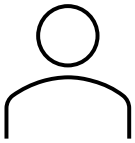
This book answers the following concerns:



Backend Developer

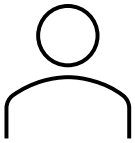
Should I expose the physical model (DB tables) or the logical model (classes) as API resources?

What naming conventions should I use to serialize classes to DTOs?



Frontend Developer

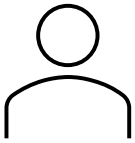
Does the API support the UI components needs, such as pagination, sorting, etc?



Tester

What aspects of the REST API should I test?

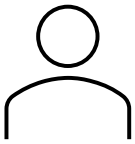
How should I test it?



System Architect

Should API support our DDD's Ubiquitous Language?

Is it possible to use REST API with CQRS?



Solution Architect

How can I make sure that developers will create System RESTful APIs that follow consistent rules?

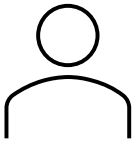
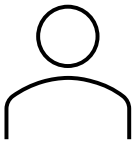
 Security Architect	*What to check to verify that API does not expose sensitive information?*
 Enterprise Architect	*Which API Design Guidelines should the, such as naming conventions) should the APIs in the Application Architecture follow? Can I enforce them?*

Table 2 Book Target Audience

1.8. Book Structure

- **Chapter 1. Introduction**: This Chapter provides the goal of this book and instructions how to read it to get value as fat as possible.
- **Chapter 2. REST API Cookbook**: This Chapter provides a summary of the RESTful API guidelines.
- **Chapter 3. Fictional Case Study**Error! Reference source not found.**:** This Chapter applies the Cookbook of the previous Chapter to design the API for a fictitious Case Study.
- **Chapter 4. The History and evolution of APIs:** This Chapter provides a brief history from RPC to RESTful APIs to connect the dots why RESTful APIs are so popular. It also presents a few other solutions to better describe how RESTful APIs compare with them.
- **Chapter 5. On the shoulders of giants.:** This Chapter provides in a central location the fundamental concepts from the Internet Standards that RESTful APIs use to avoid repeatedly cluttering the book with definitions or/and references.
- **Chapter 6. API Maturity Models**: This Chapter describes industry frameworks that compare APIs based on quality criteria. Then it concludes how "mature" an API should be to balance maturity with simplicity.
- **Chapter 7. API Design Patterns Analysis**: This Chapter lists for each problem alternative solutions. Then it concludes to a guideline considering the previous Chapter's selected API maturity level.
- **Annex. Terminology**: Provides an index for terms defined in the book.

To make this book useful as soon as possible jump to Section Chapter 2 - REST API Cookbook. To understand the reasoning that supports each design decision, just to the relative referenced paragraph in Chapter 7.

Chapter 2. REST API Cookbook

2.1. Principles

A Beautiful & Simple RESTful API should respect the following principles[4]:

- Use URI-identifiable APIs.
- Use multiple uniquely URI-identified resources that have a single resource representation.
- Support multiple resources representations.
- Design server as stateless.
- Use HTTP semantics.
- Use self-described messages.
- Label server responses as cacheable or not.
- Design server as layered, for example, security, business logic, application.

2.2. URL Template

2.2.1. API Versioning

Versioning Strategy

For API Versioning, use Path Segment Versioning

Versioning Scheme

For API Versioning Scheme, use Semantic Versioning (only MAJOR)

Deployment Strategy

- Backward compatible change ➔ new API uses only the major version and replaces previous version (transparent to clients)
- Non-backward compatible change ➔ deploy new API version using only the major version + preserve previous API version (for a transient period to give time to clients to migrate

2.2.2. Multi-environment APIs

For multi-environment APIs, use Subdomains to indicate each environment (for example, qa, dev). For the production environment the subdomain <env.> should not be used.
Wildcard certificates are required to support all the environments (subdomains).

2.2.3. URL Collision

To avoid URL Collision, use the Path Segment “api” for the URI, for example, acme.com/**api**/.

2.2.4. Resource Modelling

Resource Location

[4] See Section Chapter 6 - API Maturity Models

Do not use a trailing forward slash (/) in the URI

Mapping concepts to resources

An API may map its internal domain (concepts) as API resources with the following approaches:

- Expose the Ubiquitous Language directly (to **Conformist** clients). This alternative should be preferred if the client is developed by the same team that develops the API.
- Map the Ubiquitous Language to a new standard language (with **OHS** and a **Published Language).** This alternative should be preferred if the client is developed by another team and there is a Published Language that can be used.
- Map the Ubiquitous Language to a new non-standard language. The API client is recommended to use an **ACL**. This alternative should be preferred if another team develops the API client.

Organizing resources as CRUD resources

Model all resources as CRUD resources (Document, Collection or Store). Convert Intent resources (Controller) to CRUD resources. Conversion is always possible.

Naming Resource

The recommended convention for naming resources, is **kebab-case** and abbreviated if possible. Resource names should be nouns in plural nouns. Example: /<resource-name>s.

2.3. Methods

Use only Use the following HTTPS methods only: GET, PUT, DELETE, POST, PATCH.

2.4. Data Dictionary

- Date & Time: Should be formatted as specified in ISO 8601
  - Datetime (i.e., timestamp) as yyyy-MM-dd'T'HH:mm:ssZ
    - If time should be localized on Client's timezone: <see paragraph for Localization>
  - Date as yyyy-MM-dd
  - Time zone information: As specified in IETF RFC 3339. For example: 20:54:21+00:00
- Numbers can be integers (whole counting numbers) or floating points (high precision numbers)
- Currency: Should be formatted as specified in ISO 4217-Alpha (3-Letter Currency Codes)
- Country names: Should be formatted as specified in ISO 3166-1-Alpha-2 Code Elements (2 letter country codes)
- Language codes: Should be formatted as specified in ISO 639-1 (2-Letter Language Codes)
- Units of Measure: Should use the units of measure as described in The Unified Code for Units of Measure (based on ISO 80000 definitions). For example, for weight measuring using kilograms (kg)
- Empty fields: If a field is empty (null) then do not include it in the request or response

2.5. Payload

2.5.1. Content Negotiation

For Content Negotiation, use HTTP Headers

2.5.2. Request

Body properties names

Request/Response body property names is recommended to be lowerCamelCaseRequest/Response body property names is recommended to be lowerCamelCase

Request payload template

Request body cannot be standardized ({}).

2.5.3. Request Content Type

A CRUD API (PUT, POST, PATCH) should accept application/json.

2.5.4. Response

Body properties names

Request/Response body property names is recommended to be lowerCamelCaseRequest/Response body property names is recommended to be lowerCamelCase

Response Content Type

A CRUD API (GET, POST, PATCH) should return application/json

Response Payload Template (on success)

The response payload of a successfully processed request should have the following structure:

```
{
  "data": { ... },
  "metadata": { ... }
}
```

HTTP Code (on success)

GET|PUT|PATCH|DELETE|POST should use 200 OK to simplify.

Response Payload Template (on failure)

An application-level error should have the following structure that extends RFC 7807:

- **type** (string, optional): Cheat: use a code.
- **title** (optional, string): Localizable based on the request HTTP header Accept-Language.
- **status** (optional, number): Duplicate of the HTTP status code to be close to the associated payload.
- **detail** (string)
- **instance** (string)
- **{extension-members}**: For example: invalid-params[]{name, reason} for validation errors (name can be ref to DOM element id)
- **timestamp** (mandatory, ts): when the error was raised.

- **debugMessage** (optional): technical message (not for PROD), for example a UUID, for logging purposes

HTTP Code (on failure)

Application-level failures from GET|PUT|PATCH|DELETE|POST should return **200 OK** to simplify. Specific use cases may override this simple guideline.

2.6. Parameters

Parameter Types

- Use **Request Parameters** to limit the query length.
- Use **Path Parameters** for mandatory parameters (such as resource id)
- Use **Query Parameters** for optional parameters (see also Filtering & Searching recommended reserved names)

Query parameter names

Query parameter names is recommended to be snake case.

Query parameter names is recommended to be snake case.

2.7. Filtering & Searching

- Use **Filtering** for querying based on a few query criteria using GET or POST (consider the pragmatic query length restriction of URLs).
- Use **Searching** for querying based on complex query expressions using GET or POST (consider the pragmatic query length restriction of URLs).
- Use **Searching** for querying based on complex query expressions using GET or POST (consider the pragmatic query length restriction of URLs).
- Use **Predefined Search** for remembering queries.

2.8. API Composition

Use API Composition in the API GW instead of making complex the upstream APIs. API Composition should not be used when pagination and sorting should be applied to multiple indexes

2.9. Concurrency Control

Implement Optimistic Concurrency using ETag and If-Match HTTP Headers and the HTTP Status Code 412 Precondition Failed if the entity has been changed between fetching it and changing it.

2.10. Consistency

Implement Optimistic Concurrency using ETag and If-Match HTTP Headers and the HTTP Status Code 412 Precondition Failed if the entity has been changed between fetching it and changing it.

2.11. Large File Transfers

- For downloads, prefer Content-Length.
- For uploads, prefer Transfer-Encoding: chunked

- Consider using the **Valet Key Pattern** to offload data transfer from the application.
- If using ETag should use Strong Validator instead of Weak Validator so that byte ranges (**Accept-Range**) can be cached

2.12. Pushing data to Client

- For unidirectionally pushing data to Client, use SSE.
- For bidirectionally pushing data from/to Client, use WebSockets.

2.13. API Localization

2.13.1. Language

Use the HTTP Header Accept-Language to fetch localized content from the API

2.13.2. Timezone

If business-wise is acceptable the API's Client to see the API's Timezone the then API should accept and return datetimes in its Timezone. If business-wise the API's Client must use its own Timezone, then datetime should be (a) transferred to server with the format ISO8601[TZ], (b) processed by the Server as a data type that contains the Timezone and (c) persisted in server's db as a data type that contains the Timezone.

2.13.3. Caching

Cache Control

For overriding the default caching behaviour, use the HTTP Header Cache-Control with a random jitter (to avoid the thundering herd effect).

ETag

Combine Cache-Control and the HTTP Validator Header ETag for Content-based determination if an entity has been modified.

ETag Generation

- For transferring dynamic generated content use Strong ETags. To generate Strong hash (using MD5) the message's payload is used (body) and its Content-Encoding.
- For transferring strictly version-controlled resources use Strong ETag populated with the entity's version number (should be stored in DB). For transferring static resources (such as JS, CSS, PNG, etc.) use Weak ETags.

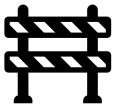
2.14. Async API

For long (async) running operations, such as PDF generation, the following steps must be followed:

a) Client submits long running operation (POST)
b) API returns 202 Accepted which means request accepted for processing but processing has not been completed. It returns the location where the Client should check for status update using the HTTP Header Location: <op-status-location>
c) Client checks status of operation with GET <op-status-location>

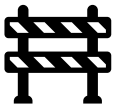
a. API returns 200 OK which means operation still in progress. Response may contain other information for the status of the operation, for example, progress indicator or link to cancel or delete the task.
b. API returns 200 OK and the response contain a description of the error in case the operation failed.
c. API returns 303 See Other which means operation has finished successfully. The HTTP Header Location indicates the URL where the result should be retrieved. Client uses the Location to get the generated result of the long operation.

2.15. API Resiliency



2.16. API Documentation

Workflows



API Specifications

For API Specifications, use OAS 3.x

2.17. Extending HTTP

Custom HTTP Headers

Use custom HTTP Headers but without using the X- prefix.

2.18. API Style Guide Automation

Use an API Linter to enforce the API Style Guide

2.19. API Security

Secure Access

CSRF

Data Tampering

Use SHA-3 or SHA-256 Cryptographic Hashes to protect against Data Tampering.

2.19.1. OWASP API Security Top 10

Mitigate against OWASP API Security Top 10 threats. Use an API Linter to enforce OWASP security guidelines.

2.20. Observability

The following response should be returned (following common-v1.yaml):

Service is up:

```
{
  "status": "UP"
}
```

Service is down (with additional details):

```
{
  "status": "DOWN",
  "details": {
    "datastore": {
      "status": "DOWN",
      "errorMessage": "connection timeout"
    }
  }
}
```

2.21. API Deployment

Follow the Port binding principle of the Twelve Factor application methodology to deploy multiple APIs.

2.22. API Testing

2.23. Discouraged Practices

Query expansion

Do not use query expansion

Semantic RESTful APIs

Do not design RESTful APIs as semantic RESTful APIs.

Hypermedia

Do not do HATEOAS since it will make the implementation of the API (and the client) more difficult (very complex) and most of the time it is not even used (client should be adapted to support it).

Matrix Parameters

Do not use Matrix Parameters

Chapter 3. Fictional Case Study

In this example, we will apply the guidelines of Chapter 2 REST API Cookbook in a fictional company.

3.1. ACME Corporation

Acme Corporation[5] manufactures outlandish products that fail or backfire catastrophically at the worst possible times. Acme wants to create an e-shop to provides access to its products to more clients and increase its revenue.

3.2. ACME's organization model

A single team will work to develop the Acme System for the MVP. If the System becomes bigger more teams may be formed.

3.3. ACME's domain model

Acme's domain model is the following:

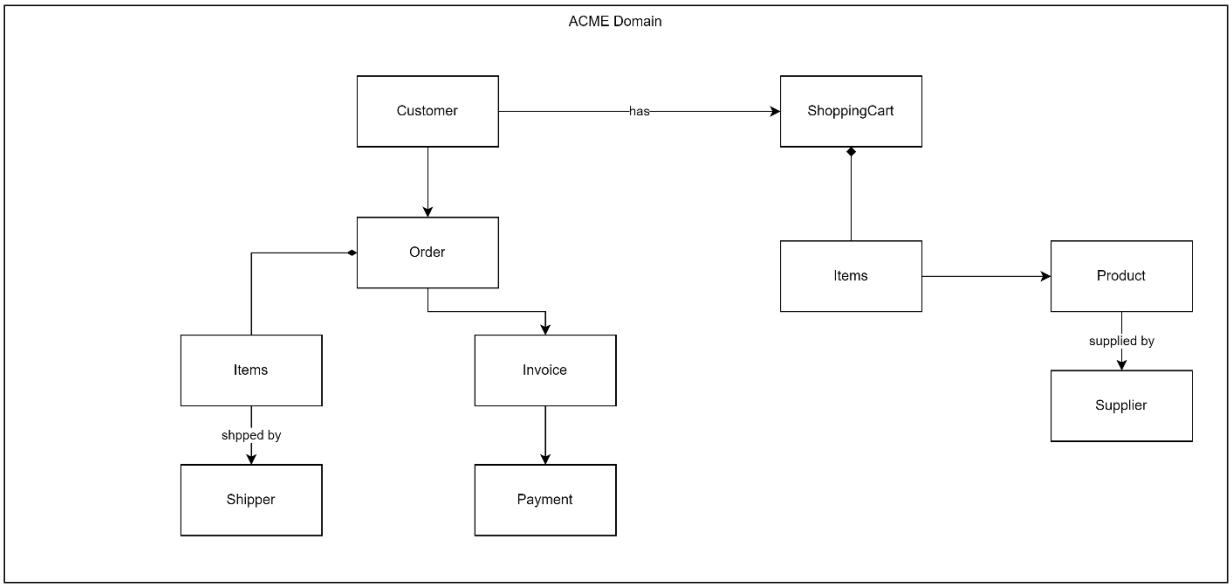


Figure 1 Acme Domain Model

Acme sells **Products** to **Customers**. **Suppliers** provide products**.** The Customer creates a **ShoppingCart** with the **Items** that will be bought. The Customer proceeds to an **Order** where the ShoppingCart Items promoted to Order **Items.** For each Order, an **Invoice** is created for the chosen **Payment** method (for example, credit card).

[5] https://en.wikipedia.org/wiki/Acme_Corporation