Full Stack Application Development

API

Akshaya Ganesan

Agenda

- REST Design
- Documentation
- Versioning

REST Design

- Both object-oriented design and database modeling techniques use domain entities as a basis for design.
- You can use the same technique to identify resources.
- Analyze your use cases to find domain nouns that can be operated using CRUD operations
- Discussion

  Consider a web service for managing photos. Clients can upload a new photo, replace an existing photo, view a photo, or delete a photo.

  In this example, “photo” is an entity in the application domain.

  The actions a client can perform on this entity include “create a new photo,” “replace an existing photo,” “view a photo,” and “delete a photo.”

REST Design

- *Provide a way for a client to get a user's profile with a minimal set of properties.*
- *List of the 10 latest photos uploaded by the user.*
- In all these use cases, it is easy to spot the nouns.
- But in each case you will find that the corresponding actions do not map to the basic CRUD operations
- Bluntly mapping domain entities into resources may lead to resources that are inefficient and inconvenient to use.
- You will need additional resources to tackle such use cases.

REST Design

- A resource doesn't have to be based on a single physical data item.
- For example, an order resource might be implemented internally as several tables in a relational database, but presented to the client as a single entity.
- Avoid creating APIs that simply mirror the internal structure of a database.
- The purpose of REST is to model entities and the operations that an application can perform on those entities.

REST Design

- Designing Collections
- A collection is a separate resource from the item within the collection, and should have its own URI.
- The relationships between different types of resources
- /customers/1/orders/99/products --- Cumbersome!!
- /customers/1/orders to find all the orders for customer 1
- /orders/99/products to find the products in this order.
- **Use HATEOAS to enable navigation to related resources**
- *collection/item/collection* is suitable, avoid more complexity

API Versioning

- Breaking changes primarily fit into the following categories:
  - Changing the request/response format (e.g. from XML to JSON)
  - Changing a property name (e.g. from name to productName) or data type on a property (e.g. from an integer to a float)
  - Adding a required field on the request (e.g. a new required header or property in a request body)
  - Removing a property on the response (e.g. removing description from a product)

API Change Management

- Effective change management in the context of an API is summarized by the following principles:
- Continue support for existing properties/endpoints
- Add new properties/endpoints rather than changing existing ones
- Thoughtfully sunset obsolete properties/endpoints

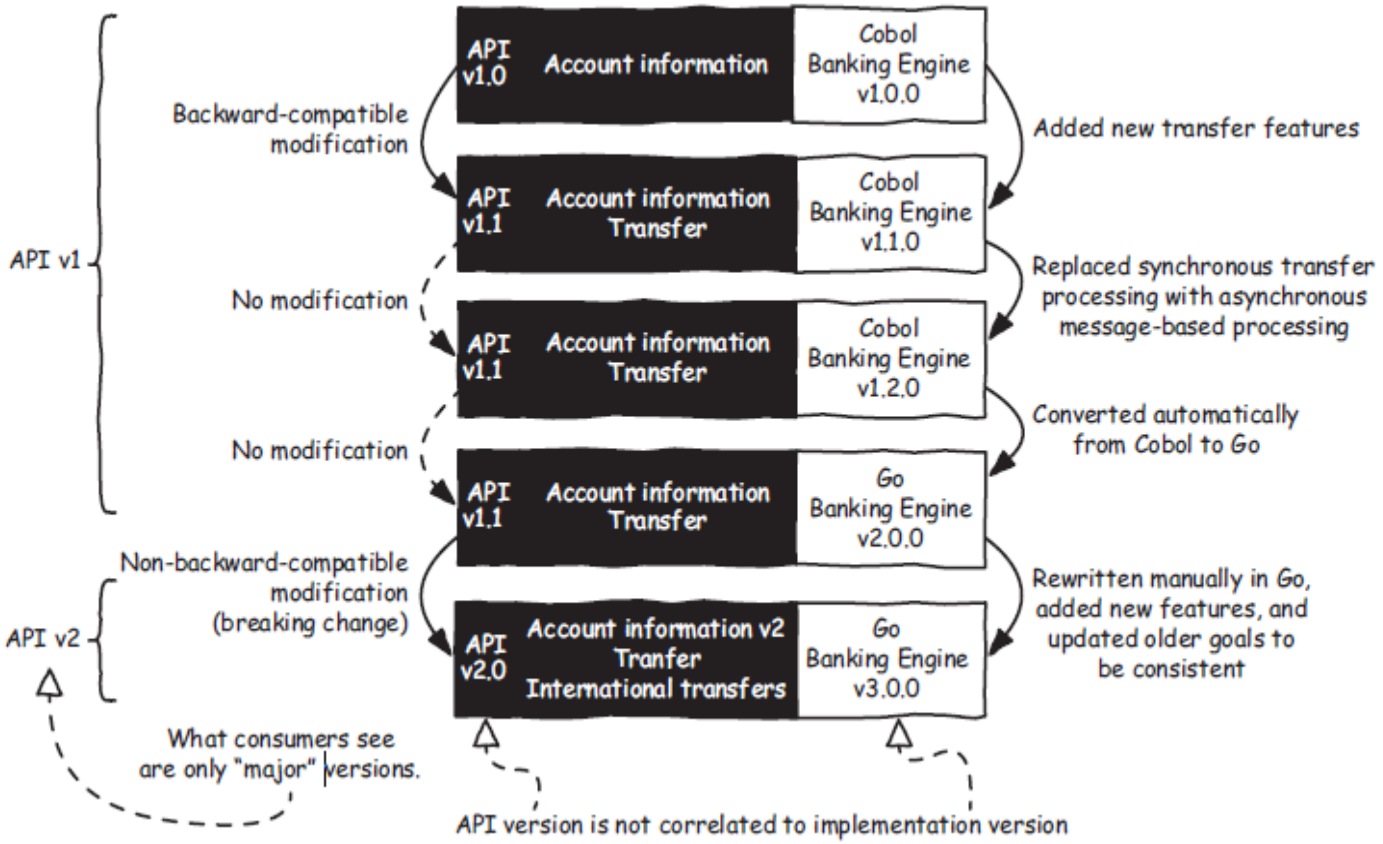
API Change Management

- We can think of levels of scope change within a tree analogy:
- Leaf - A change to an isolated endpoint with no relationship to other endpoints
- Branch - A change to a group of endpoints or a resource accessed through several endpoints
- Trunk - An application-level change, warranting a version change on most or all endpoints
- Root - A change affecting access to all API resources of all versions
- As you can see, moving from leaf to root, the changes become progressively more impactful and global in scope.

API versioning representation

- There are four different ways that we can implement the versioning
- **Versioning through the URI path**
  - http://www.example.org/v1/customer/1234
  - Ideal when major changes are introduced that completely break backward compatibility.
- **Versioning through query parameters**
  - http://www.example.org/customer/1234?version=v3
  - Ideal for simple APIs where backward compatibility is maintained across versions.
- **Versioning through custom headers**
  - Custom-Header: api-version=1
- **Versioning through content-negotiation**
  - Accept: application/vnd.adventure-works.v1+json

API Versioning



Semantic Versioning

- format: *MAJOR.MINOR.PATCH*.
- The MAJOR digit is incremented only on breaking changes, such as adding a new mandatory parameter
- The MINOR digit is incremented when new features are added in a backward-compatible manner, like adding new HTTP methods or resource paths in a REST API.
- The PATCH digit is incremented when the modifications made involve backward-compatible bug
- This makes sense for an implementation, but not for an API.
- Semantic versioning applied to APIs consist of just two digits: *BREAKING.NONBREAKING*.
- This two-level versioning is interesting from the provider's perspective; it helps to keep track of all the different backward-compatible and non-backward-compatible versions of an API.

API Documentation

- API documentation is a technical content deliverable containing instructions on using and integrating with an API effectively.
- API description formats like the OpenAPI/Swagger Specification have automated the documentation process, making it easier for teams to generate and maintain them.
- OpenAPI Specification (formerly Swagger Specification) is an API description format for REST APIs.

OAS

- The *OpenAPI Specication* (OAS) is a popular REST API description format
- An OpenAPI file allows you to describe your entire API, including:
  - Available endpoints (/users) and operations on each endpoint (GET /users, POST /users)
  - Operation parameters Input and output for each operation
  - Authentication methods
  - Contact information, license, terms of use and other information.
  - API specifications can be written in YAML or JSON.

Swagger

- **Swagger** is a set of open-source tools built around the OpenAPI Specification that can help you design, build, document and consume REST APIs.
- http://swagger.io/docs/specification/basic-structure/

API Documentation

- An OAS document describing the search for products goal of the Shopping API

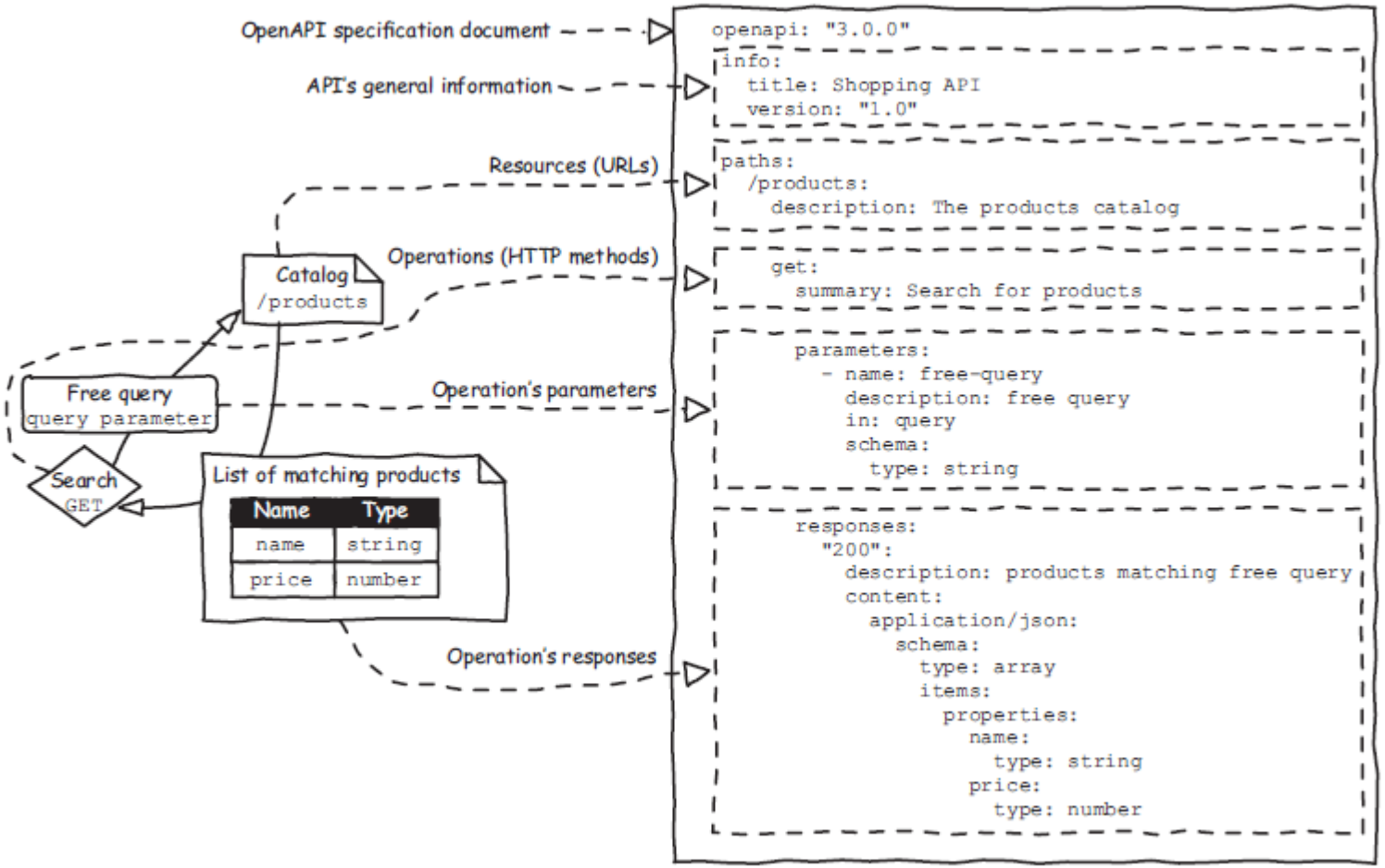
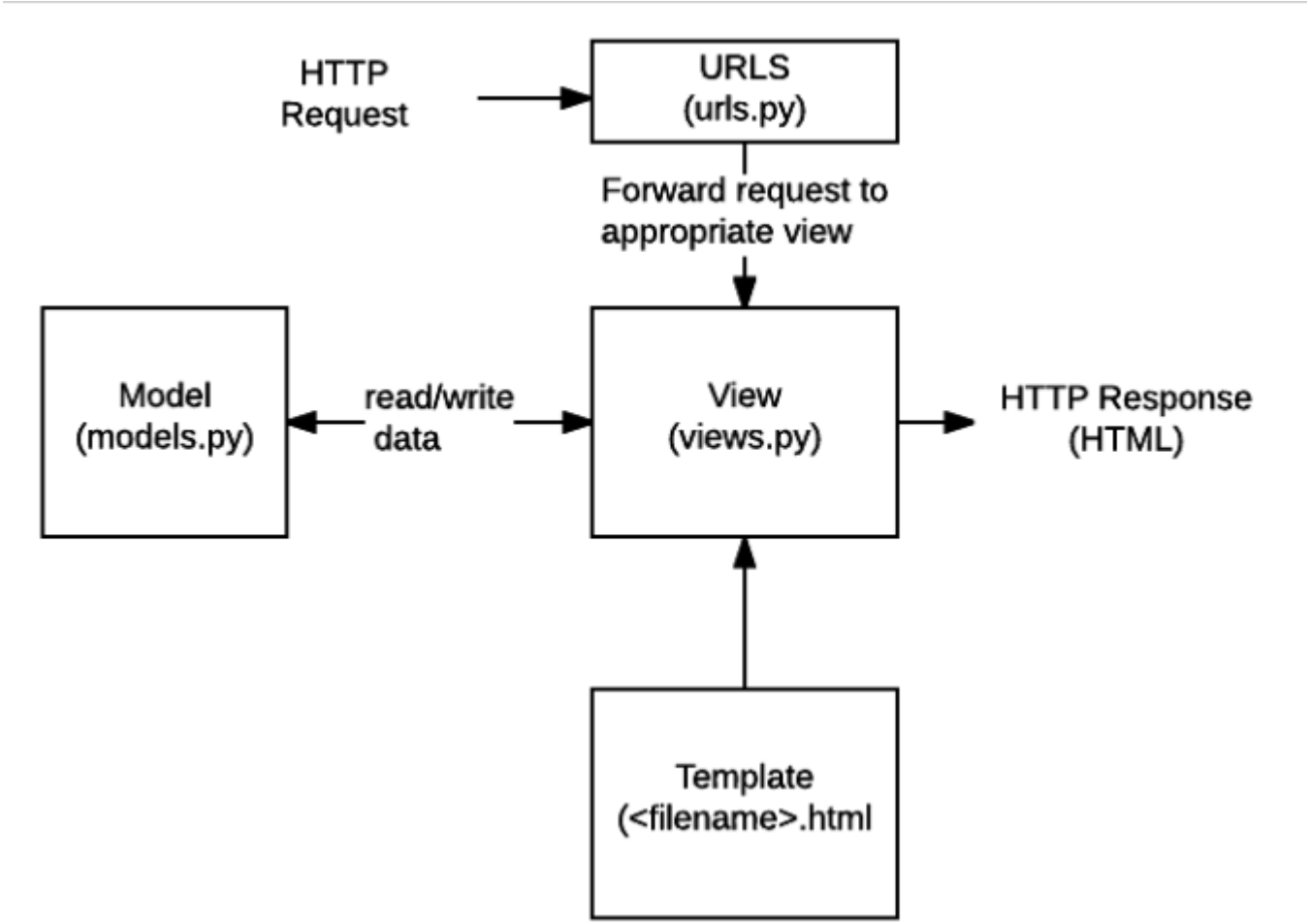


Image: The Design of Web APIs by ARNAUD LAURET

API Documentation

- RAML and Blueprint are other alternatives
- API Blueprint is a markdown format to generate documentation. It was developed by Apiary in 2013 and then acquired by Oracle.
- RAML, which stands for RESTful API Modeling Language, is an API design format developed by MuleSoft in 2013 and then acquired by Salesforce.
- Swagger is an API description format developed by Wordnik in 2010 and then acquired by SmartBear. It was later renamed OpenAPI and donated to the Linux Foundation.

Django Application



Thank You!