TUTORIAL

Designing a resource model for a Public API

O’Reilly Software Architecture Conference
New York – February 26th, 2018

Tom Höfte, IT Architect, Xebia
Marco van der Linden, IT Architect, Xebia
Agenda – 3.5 hours

- Introduction & API Domain discovery: ~ 20 minutes
- API Domain discovery: ~ 45 minutes
- REST Resource modeling: ~ 20 minutes
- Coffee Break: ~ 30 minutes (3pm)
- REST Resource modeling: ~ 60 minutes
- Other Topics & Conclusion: ~ 20 minutes (5pm)
Part 0 – Introduction
The API Economy

APIs allow companies to grow and extend businesses at unprecedented rates by sharing services with third parties – source: Harvard Business Review (https://hbr.org/2015/01/the-strategic-value-of-apis)

Note: Slide picture inspired by an idea of Willem van Schieveen, Bol.com
What we, *techies*, used to think an API is

With the rise of web technologies, the **term API** is now in general use and a *unique selling point* on the *web*
A public API is a business service with a web access point that is managed as a product.

We think a public API:

- is an open or partner service
- that exposes business capabilities
- through consumer-friendly operations
- that are used to build new applications
- accessible from a web access point
- and follows a product life cycle
A typical API architecture: A RESTful API as the channel to your domain

API Consumers

RESTful API Interface

Domain model

Business Domain Model containing data entities, logic and operations

Message-based Interface contract

Own devices, partners or open

But why REST?
An API needs a uniform interface, client-server decoupling and scalability. REST provides that.
... how do you create a REST API contract for your domain?

Describing complex domain behavior in a simple REST manner is not straightforward.
Example: Webshop

**Domain**

Customer
- internalId
- referenceId
- firstname
- lastname
- addressId
- status
- groupID

Address

**REST**

POST \customers

Create customer

> Domain Operations mapped to REST operations

Customer

Address

De-normalized in a single REST resource

Representation of required attributes in JSON, XML or ..

```json
{  
  "customer": {  
    "referenceId": "1234",  
    "firstName": "John",  
    "lastName": "Doe",  
    "Address": {   
    }  
  }  
}
```

```xml
<customer>  
  <referenceId>1234</referenceId>  
  <firstName>John</firstName>  
  <lastName>Doe</lastName>  
  <Address>...
  </Address>  
</customer>
```
So, resources are the core of a RESTful API...

Resource modeling is therefore a key activity to...

• Map your domain operations to REST operations
• Map your domain data entities to REST resource representations

...in order create an API that is user-friendly, extensible and maintainable
Resource Modeling Process

- Discover the API domain
- Discover the REST resources & operations
- Define the API interface
Part 1 – Discover the API domain
Goal

Practice how to discover API domain entities, relations and operations that you need in your API
Discover the API functionality

Outside API Model $\subset$ Inside Business Model
Use a Ubiquitous Language

Protest against any form of:

• Ambiguity
• Inconsistencies
Create bounded context

Bounded context helps:

• To logically group API behavior and form an API outline
• To structure the later implementation
Different methods

Input:
- Use case descriptions
- Domain experts, external and internal

Methods:
- Event storming or other work shops
- Text and domain analysis (verbs and nouns)

Output
- A logical domain model depicting
  - Set of nouns (entities) and verbs (operations) grouped in bounded contexts
  - Entity relations
Discover your API model
Example: Partial Event-Storming output Webshop

Shopping
- Search products
- View products in shopping cart
- Add product to shopping cart
- Checkout Shopping Cart

Ordering
- Create Order
- Approve Order
- Execute Payment
- Order
- OrderItem
- Product

Customers
- Add customer
- Change customer
- Customers
Discover your API model

Example: Partial Webshop domain model

**Shopping**

- Shopping Cart
- Product

**Ordering**

- Order
- OrderItem
- Product
- Payment
Exercise 1: Discover the API Domain
Exercise Input - Case Description: FlyWithMe airline

FlyWithMe is an airline that wants to increase sales by extending its current business model towards third parties. They plan to do this by providing a Public API.

You are an architect at FlyWithMe, responsible for designing the API. You and your team decide to start with discovering the domain entities and operations...

...more input is provided in the hand-outs
Exercise outcome

- List of domain entities
- List of domain operations
- Grouped by bounded context
- Time left? Try to define the entity relations
Tips

➢ Don’t try to over-complicate the business domain; the goal is not to get a complete understanding of the domain.
➢ Make assumptions about the business domain where needed
➢ Focus on API consumer functionality
➢ Do not focus on the attributes of an entity, solely focus on the *nouns* and *verbs*
And now to work!

- Organize yourselves in groups of 3-4
- Work on exercise till 14:45
- Make use of whiteboard, paper, flip overs (easel pad) etc. to visualize your answer.

Retro time

- What went well?
- What was difficult?
Part 2 – REST resource modeling
Goal

- Learn how to map domain entities and operations to REST resources and operations
- Learn about API documentation options
- Practice by modelling basic use cases from the case study
REST API Resource Modeling Process

1. Discover the API domain
2. Discover the REST resources & operations
3. Define the API interface

The process is iterative, as indicated by the circular arrows.
From a domain model to a REST contract

A typical flow

**Step 1 – Define sequence diagrams**
- Group related domain operations in user stories
- Map domain operations and entities to REST resources and operations

**Step 2 – Define the API in a specification language**
- Determine the resource representations
- Operation granularity
REST API specification language

- Start defining the API using a API description languages:
  - Open API (fka Swagger)
  - RAML
  - API Blueprint

- Kickstart your development with an API design tool
  - SwaggerHub
  - Apiary for API blueprint
  - Confluence + swagger.io plugin
Anatomy of a RESTful API

**REST** = Representational State Transfer

Resource location = URL

Resource Operations = VERBS = GET | POST | PUT | DELETE

Resources representations = NOUNS representations = Media Types (JSON, XML)

resources = NOUNS
REST resource types

- **Document**: A single document with optional subresources
  - https://api.flywithme.com/flightoffers/{id}

- **Collection**: A server managed collection: a plural noun
  - https://api.flywithme.com/airports
REST Operations - Standard HTTP operations

Retrieve resources
GET https://api.flywithme.com/airports/{airportid}

Create a new resource
POST https://api.flywithme.com/bookings body: trip

Update an existing resource
PUT https://api.flywithme.com/bookings/{bookingid} body: updated trip

Delete an existing resource
DELETE https://api.flywithme.com/bookings/{bookingid}
REST Operations – non-CRUD operations

- Sometimes it can be difficult to model a business capability spanning multiple resources with fine-grained HTTP CRUD operations.
  - A business capability is a long-running process
  - A business capability is a synchronous function

- Add non-CRUD verbs or noun-ified verbs to your API
REST Operations – non-CRUD Verbs

- A verb in your URL representing a synchronous action or function
  - Typically modelled with GET

**GET /properties/availability/{propertyId}/rooms/{roomId}/rates/{rateId}/price-check**

Get Current Price for Pre-Booking

Confirms the price returned by the Shop response. Use this API to verify a previously-selected rate is still valid before booking. If the price is matched, the response returns a link to request a booking. If the price has changed, the response returns new price details and a booking link for the new price. If the rate is no longer available, the response will return a new shop request link to search again for different rates.

Expedia API: https://developer.ean.com/documentation/rapid-shopping-docs
REST Operations – Noun-ified verbs

- A noun representing an asynchronous action
  - Long running
  - Action can be monitored
  - Typically modelled with HTTP POST
  - REST without PUT / CQRS

Create a fork

Create a fork for the authenticated user.

POST /repos/:owner/:repo/forks

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>organization</td>
<td>string</td>
<td>Optional parameter to specify the organization name if forking into an organization.</td>
</tr>
</tbody>
</table>

Response

Forking a Repository happens asynchronously. Therefore, you may have to wait a short period before accessing the git objects. If this takes longer than 5 minutes, be sure to contact GitHub support.

Source: GitHub API
REST Operations – Fine-grained HTTP CRUD operations or course-grained non-CRUD operations?

It depends:

- **CRUD** should be your first option
- Use **Non-CRUD verbs for synchronous operations** for which non-CRUD becomes clumsy
- Use **noun-ified verbs for long running actions** or events that must be monitored
REST Resource Relations

Order

Customer

OrderItem

Order

existentially dependent

non-existentially dependent
REST Resource Relations - Modeling

Resource modeling options:

- **Linked Resources**
  ```json
  {  
    orderId: 1  
    customerId: 1  
  }
  ``

- **Embedded resources**
  ```json
  {  
    id: 1  
    orderItems: [  
      {  
        id: 1  
        product: "X"  
      }  
    ]  
  }
  ```

Path modeling rules:

- existentially dependent
  ```html
  \orders\{orderid\}\orderitems\{orderitemid\}
  ```

- non-existentially dependent
  ```html
  \customers\{customerid\}
  ```
REST Resource Representations – Media Types

- Mime-types used most often in the Content-Type header:
  - application/json
  - application/xml

- But binary content is also possible:
  - Content-Type: application/pdf

- Or create your own custom type:
  - Content-Type: application/vnd.amazon.ebook
REST operation response codes

Reuse HTTP response code semantics

- **Range 2xx** – Client message has been received, understood and processed successfully
  - E.g. Use HTTP 201 when a resource is created
- **Range 3xx** – Location redirection
- **Range 4xx** – Request could not be processed due to an error caused by the client
- **Range 5xx** – Request could not be processed due to an error caused by the server
Exercise 2: Discover REST resources and operations
Exercise input

- Take the output of exercise 1 and ...
  - Take your team result
  - Or take our example result

- Look at the 2 user stories (*) and define a RESTful API by
  - Creating a sequence flow diagram
  - A Swagger file specification

(*) User stories will be provided in the exercise
Exercise outcome

For each of the 2 user stories:

- Sequence flow diagrams highlighting the REST operations and resources
- An initial swagger file specifying the REST operations and resources needed to implemented the user stories
And now to work!

› Work on the exercise till 16:30
› Make use of
  - https://www.websequencediagrams.com
  - https://editor.swagger.io
  - https://swagger.io/specification/

Coffee Break
Running out of material, but not out of time?

You can add the following features:

- Notify partner websites with flight updates
- Update bookings
Retro time

- What went well?
- What was difficult?
Part 3 – Other *Important* Topics
Hypermedia

- **HATEOAS** (Hypermedia as the Engine of Application State) – Richardson Maturity Model – Level 3
- Self describing, discoverable API
- Changes do not break the contract

Some frameworks / standardisation efforts:
- Hypertext Application Language – HAL
- Structured Interface for Representing Entities – SIREN
- JSON-LD and Hydra
- JSON-API

```json
{ "flightOfferSearch": 1 "offset": 0 |
  "flightoffer": 1, "id": 2 "accommodations", |
  "href": "1/services", "rel": "services", "type": "GET"
}, |
  "href": "1/optionalAccommodations", "rel": "optionalAccommodations", "type": "GET"
} |
"links": |
{ "href": "1/flightoffers/next" "rel": nextOffset "type": "GET"
} |
"links": |
```

Possible resource operations defined in the response
Evolving the API

Changes will happen, so design for them.

- By using hypermedia or ...
- Support versioning, but only on MAJOR versions
  - Make backwards compatible changes and avoid breaking changes
  - Deprecate but leave in old functionality
Webhook pattern

- Event-driven integration via server callbacks
  - Flight updates
  - Price changes

- Polling is bad for everyone
  - ~95% of polling requests are useless
  - Reduce load on server
Process & People

- API Strategy
- API revenue model
- API team
- API product development
What have we learned?
Learning points

- A RESTful API interface <> your domain model
- Embrace non-CRUD operations
- Don’t be afraid of changes, but facilitate them in your API design and in communication to your API consumers
- Be pragmatic and work iteratively
- Don’t forget about the people and the process
TUTORIAL

Thank you for attending!

Tom Höfte, IT Architect, Xebia, thofte@xebia.com
Marco van der Linden, IT Architect, Xebia, mvanderlinden@xebia.com