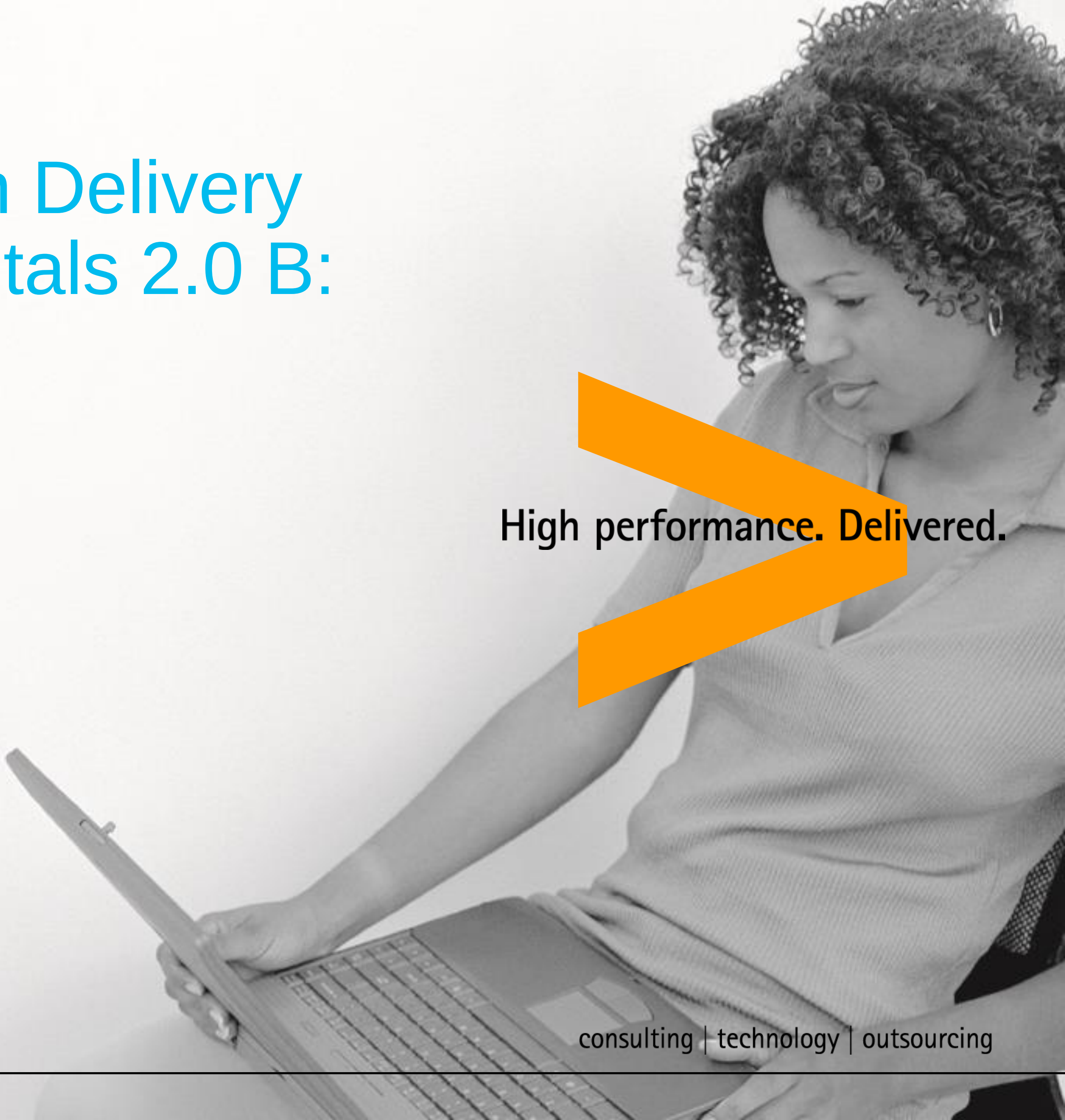
Application Delivery Fundamentals 2.0 B: Java

API Design


High performance. Delivered.
consulting | technology | outsourcing

API Design

Goals

- Web API Design - Crafting Interfaces for Developers
- Nouns are good; verbs are bad
- Plural nouns and concrete names
- Simplify associations - sweep complexity under the ‘?’
- Handling errors
- Pagination and partial response
- What about responses that don’t involve resources?

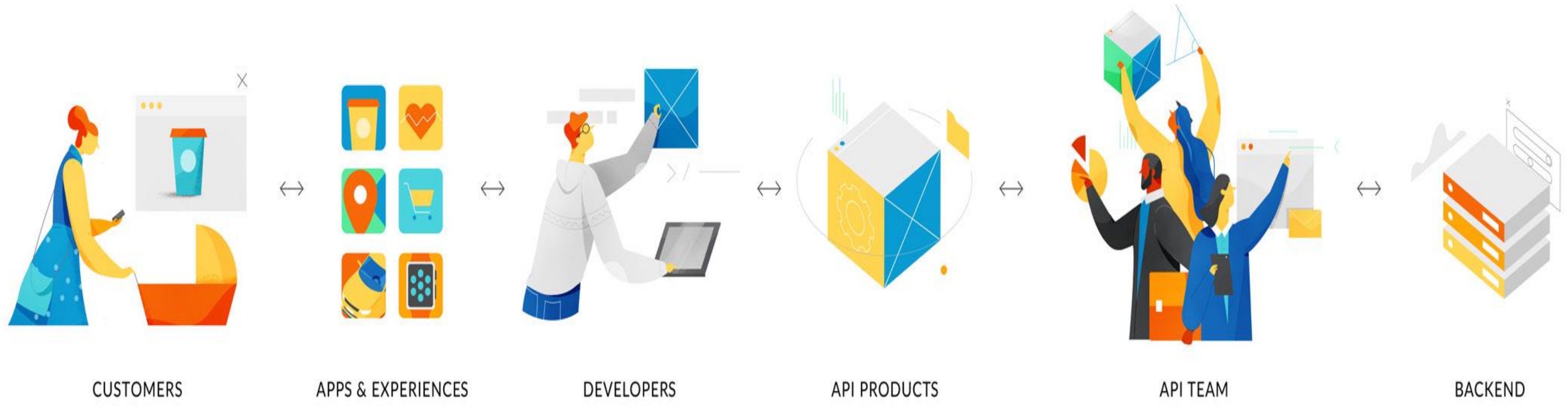
API Design

Goals

- Supporting multiple formats
- What about attribute names?
- Consolidate API requests in one subdomain
- Authentication
- Making requests on your API
- Chatty APIs
- The API Façade Pattern

Web APIs and REST

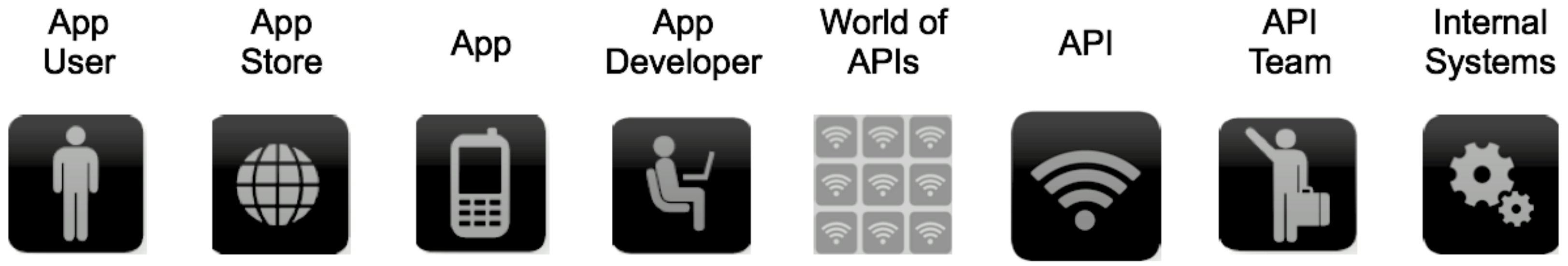
- A web API is the pattern of HTTP requests and responses.
- Used to access a website that is specialized for access by arbitrary computer programs.
- REST is the name that has been given to the architectural style of HTTP itself, described by one of the leading authors of the HTTP specifications.
- HTTP is the reality—REST is a set of design ideas that shaped it.
- From a practical point of view, we can focus our attention on HTTP and how we use it to develop APIs.
- The importance of REST is that it helps us understand how to think about HTTP and its use.



Web API Design - Crafting Interfaces for Developers

- The primary design principle when crafting your API should be to maximize developer productivity and success. This is what we call pragmatic REST.

- **Pragmatic REST is a design problem**
  - You have to get the design right, because design communicates how something will be used.



How to design or build great Web API Applications?

- Basically we’re building applications (web, windows, etc...) for end users who are not programmers, who can easily use your application.
- Here, they are just end users for our application but API is designed only for programmers who can consume our great API and perform whatever operations they want for their applications based on the requirement.

API implementation point of view

- What does this service need to do?
- What does this service need to provide?
- How my API will act as more generic (input, output and extensible)?

API consumer point of view:

- How can consumers assuredly integrate with our API?
- How nimbly, they can provide input to us and output for their use from our API?
- How can consumers spend the bare minimum of effort to get what they need out of this API?

Guidelines for designing a great Web API

1. Documentation
2. Naming Conventions for Controllers and Actions
3. Don’t change the state of HTTP Methods
4. Stability and Consistency
5. Flexibility
6. Security
7. Validation
8. HTTP Status Codes
9. Web API Helper page
10. Logging

Documentation:

- Documentation is very important for Web API’s because in this, each method is describing two things i.e., what it’ll accept and what it’ll provide.
- As a developer, I worked on consuming several external web API’s like mail chimp, klaviyo, sales force, bronto and etc. these all are email service providers.
- My Requirement is to get a list of emails from each web API.
- I have many API's to go through and it needs to integrate each of them.
- If each API will take only a minimum amount of time to understand what API is doing and how I can get the result then it’s a great API because of well marked documentation.
- A few API's are hell; I need to spend more time on it because of poor documentation.

Documentation:

- Here I am thinking like an API user. So every Web API developer should think from the user's perspective too.
- Although you are building great API's for end users, if they are facing a lot of difficulties or spending more time to integrate/understand it, then it’s not a great API.

Keep your base URL simple and intuitive

- The base URL is the most important design affordance of your API.
- A simple and intuitive base URL design makes using your API easy.
- Affordance is a design property that communicates how something should be used without requiring documentation.
- A door handle's design should communicate whether you pull or push.
- Here's an example of a conflict between design affordance and documentation - not an natural interface!

Keep your base URL simple and intuitive

Keep your base URL simple and intuitive

- A key litmus test we use for Web API design is that there should be only 2 base URLs per resource.
- Let's model an API around a simple object or resource, a product, and create a Web API for it.
- The first URL is for a collection; the second is for a specific element in the collection.
- /products /products/1234
- Boiling it down to this level will also force the verbs out of your base URLs.

Naming Conventions for Controllers and Actions

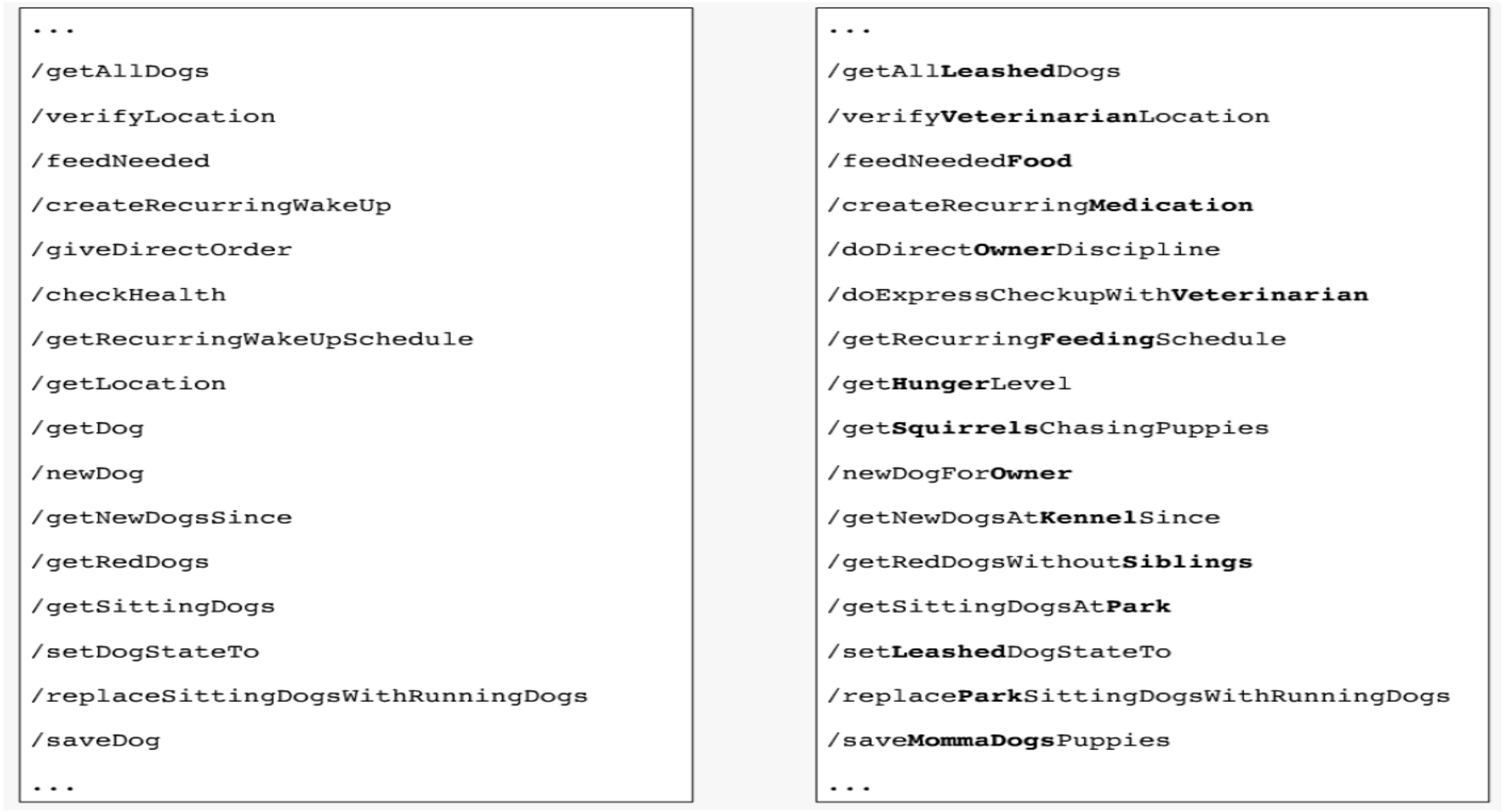
- Use plural nouns for Controllers
- Don’t mix up singular and plural nouns.
- Keep it simple and use only plural nouns for all resources.
- Example:
- /products instead of /product
- /users instead of /user

Keep verbs out of your base URLs

- Many Web APIs start by using a method-driven approach to URL design.
- These method-based URLs sometimes contain verbs - sometimes at the beginning, sometimes at the end.
- For any resource that you model, like our dog, you can never consider one object in isolation.
- Rather, there are always related and interacting resources to account for - like owners, veterinarians, leashes, food, squirrels, and so on.

Keep verbs out of your base URLs

Think about the method calls required to address all the objects in the dogs' world. The URLs for our resource might end up looking something like this.



...

/getAllDogs

/verifyLocation

/feedNeeded

/createRecurringWakeUp

/giveDirectOrder

/checkHealth

/getRecurringWakeUpSchedule

/getLocation

/getDog

/newDog

/getNewDogsSince

/getRedDogs

/getSittingDogs

/setDogStateTo

/replaceSittingDogsWithRunningDogs

/saveDog

...

...

/getAll**Leashed**Dogs

/verify**Veterinarian**Location

/feedNeeded**Food**

/createRecurring**Medication**

/doDirect**Owner**Discipline

/doExpressCheckupWith**Veterinarian**

/getRecurring**Feeding**Schedule

/get**Hunger**Level

/get**Squirrels**ChasingPuppies

/newDogFor**Owner**

/getNewDogsAt**Kennel**Since

/getRedDogsWithout**Siblings**

/getSittingDogsAt**Park**

/set**Leashed**DogStateTo

/replace**Park**SittingDogsWithRunningDogs

/save**MommaDogs**Puppies

...

Naming Conventions for Controllers and Actions

- **Use nouns but no verbs for Actions**
- Don’t use fully described names for actions. It’s a bad practice.

	GET	POST	PUT	DELETE
RESOURCE	Read	Create	Update	Delete
/Products	Returns a list of products	Create a new product	Bulk update of products	Delete all products
/Products/101	Returns a specific product	Method not allowed (405)	Updates a specific product	Deletes a specific product

Naming Conventions for Controllers and Actions

- **Don’t use verbs:**

  /getAllProducts
  /createNewProduct
  /deleteAllProducts .

Naming Conventions for Controllers and Actions

- Don’t change the state of HTTP Methods
- HTTP methods have their own state of behavior. Don’t change or mix their state.

GET	**POST**	**PUT**	**DELETE**	
RESOURCE	Read	Create	Update	Delete

Plural nouns and concrete names

- Should you choose singular or plural nouns for your resource names?
- Let's look at a few examples:
- Foursquare GroupOn Zappos
- **/checkins /deals /Product**
- Given that the first thing most people probably do with a RESTful API is a GET, we think it reads more easily and is more intuitive to use plural nouns.
- But above all, avoid a mixed model in which you use singular for some resources, plural for others.
- Being consistent allows developers to predict and guess the method calls as they learn to work with your API.

Concrete names are better than abstract

- is not always meaningful for developers.
- Take for example an API that accesses content in various forms - blogs, videos, news articles, and so on.
- An API that models everything at the highest level of abstraction - as /items or /assets in our example - **loses the opportunity to paint a tangible picture for developers to know what they can do with this API.**
- It is more compelling and useful to see the resources listed as blogs, videos, and news articles.

Concrete names are better than abstract

- The level of abstraction depends on your scenario.
- You also want to expose a manageable number of resources.
- Aim for concrete naming.
- In summary, an intuitive API **uses plural** rather than singular nouns, and **concrete** rather than abstract names.

Stability and Consistency

- API is more successful and it has a lot of clients.
- Now, you have a requirement to rename one of the fields or add a new field for JSON response.
- What happens if you modify the changes and publish a web API?
- Everyone that’s already integrated with you is going to break.
- In the software world, we can't blame changes or new feature requests.
- They are part of the software development. We can't stop them or refuse them either.
- The most common way of handling changes is, to have versions of an API.

Versioning

- Changes that don’t break existing code using the API can be handled within one version.
- If there are breaking changes i.e. it risks breaking the code which is using the API, those changes should be introduced in a new version of the API.
- Every developer should do some planning ahead of your web API.
- Make the API Version mandatory and never release an unversioned API.

Versioning

- **Never release an API without a version and make the version mandatory.**
- Let's see how three top API providers handle versioning.
- Twilio /2010-04-01/Accounts/
- salesforce.com /services/data/v20.0/sobjects/Account
- Facebook ?v=1.0

Versioning

- Twilio uses a timestamp in the URL (note the European format).
- At compilation time, the developer includes the timestamp of the application when the code was compiled.
- That timestamp goes in all the HTTP requests.
- When a request arrives, Twilio does a look up.
- Based on the timestamp they identify the API that was valid when this code was created and route accordingly.
- For example, it can be confusing to understand whether the timestamp is the compilation time or the timestamp when the API was released.

Versioning

- Salesforce.com uses v20.0, placed somewhere in the middle of the URL.
- We like the use of the v. notation.
- However, we don't like using the .0 because it implies that the interface might be changing more frequently than it should.
- The logic behind an interface can change rapidly but the interface itself shouldn't change frequently.

Versioning

- Facebook also uses the v. notation but makes the version an optional parameter.
- This is problematic because as soon as Facebook forced the API up to the next version, all the apps that didn't include the version number broke and had to be pulled back and version number added.

How to think about version numbers in a pragmatic way with REST?

- Never release an API without a version. Make the version mandatory.
- Specify the version with a 'v' prefix. Move it all the way to the left in the URL so that it has the highest scope (e.g. /v1/products).
- Use a simple ordinal number.
- Don't use the dot notation like v1.2 because it implies a granularity of versioning that doesn't work well with APIs--it's an interface not an implementation.
- Stick with v1, v2, and so on.

How to think about version numbers in a pragmatic way with REST?

- How many versions should you maintain? Maintain at least one version back.
- For how long should you maintain a version? Give developers at least one cycle to react before obsoleting a version.
- Sometimes that's 6 months; sometimes it’s 2 years.
- It depends on your developers' development platform, application type, and application users.
- For example, mobile apps take longer to rev’ than Web apps

Should version and format be in URLs or headers?

- There is a strong school of thought about putting format and version in the header.
- Simple rules we follow:
  - If it changes the logic you write to handle the response, put it in the URL so you can see it easily.
  - If it doesn't change the logic for each response, like OAuth information, put it in the header

Should version and format be in URLs or headers?

- These for example, all represent the same resource:
- products/1
  - Content-Type: application/json
- products/1
  - Content-Type: application/xml
- products/1
  - Content-Type: application/png
- The code we would write to handle the responses would be very different.

Versioning

- **Sample of URL version schema:**
  /api/v1/products
  /api/products?version=v1
  /api/products?api-version=1
- /api/v2-Alpha/products
  /api/products?api-version=2-Alpha
-
  /api/v2015-05-01.3.0/products
  /api/products?api-version=2015-05-01.3.0

Flexibility

- Flexibility is most important for Web API’s.
- Let’s say, if you have a lot of users for your web API, users may needed output as JSON format or XML format as per their business needs.
- So, you can’t say that my API will always return output as JSON.
- If you say that, you’ll lose your business.
- How will you manage that?
- How should your API act flexible (about input, output)?

Flexibility

- 5.1) Flexibility on Input:
  - By default, Web API will support multiple formats of input types. Those are text/html, Json, XML, etc.
- 5.2) Flexibility on Output:
  - The below formats are giving flexibility to end users to choose their output format.
  - Example Resources:
  - /api/v1/products.Json
  - /api/v1/products.XML
  - /api/v1/products/?format=Json
  - /api/v1/products?format=XML

Flexibility

- 5.3) Flexibility on Filtering:
  - Use a unique query parameter for all fields or a query language for filtering.
  - GET /api/v1/products?price>1000 Returns a list of products(Id, Name, Description, Code, Price), which price is greater than 1000
  - GET /api/v1/products?code=P123 Returns a list of products(Id, Name, Description, Code, Price) which code is "P123”
  - If you provide multiple filters, you can only return resources that match all filters.

Flexibility

- 5.4) Flexibility on Field selection:
  - The above get request, gives a list of products along with all fields (Id, Name, Description, Code, Price).
  - Let’s say, your API will be consumed by multiple users.
  - Those users may be desktop, tablet, or mobile developers. Desktop users need all fields of products, tablet users need some fields and mobile users need only few fields.
  - They don’t need all attributes of a resource all the time.
  - So, give the API consumer the ability to choose returned fields.
  - This will also reduce the network traffic and speed up the usage of the API.
  - GET /api/v1/products?fields=name,price

Flexibility

- 5.4) Flexibility on Field selection:
  - The above get request, gives a list of products along with all fields (Id, Name, Description, Code, Price).
  - Let’s say, your API will be consumed by multiple users.
  - Those users may be desktop, tablet, or mobile developers. Desktop users need all fields of products, tablet users need some fields and mobile users need only few fields.
  - They don’t need all attributes of a resource all the time.
  - So, give the API consumer the ability to choose returned fields.
  - This will also reduce the network traffic and speed up the usage of the API.
  - GET /api/v1/products?fields=name,price

Flexibility

- **5.5) Pagination**
  - Paginate your API requests to limit response results and make them easier to work with.
  - We use offset and count in the URL query string to paginate because it provides greater control over how you view your data.

Flexibility

- **5.5) Pagination**
  - Make it easy for developers to paginate objects in a database
  - It's almost always a bad idea to return every resource in a database.
  - Let's look at how Facebook, Twitter, and LinkedIn handle pagination.
  - Facebook uses offset and limit.
  - Twitter uses page and rpp (records per page).
  - LinkedIn uses start and count
  - Semantically, Facebook and LinkedIn do the same thing. That is, the LinkedIn start & count is used in the same way as the Facebook offset & limit.

Flexibility

- **5.5) Pagination**
  - To get records 50 through 75 from each system, you would use:
  - Facebook - offset 50 and limit 25
  - Twitter - page 3 and rpp 25 (records per page)
  - LinkedIn - start 50 and count 25
  - Use limit and offset
  - We recommend limit and offset. It is more common, well understood in leading databases, and easy for developers.
  - /products?limit=25&offset=50

Flexibility

- **5.5) Pagination**
  - Metadata
  - We also suggest including metadata with each response that is paginated that indicated to the developer the total number of records available.
  - What about defaults?
  - My loose rule of thumb for default pagination is limit=10 with offset=0. (limit=10&offset=0)
  - The pagination defaults are of course dependent on your data size.
  - If your resources are large, you probably want to limit it to fewer than 10; if resources are small, it can make sense to choose a larger limit.

Partial Responses

- Partial response allows you to give developers just the information they need.
- Take for example a request for a tweet on the Twitter API.
- You'll get much more than a typical twitter app often needs - including the name of person, the text of the tweet, a timestamp, how often the message was re-tweeted, and a lot of metadata.
- Let's look at how several leading APIs handle giving developers just what they need in responses, including **Google who pioneered the idea of partial response.**

Partial Responses

- LinkedIn
- /people:(id,first-name,last-name,industry)
  - This request on a person returns the ID, first name, last name, and the industry.
- Facebook
  - /joe.smith/friends?fields=id,name,picture
- **Google**
  - ?fields=title,media:group(media:thumbnail)
- Google and Facebook have a similar approach, which works well.
- They each have an optional parameter called fields after which you put the names of fields you want to be returned.

Partial Responses

- **Add optional fields in a comma-delimited list**
  - The Google approach works extremely well.
  - Here's how to get just the information we need from our products API using this approach:
  - /products?fields=name,color,price
  - It's simple to read; a developer can select just the information an app needs at a given time; it cuts down on bandwidth issues, which is important for mobile apps.

What about responses that don’t involve resources?

- API calls that send a response that's not a resource per se are not uncommon depending on the domain.
- Actions like the following are your clue that you might not be dealing with a "resource" response.
  - Calculate
  - Translate
  - Convert
  - For example, you want to make a simple algorithmic calculation like how much tax someone should pay, or do a natural language translation (one language in request; another in response), or convert one currency to another.
  - **None involve resources returned from a database**.

What about responses that don’t involve resources?

- Use verbs not nouns
- For example, an API to convert 100 euros to Chinese Yen:
- /convert?from=EUR&to=CNY&amount=100
- Make it clear in your API documentation that these “**non-resource**” scenarios are different.
- Simply separate out a section of documentation that makes it clear that you use verbs in cases like this – where some action is taken to generate or calculate the response, rather than returning a resource directly.

Flexibility

- **5.6) Sorting:**
- Allow ascending and descending sorting over multiple fields.
- GET /api/v1/products?sort=-price

- Retrieves a list of products in descending order of price

- GET /api/v1/products?sort=+price,name
- Retrieves a list of products in ascending order of price and name

Simplify associations - sweep complexity under the ‘?’

- Associations Resources almost always have relationships to other resources.
- What's a simple way to express these relationships in a Web API?
- Let's look again at the API we modeled in nouns are good, verbs are bad - the API that interacts with our products resource.
- Remember, we had two base URLs: /products and products/1234.

Simplify associations - sweep complexity under the ‘?’

- We're using HTTP verbs to operate on the resources and collections.
- Our products belong to categories.
- To get all the products belonging to a specific category, or to create a new product for that category, do a GET or a POST:
- GET /categories/5678/products
- POST /categories/5678/products

Simplify associations - sweep complexity under the ‘?’

- Now, the relationships can be complex.
- Categories have relationships with suppliers, who have relationships with products, who have relationships with brand, and so on.
- It's not uncommon to see people string these together making a URL 5 or 6 levels deep.
- Remember that once you have the primary key for one level, you usually don't need to include the levels above because you've already got your specific object.
- In other words, you shouldn't need too many cases where a URL is deeper than what we have above /resource/identifier/resource.

Sweep complexity behind the ‘?’

- Most APIs have intricacies beyond the base level of a resource.
- Complexities can include many states that can be updated, changed, queried, as well as the attributes associated with a resource.
- Make it simple for developers to use the base URL by putting optional states and attributes behind the HTTP question mark.
- To get all Citi Products available in US:
- GET /products?bank=citi&state=running&location=NJ
- In summary, keep your API intuitive by simplifying the associations between resources, and sweeping parameters and other complexities under the rug of the HTTP question mark.

Handling errors

- Why is good error design especially important for API designers?
- From the perspective of the developer consuming your Web API, everything at the other side of that interface is a black box.
- Errors therefore become a key tool providing context and visibility into how to use an API.
- First, developers learn to write code through errors.
- The "test-first" concepts of the extreme programming model and the more recent "test driven development" models represent a body of best practices that have evolved because this is such an important and natural way for developers to work.

Handling errors

- Secondly, in addition to when they're developing their applications, developers depend on well-designed errors at the critical times when they are troubleshooting and resolving issues after the applications they've built using your API are in the hands of their users.

Handling errors

- **How to think about errors in a pragmatic way with REST?**
- Let's take a look at how three top APIs approach it.
- **Facebook**
- **HTTP Status Code: 200**
- **{"type" : "OauthException", "message":"(#803) Some of the aliases you requested do not exist: foo.bar"}**

Handling errors

- **Twilio**
  - **HTTP Status Code: 401**
  - **{"status" : "401", "message":"Authenticate","code": 20003, "more info": "http://www.twilio.com/docs/errors/20003"}**
  - **SimpleGeo**
  - **HTTP Status Code: 401**
  - **{"code" : 401, "message": "Authentication Required"}**
- **SimpleGeo** SimpleGeo provides error codes but with no additional value in the payload.

A couple of best practices

- Use HTTP status codes Use HTTP status codes and try to map them cleanly to relevant standard-based codes.
- There are over 70 HTTP status codes.
- However, most developers don't have all 70 memorized.
- **So if you choose status codes that are not very common** you will force application developers away from building their apps and over to Wikipedia to figure out what you're trying to tell them.

A couple of best practices

Therefore, most API providers use a small subset. For example, the Google GData API uses only 10 status codes; Netflix uses 9, and Digg, only 8.

Google GData

200 201 304 400 401 403 404 409 410 500

Netflix

200 201 304 400 401 403 404 412 500

Digg

200 400 401 403 404 410 500 503

How many status codes should you use for your API?

- When you boil it down, there are really only 3 outcomes in the interaction between an app and an API:
  - Everything worked - success
  - The application did something wrong – client error
  - The API did something wrong – server error

How many status codes should you use for your API?

- Start by using the following 3 codes. If you need more, add them. But you shouldn't need to go beyond 8.
  - **200 - OK**
  - **400 - Bad Request**
  - **500 - Internal Server Error**
- If you're not comfortable reducing all your error conditions to these 3, try picking among these additional 5:
  - **201 - Created**
  - **304 - Not Modified**
  - **404 – Not Found**
  - **401 - Unauthorized**
  - **403 - Forbidden**

Make messages returned in the payload as verbose as possible

- Code for code
  - **200 – OK 401 – Unauthorized**
- Message for people
  - {"developerMessage" : "Verbose, plain language description of the problem for the app developer with hints about how to fix it.", "userMessage":"Pass this message on to the app user if needed.", "errorCode" : 12345, "more info": "http://dev.teachdogrest.com/errors/12345"}

Supporting multiple formats

- We recommend that you support more than one format - that you push things out in one format and accept as many formats as necessary.
- You can usually automate the mapping from format to format.
- Here's what the syntax looks like for a few key APIs.
- Google Data?alt=json
- **Foursquare/venue.json**
- Digg*
- Accept: application/json ?type=json

What about attribute names?

- **Here are API responses from a few leading APIs:**
- Twitter "created_at": "Thu Nov 03 05:19;38 +0000 2011"
- Bing "DateTime": "2011-10-29T09:35:00Z"
- Foursquare "createdAt": 1320296464
- They each use a different code convention.
- **Foursquare has the best approach**

What about attribute names?

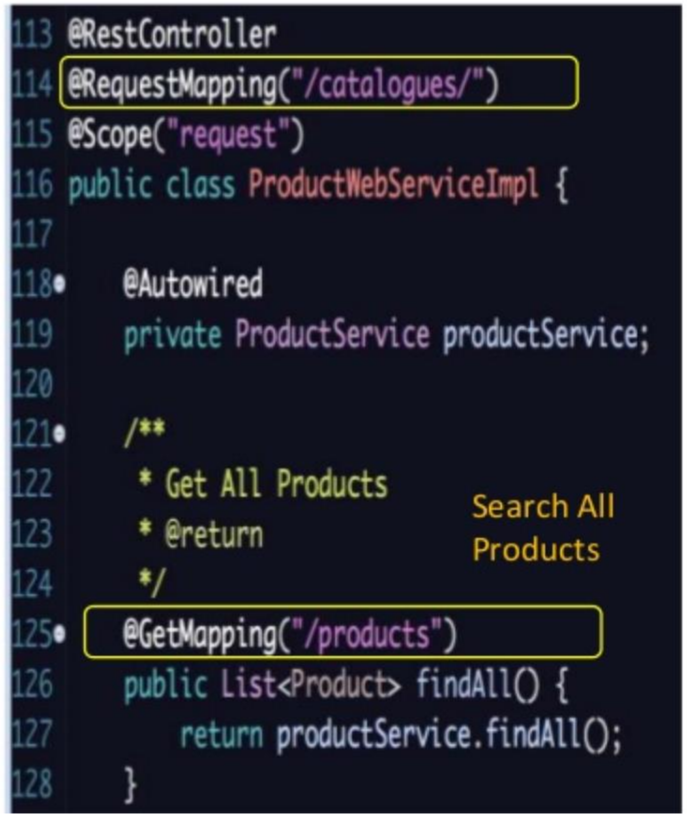
- **Recommendations**
- Use JSON as default
- Follow JavaScript conventions for naming attributes

 - Use medial capitalization (aka CamelCase) - Use uppercase or lowercase depending on type of object

- This results in code that looks like the following, allowing the JavaScript developer to write it in a way that makes sense for JavaScript.
- **"createdAt": 1320296464**
- **timing = myObject.createdAt;**

RESTful Guidelines – Query Examples

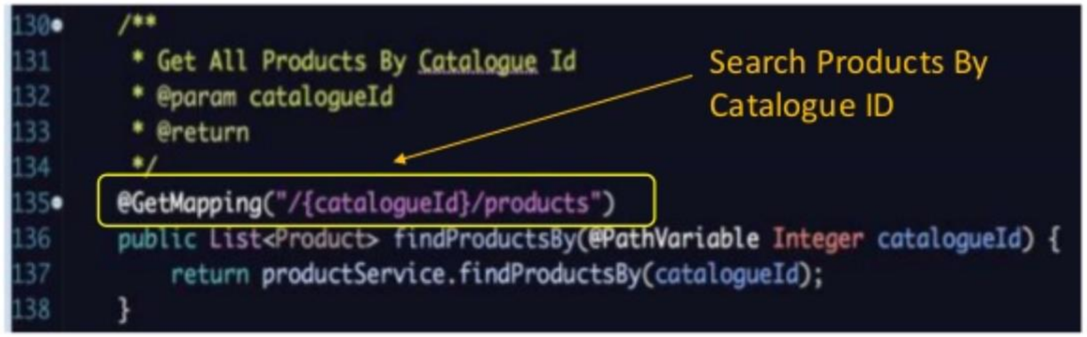
Clip slide



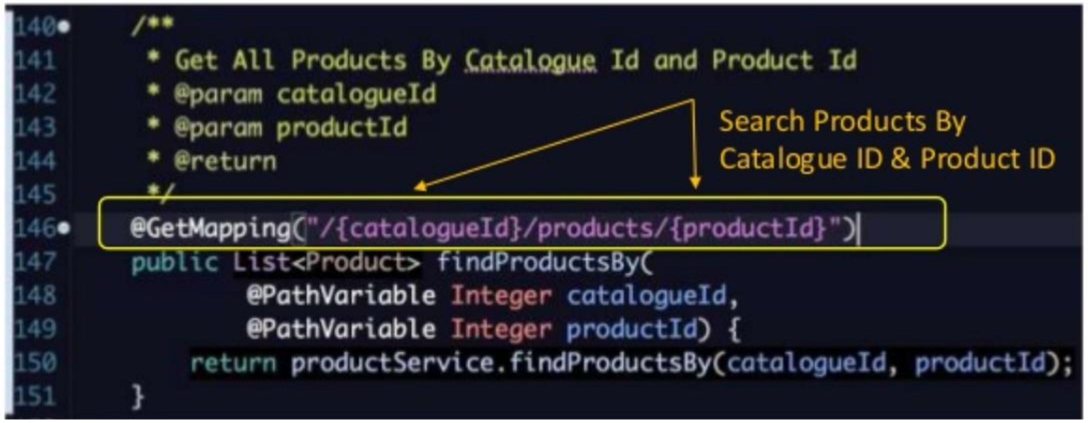
```
@RestController
@RequestMapping("/catalogues/")
@Scope("request")
public class ProductWebServiceImpl {

    @Autowired
    private ProductService productService;

    /**
     * Get All Products
     * @return
     */
    @GetMapping("/products")
    public List<Product> findAll() {
        return productService.findAll();
    }
```

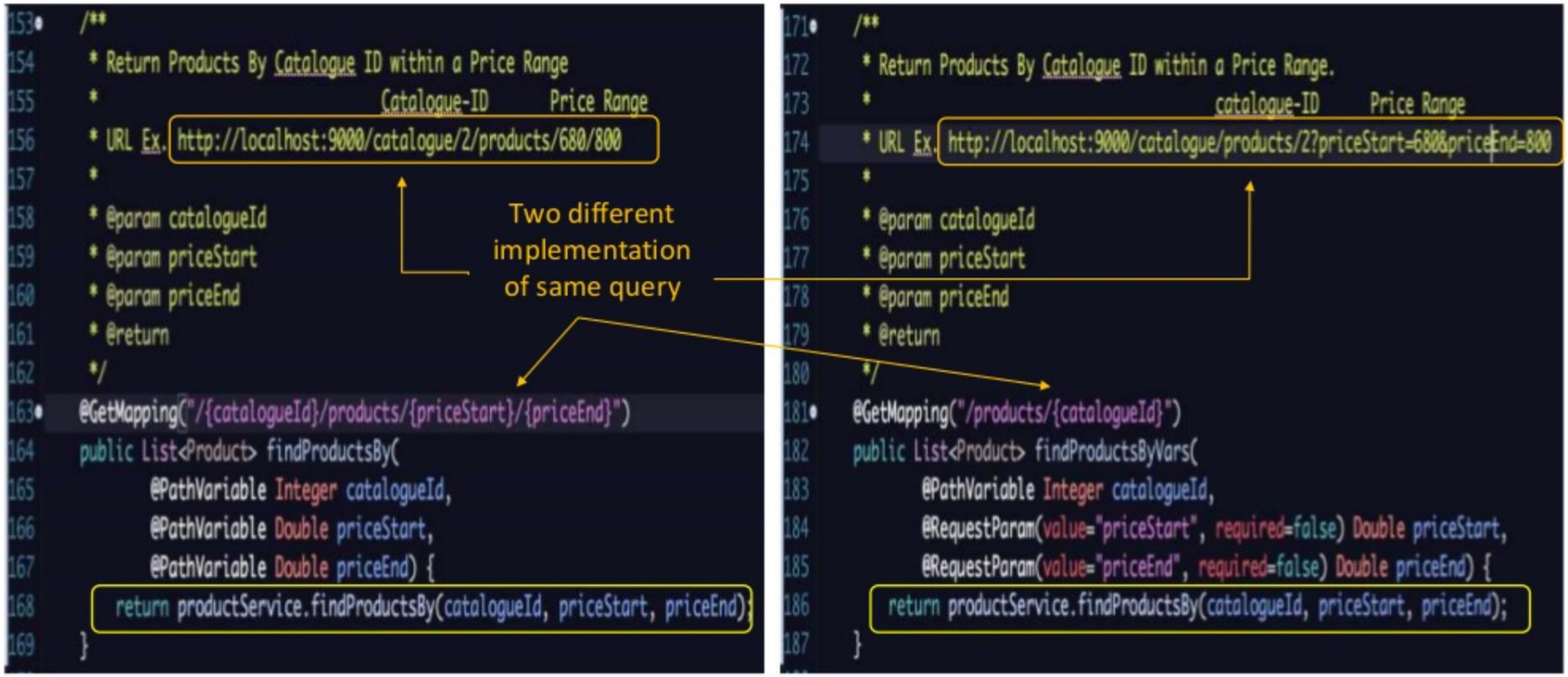


```
    /**
     * Get All Products By Catalogue Id
     * @param catalogueId
     * @return
     */
    @GetMapping("/{catalogueId}/products")
    public List<Product> findProductsBy(@PathVariable Integer catalogueId) {
        return productService.findProductsBy(catalogueId);
    }
```



```
    /**
     * Get All Products By Catalogue Id and Product Id
     * @param catalogueId
     * @param productId
     * @return
     */
    @GetMapping("/{catalogueId}/products/{productId}")
    public List<Product> findProductsBy(
            @PathVariable Integer catalogueId,
            @PathVariable Integer productId) {
        return productService.findProductsBy(catalogueId, productId);
    }
```

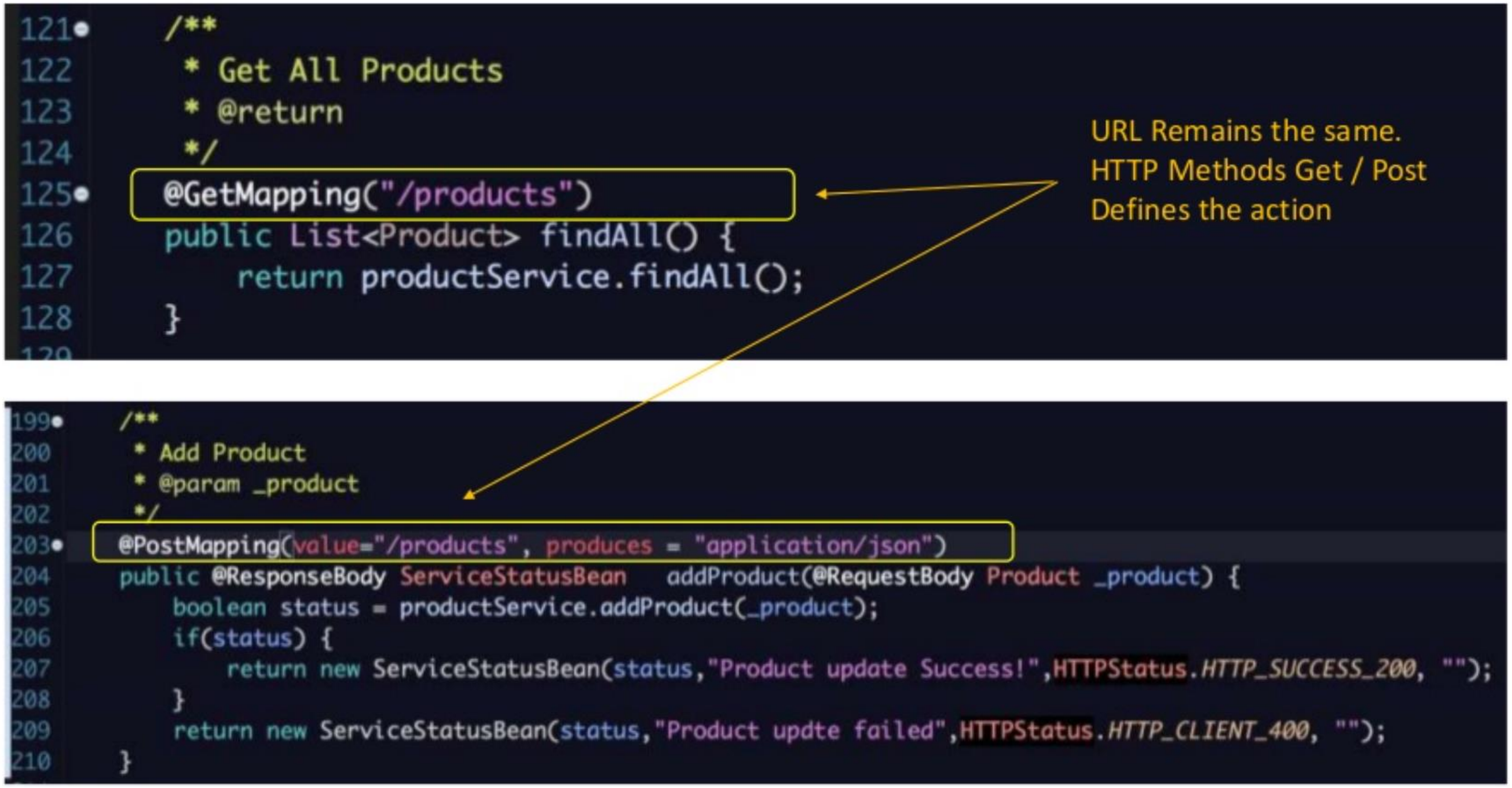
RESTful Guidelines – Query Examples



```
/**
 * Return Products By Catalogue ID within a Price Range
 *                                Catalogue-ID      Price Range
 * URL Ex. http://localhost:9000/catalogue/2/products/680/800
 *
 * @param catalogueId
 * @param priceStart
 * @param priceEnd
 * @return
 */
@GetMapping("/{catalogueId}/products/{priceStart}/{priceEnd}")
public List<Product> findProductsBy(
        @PathVariable Integer catalogueId,
        @PathVariable Double priceStart,
        @PathVariable Double priceEnd) {
    return productService.findProductsBy(catalogueId, priceStart, priceEnd);
}
```

```
/**
 * Return Products By Catalogue ID within a Price Range.
 *                                          catalogue-ID      Price Range
 * URL Ex. http://localhost:9000/catalogue/products/2?priceStart=680&priceEnd=800
 *
 * @param catalogueId
 * @param priceStart
 * @param priceEnd
 * @return
 */
@GetMapping("/products/{catalogueId}")
public List<Product> findProductsByVars(
        @PathVariable Integer catalogueId,
        @RequestParam(value="priceStart", required=false) Double priceStart,
        @RequestParam(value="priceEnd", required=false) Double priceEnd) {
    return productService.findProductsBy(catalogueId, priceStart, priceEnd);
}
```

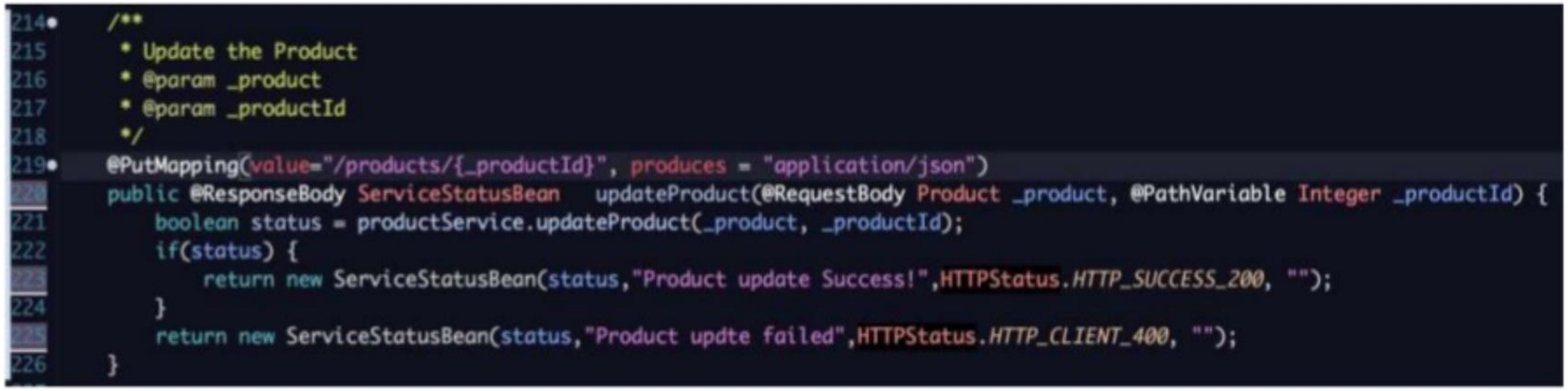
RESTful Guidelines – Get & Create Example



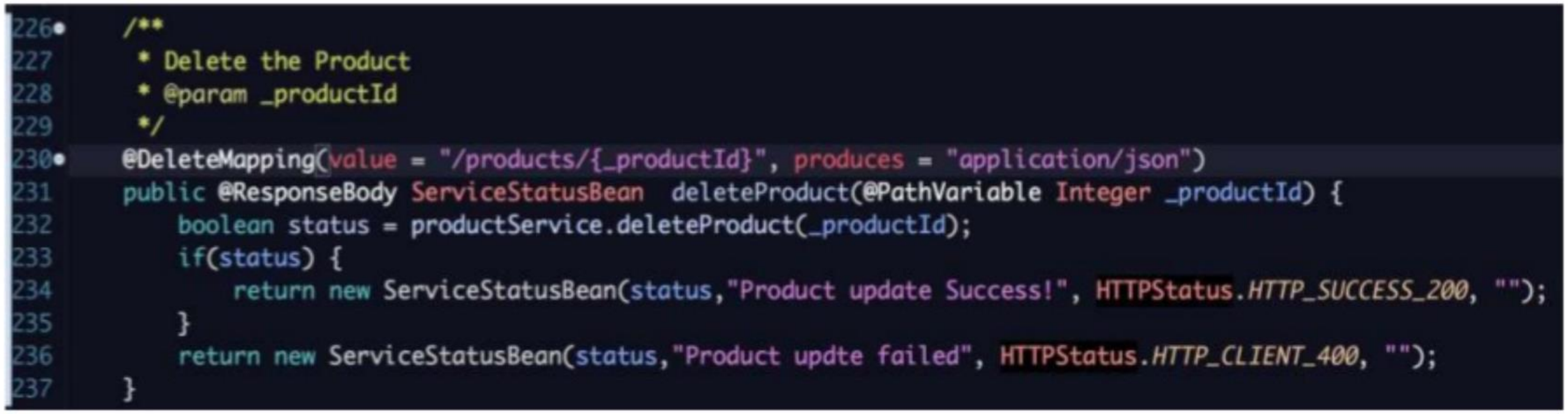
```
    /**
     * Get All Products
     * @return
     */
    @GetMapping("/products")
    public List<Product> findAll() {
        return productService.findAll();
    }
```

```
    /**
     * Add Product
     * @param _product
     */
    @PostMapping(value="/products", produces = "application/json")
    public @ResponseBody ServiceStatusBean  addProduct(@RequestBody Product _product) {
        boolean status = productService.addProduct(_product);
        if(status) {
            return new ServiceStatusBean(status,"Product update Success!",HTTPStatus.HTTP_SUCCESS_200, "");
        }
        return new ServiceStatusBean(status,"Product updte failed",HTTPStatus.HTTP_CLIENT_400, "");
    }
```

RESTful Guidelines – Update & Delete Example



```
/**
 * Update the Product
 * @param _product
 * @param _productId
 */
@PutMapping(value="/products/{_productId}", produces = "application/json")
public @ResponseBody ServiceStatusBean  updateProduct(@RequestBody Product _product, @PathVariable Integer _productId) {
    boolean status = productService.updateProduct(_product, _productId);
    if(status) {
        return new ServiceStatusBean(status,"Product update Success!",HTTPStatus.HTTP_SUCCESS_200, "");
    }
    return new ServiceStatusBean(status,"Product updte failed",HTTPStatus.HTTP_CLIENT_400, "");
}
```



```
/**
 * Delete the Product
 * @param _productId
 */
@DeleteMapping(value = "/products/{_productId}", produces = "application/json")
public @ResponseBody ServiceStatusBean  deleteProduct(@PathVariable Integer _productId) {
    boolean status = productService.deleteProduct(_productId);
    if(status) {
        return new ServiceStatusBean(status,"Product update Success!", HTTPStatus.HTTP_SUCCESS_200, "");
    }
    return new ServiceStatusBean(status,"Product updte failed", HTTPStatus.HTTP_CLIENT_400, "");
}
```

Security

- There are different types of authentication in web API.
- Token-based authentication, where the token is a random hash assigned to the user and they can reset it at any point if it has been stolen.
- Allow the token to be passed in through POST or an HTTP header.
- OAuth 2:
  - OAuth is a protocol which provides implementations of both OAuth 1 and OAuth 2 and it's used to secure a java Web API.

Security

- OAuth flow starts where the registered user will present username and password to a specific end point.
- API will validate those credentials, and if everything is valid, it will return a token for the user where the client application used by the user should store it securely and locally in order to present this token with each request to any protected end point.

Security

- Token Generation:
- Choose a secure token, not a short numeric identifier or random string. Something irreversible is best.
- Generate token by using JWT. JWT is lightweight and it has pay load. You can choose algorithm type.
- Generate token by using SHA1 algorithm.
- Finally, if user is authenticated they can access the methods, if not it returns 401 unauthorized http code

Authentication

- Let’s take three top services.
- How each of these services handles things differently:
  - PayPal Permissions Service API
  - Facebook OAuth 2.0
  - Twitter OAuth 1.0a
- Note that PayPal's proprietary three-legged permissions API was in place long before OAuth was conceived

API Validation

- Invalid response codes
- Invalid response headers
- API time-outs
- Slow API response with respect to response data bytes
- Incorrect required data in JSON responses
- Missing required text in response bodies
- Slow response for customers in specific locations

Chatty API

- Some API designs become very chatty - meaning just to build a simple UI or app, you have dozens or hundreds of API calls back to the server.
- The API team can sometimes decide not to deal with creating a nice, resource-oriented RESTful API, and just fall back to a mode where they create the 3 or 4 Java-style getter and setter methods they know they need to power a particular user interface.

Chatty API

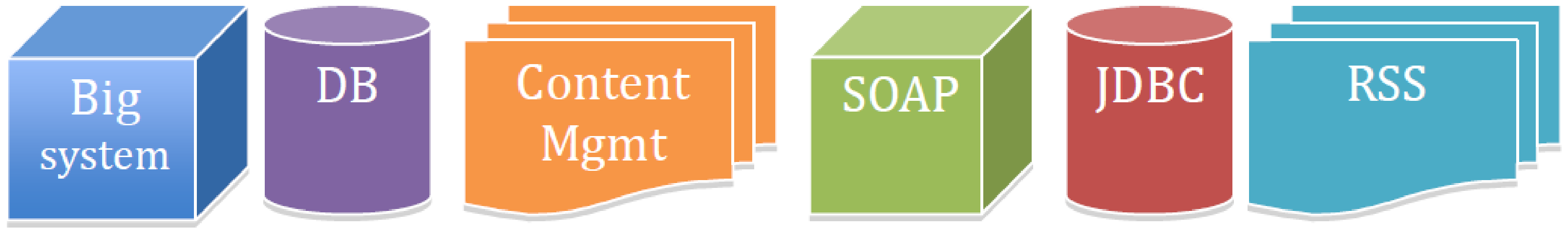
- **Be complete and RESTful and provide shortcuts**

- What kind of shortcut? Say you know that 80% of all your apps are going to need some sort of composite response, then build the kind of request that gives them what they need.

Chatty API

- **Take advantage of the partial response syntax**
- What kind of shortcut? Say you know that 80% of all your apps are going to need some sort of composite response, then build the kind of request that gives them what they need.

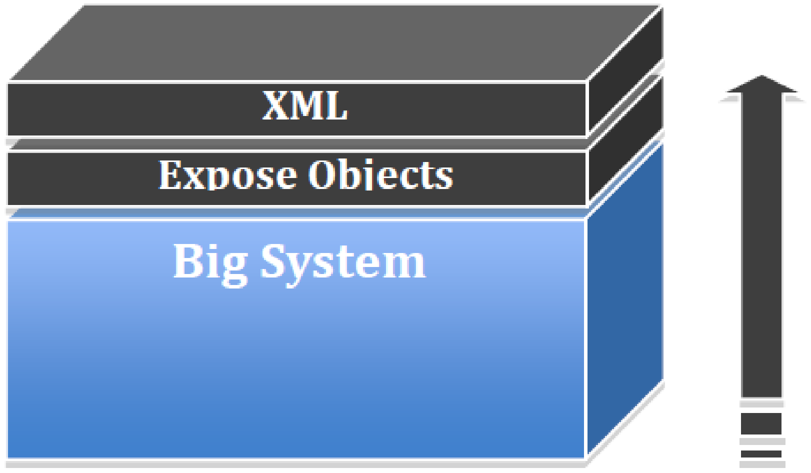
The API Façade Pattern



- Back-end systems of record are often too complex to expose directly to application developers.
- They are stable (have been hardened over time) and dependable (they are running key aspects of you business), but they are often based on legacy technologies and not always easy to expose to Web standards like HTTP.
- These systems can also have complex interdependencies and they change slowly meaning that they can’t move as quickly as the needs of mobile app developers and keep up with changing formats.

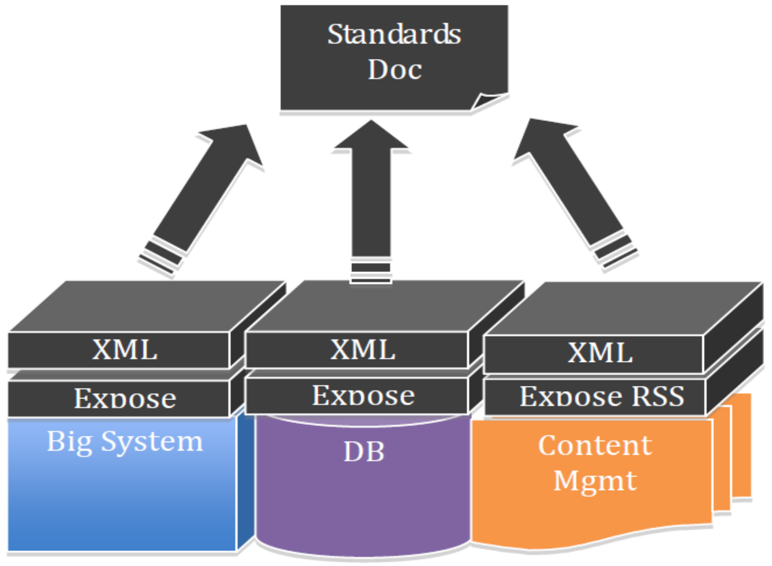
The API Façade Pattern

- In the build-up approach, a developer exposes the core objects of a big system and puts an XML parsing layer on top.



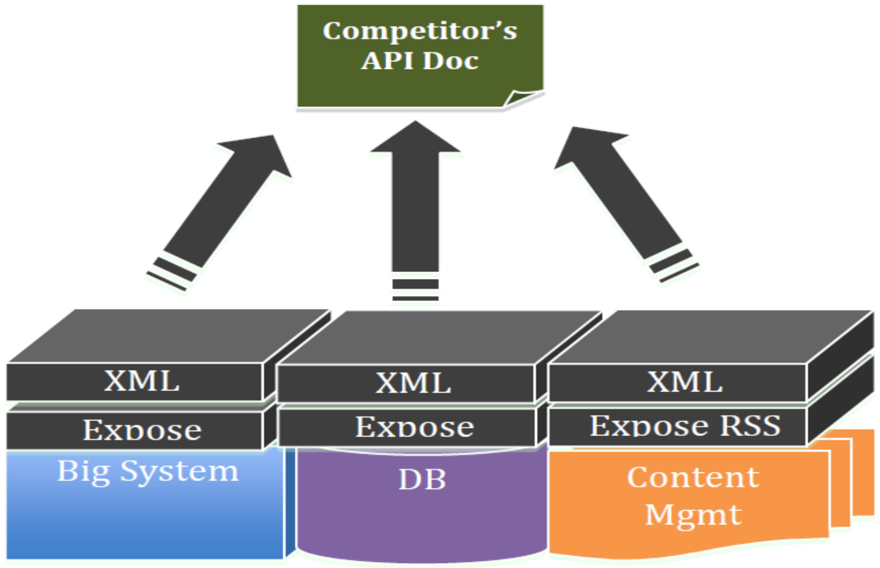
The API Façade Pattern

- **The Standards Committee Approach**
  - Often the internal systems are owned and managed by different people and departments with different views about how things should work.
  - Designing an API by a standards committee often involves creating a standards document, which defines the schema and URLs and such.
  - All the stakeholders build toward that common goal.

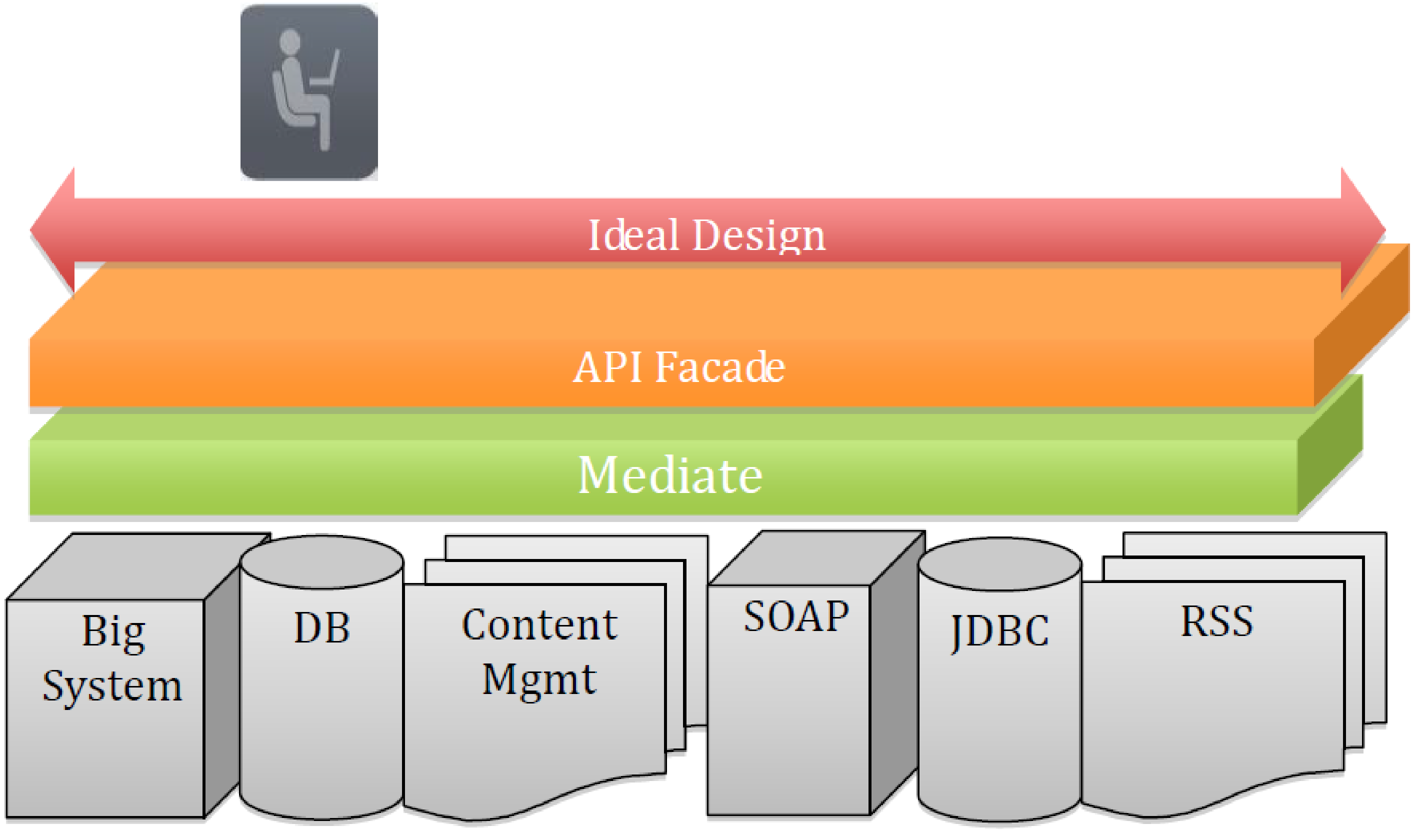


The API Façade Pattern

- **The Copy Cat Approach**
  - We sometimes see this pattern when an organization is late to market – for example, when their close competitor has already delivered a solution.
  - Again, this approach can get you to version 1 quickly and you may have a built-in adoption curve if the app developers who will use your API are already familiar with your competitor's API.



Solution – The API façade pattern



Solution – The API façade pattern

- Implementing an API façade pattern involves three basic steps.
- 1 - Design the ideal API – design the URLs, request parameters and responses, payloads, headers, query parameters, and so on. The API design should be self-consistent.
- 2 - Implement the design with data stubs. This allows application developers to use your API and give you feedback even before your API is connected to internal systems.
- 3 - Mediate or integrate between the façade and the systems.

HTTP and REST: A Data-oriented Design Paradigm

- A REST API focuses on the underlying entities of the problem domain it exposes, rather than a set of functions that manipulate those entities.
- suppose our problem domain is tracking customers and their accounts. Primary entities we might expose would include:
- The collection of known customers. Its URL might be https://customertracker.com/customers

HTTP and REST: A Data-oriented Design Paradigm

- A REST API focuses on the underlying entities of the problem domain it exposes, rather than a set of functions that manipulate those entities.
- suppose our problem domain is tracking customers and their accounts. Primary entities we might expose would include:
- The collection of known customers. Its URL might be https://customertracker.com/customers
- Individual customers. They each have a unique URL.

API Design Elements

- The following aspects of API design are all important, and together they define your API:
- The representations of your resources—this includes the definition of the fields in the resources (assuming your resource representations are structured, rather than streams of bytes or characters), and the links to related resources.
- The use of standard (and occasionally custom) HTTP headers.
- The URLs and URI templates that define the query interface of your API for locating resources based on their data.
- Required behaviors by clients—for example, DNS caching behaviors, retry behaviors, tolerance of fields that were not previously present in resources, and so on.

Designing Representations

- Representation is the technical term for the data that is returned when a web resource is retrieved by a client from a server, or sent from a client to a server.
- In the REST model, a web resource has underlying state, which cannot be seen directly, and what flows between clients and servers is a representation of that state.
- Users can view the representation of a resource in different formats, called media types.
- In principle, all media types for the representation of a particular resource should encode the same information, just in different formats.

JSON

- The dominant media type for resource representations in web APIs is JavaScript Object Notation (JSON).
- The primary reasons for JSON’s success are probably that it is simple to understand, and it is easy to map to the programming data structures of JavaScript and other popular programming languages (Python, Ruby, Java, and so on).
- It is now the de facto standard for web APIs, and you should use it.

JSON

- While JSON is very good and very popular, it is not perfect for our purposes.
- One limitation is that JSON can only represent a small number of data types (Null, Boolean, Number, String).
- The most common types we have come across in web API design that are not supported by JSON are dates and times and URLs.
- There are several options for dealing with this limitation—the simplest is to represent them as strings, and rely on the context to determine which strings are just strings and which are really stringified dates or URLs.

Keep your JSON simple

- When JSON is used well, it is simple, intuitive, and largely self-explanatory.
- If your JSON doesn’t look as straightforward as the example below, you may be doing something wrong.
- {
- “kind”: “Dog”
- “name”: “Lassie”,
- “furColor”: “brown”,
- ...
- }

Questions

Module Summary

– Spring Integration Framework.
– Message, Channel and Adapter
– Understood the different Component Integration
– Understood the Event-Driven Architecture