Pro RESTful APIs

Design, Build and Integrate with REST, JSON, XML and JAX-RS

Sanjay Patni

Apress®

Pro RESTful APIs

Design, Build and Integrate with REST, JSON, XML and JAX-RS

Sanjay Patni

Apress®

Pro RESTful APIs: Design, Build and Integrate with REST, JSON, XML and JAX-RS

Sanjay Patni
Santa Clara, California
USA

ISBN-13 (pbk): 978-1-4842-2664-3 ISBN-13 (electronic): 978-1-4842-2665-0
DOI 10.1007/978-1-4842-2665-0

Library of Congress Control Number: 2017936942

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Technical Reviewer: Massimo Nardone
Coordinating Editor: Mark Powers
Copy Editor: Larissa Shmailo
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global
Cover image designed by Freepik

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484226643. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

I dedicate this book to my family and friends for their support.
A special feeling of gratitude to Alakh Verma,
Director Platform Products, Oracle and Andy Hou, Director Technology,
UCSC Extension for constant encouragement.

Contents at a Glance

Contents

About the Author

Sanjay Patni is a results-focused technologist with extensive experience in aligning innovative technology solutions with business needs to optimize manual steps in the business processes and improving operational efficiency.

For the last five years at Oracle he has worked with the Fusion Apps Product development team, where he has identified opportunities for automation of programs related to FusionApps code lines management. This involved delivery of GA Releases for patching, as well as codelines for ongoing demo, development, and testing. He conceptualized and developed Self Service UX for codeline requests and auditing, reducing manual steps by 80%. He also rolled out 12 sprints of code line creation, automating about 100+ manual steps involving integration with other subsystems using technologies like Automation workflow and RESTful APIs.

Prior to joining Oracle, he spent 15+ years in the software industry, defining and delivering on key initiatives across different industry sectors. His responsibilities included innovation, requirement, analysis, technical architecture, design, and agile software development of Web-based enterprise products and solutions. He pioneered innovative usage of Java in building business applications and received an award from Sun Microsystems. Feedback improved for Java APIs for Enterprise in building business application software using Java.

He has worked as a visiting technical instructor or mentor and conducted classes or training on RESTful APIs design and integration.

He has a strong educational background in computer science with masters from IIT, Roorkee, India.

About the Technical Reviewer

Massimo Nardone has more than 22 years of experiences in security, Web/mobile development, Cloud and IT architecture. His true IT passions are security and Android.

He has been programming and teaching how to program with Android, Perl, PHP, Java, VB, Python, C/C++, and MySQL for more than 20 years.

He holds a master of science degree in computing science from the University of Salerno, Italy.

He has worked as a project manager, software engineer, research engineer, chief security architect, information security manager, PCI/SCADA auditor and senior lead IT security/Cloud/SCADA architect for many years.

His technical skills include security, Android, Cloud, Java, MySQL, Drupal, Cobol, Perl, Web and mobile development, MongoDB, D3, Joomla, Couchbase, C/C++, WebGL, Python, Pro Rails, Django CMS, Jekyll, Scratch, etc.

He currently works as Chief Information Security Office (CISO) for Cargotec Oyj.

He worked as visiting lecturer and supervisor for exercises at the Networking Laboratory of the Helsinki University of Technology (Aalto University). He holds four international patents (PKI, SIP, SAML, and Proxy areas).

Massimo has reviewed more than 40 IT books for different publishing companies and he is the coauthor of *Pro Android Games* (Apress, 2015).

Introduction

Databases, web sites, and business applications need to exchange data. This is accomplished by defining standard data formats such as Extensible Markup Language (XML) or JavaScript Object Notation (JSON), as well as transfer protocols or Web services such as the Simple Object Access Protocol (SOAP) or the more popular Representational State Transfer (REST). Developers often have to design their own Application Programming Interfaces (APIs) to make applications work while integrating specific business logic around operating systems, or servers. This book introduces these concepts with a focus on the RESTful APIs.

This book introduces the data exchange mechanism and common data formats. For Web exchange, you will learn the HTTP protocol, including how to use XML. This book compares SOAP and REST, and then covers the concepts of stateless transfer. It introduces software API design and best design practices. The second half of the book focuses on RESTful API design and implementations that follow the JAX-RS standard, and Java API for RESTful Web Services. You will learn how to build and consume JAX-RS services using JSON and XML, and integrate RESTful API with different data sources like relational databases and NoSQL databases through hands-on exercises. You will apply these best practices to complete a design review of publicly available APIs with a small-scale software system in order to design and implement RESTful API.

This book is intended for software developers who use data in projects. It is also useful for data professionals who need to understand the methods of data exchange and how to interact with business applications. Java programming experience is required for the exercises.

Topics include:

- Data exchange and Web services
- SOAP vs. REST, state vs. stateless
- XML vs. JSON
- Introduction to API design: REST and JAX-RS
- API design practices
- Designing RESTful API
- Building RESTful API
- Interacting with RDBMS (MySQL)
- Consuming RESTful API (i.e., JSON, XML)
- API Security-OAuth
- API Caching

CHAPTER 1

Fundamentals of RESTful APIs

APIs are not new. They've served as interfaces that enable applications to communicate with each other for decades. But the role of APIs has changed dramatically in the last few years. Innovative companies have discovered that APIs can be used as an interface to the business, allowing them to monetize digital assets, extend their value proposition with partner-delivered capabilities, and connect to customers across channels and devices. When you create an API, you are allowing others within or outside of your organization to make use of your service or product to create new applications, attract customers, or expand their business. Internal APIs enhance the productivity of development teams by maximizing reusability and enforcing consistency in new applications. Public APIs can add value to your business by allowing third party developers to enhance your services or bring their customers to you. As developers find new applications for your services and data, a network effect occurs, delivering significant bottom-line business impact. For example, Expedia opened up their travel booking services to partners through an API to launch the Expedia Affiliate Network, building a new revenue stream that now contributes $2B in annual revenue. Salesforce released APIs to enable partners to extend the capabilities of their platform and now generates half of their annual revenue through those APIs, which could be SOAP-based (JAX-WS) and, more recently, RESTful (JAX-RS).

SOAP web service depends upon a number of technologies (such as UDDI, WSDL, SOAP, HTTP) and protocols to transport and transform data between a service provider and the consumer, and can be created with JAX-WS.

Later, Roy Fielding (in the year 2000) presented his doctoral dissertation, "Architectural Styles and the Design of Network-based Software Architecture." He coined the term "REST," an architectural style for distributed hypermedia systems. Put simply, REST (short for REpresentational State Transfer) is an architectural style defined to help create and organize distributed systems. The key word from that definition should be "style," because an important aspect of REST (and which is one of the main reasons books like this one exist) is that it is an architectural style—not a guideline, not a standard, or anything that would imply that there are a set of hard rules to follow in order to end up having a RESTful architecture.

This chapter has details about REST fundamentals, SOAP vs. REST, and Web Architectural Style.

S. Patni, *Pro RESTful APIs*, DOI 10.1007/978-1-4842-2665-0_1

The main idea behind REST is that a distributed system, organized RESTfully, will improve in the following areas:

- **Performance**: The communication style proposed by REST is meant to be efficient and simple, allowing a performance boost on systems that adopt it.
- **Scalability of component interaction**: Any distributed system should be able to handle this aspect well enough, and the simple interaction proposed by REST greatly allows for this.
- **Simplicity of interface**: A simple interface allows for simpler interactions between systems, which in turn can grant benefits like the ones previously mentioned.
- **Modifiability of components**: The distributed nature of the system, and the separation of concerns proposed by REST (more on this in a bit), allows for components to be modified independently of each other at a minimum cost and risk.
- **Portability**: REST is technology- and language-agnostic, meaning that it can be implemented and consumed by any type of technology (there are some constraints that I'll go over in a bit, but no specific technology is enforced).
- **Reliability**: The stateless constraint proposed by REST (more on this later) allows for the easier recovery of a system after failure.
- **Visibility**: Again, the stateless constraint proposed has the added full state of said request (this will become clear once I talk about the constraints in a bit). From this list, some direct benefits can be extrapolated. A component-centric design allows you to make systems that are very fault-tolerant. Having the failure of one component not affect the entire stability of the system is a great benefit for any system. Interconnecting components is quite easy, minimizing the risks when adding new features or scaling up or down. A system designed with REST in mind will be accessible to a wider audience, thanks to its portability (as described earlier). With a generic interface, the system can be used by a wider range of developers. In order to achieve these properties and benefits, a set of constraints were added to REST to help define a uniform connector interface. REST is not suggested to use when you need to enforce a strict contract between client and server and when performing transactions that involve multiple calls.

SOAP vs. REST

Table 1-1 has a comparsion between SOAP and REST with an example of use cases each can support.

Table 1-1. *SOAP vs. REST comparision*

Topic	SOAP	REST
Origin	SOAP (Simple Object Access Protocol) was created in 1998 by Dave Winer et al. in collaboration with Microsoft. Developed by a large software company, this protocol addresses the goal of addressing the needs of the enterprise market.	REST (Representational State Transfer) was created in 2000 by Roy Fielding at UC, Irvine. Developed in an academic environment, this protocol embraces the philosophy of the open Web.
Basic Concept	Makes data available as services (verb + noun), for example “getUser” or “PayInvoice”	Makes data available as resources (nouns), for example “user” or “invoice”
Pros	Follows a formal enterprise approach Works on top of any communication protocol, even asynchronously Information about objects is communicated to clients. Security and authorization are part of the protocol. Can be fully described using WSDL	Follows the philosophy of the Open Web Relatively easy to implement and maintain Clearly separates client and server implementations Communication isn’t controlled by a single entity Information can be stored by the client to prevent multiple calls. Can return data in multiple formats (JSON, XML etc.)
Cons	Spends a lot of bandwidth communicating metadata Hard to implement and is unpopular among Web and mobile developers	Only works on top of the HTTP protocol Hard to enforce authorization and security on top of it
When to use	When clients need to have access to objects available on servers When you want to enforce a formal contract between client and server	When clients and servers operate on a Web environment When information about objects doesn’t need to be communicated to the client
When not to use	When you want the majority of developers to easily use your API When your bandwidth is very limited	When you need to enforce a strict contract between client and server When performing transactions that involve multiple calls

(*continued*)

Table 1-1. (*continued*)

Topic	SOAP	REST
Use cases	Financial services Payment gateways Telecommunication services	Social media services Social networks Web chat services Mobile services
Examples	https://www.salesforce.com/developer/docs/api/ - Salesforce SOAP API https://developer.paypal.com/docs/classic/api/PayPalSOAPAPIArchitecture/ -Paypal SOAP API	https://dev.twitter.com/ https://developer.linkedin.com/apis
Conclusion	Use SOAP if you are dealing with transactional operations and you already have an audience that is satisfied with this technology.	Use REST if you're focused on wide-scale API adoption or if your API is targeted at mobile apps.

Web Architectural Style

According to Fielding, there are two ways to define a system.

- One is to start from a blank slate—an empty whiteboard—with no initial knowledge of the system being built or the use of familiar components until the needs are satisfied.
- A second approach is to start with the full set of needs for the system, and constraints are added to individual components until the forces that influence the system are able to interact in harmony with each other.

REST follows the second approach. In order to define a REST architecture, a null-state is initially defined—a system that has no constraints whatsoever and where component differentiation is nothing but a myth—and constraints are added one by one. The following subsections cover web architectural style constraints. Each of these constrations defines how the framework for REST APIs should be architected and designed. Security is another aspect which needs to be considered independently as part of this framework when rolling out RESTful APIs to the end users.

Client-Server

The separation of concerns is the core theme of the Web's client-server constraints.

The Web is a client-server-based system, in which clients and servers have distinct parts to play.

They may be implemented and deployed independently, using any language or technology, so long as they conform to the Web's uniform interface.

Uniform Resource Interface

The interactions between the Web's components—meaning its clients, servers, and network-based intermediaries—depend on the uniformity of their interfaces.

Web components interoperate consistently within the uniform interface's four constraints, which Fielding identified as:

- Identification of resources
- Manipulation of resources through representations
- Self-descriptive messages
- Hypermedia as the engine of application state (HATEOAS)

Layered System

Generally speaking, a network-based intermediary will intercept client-server communication for a specific purpose.

Network-based intermediaries are commonly used for enforcement of security, response caching, and load balancing

The layered system constraints enable network-based intermediaries such as proxies and gateways to be transparently deployed between a client and server using the Web's uniform interface.

Caching

Caching is one of web architecture's most important constraints. The cache constraints instruct a web server to declare the cache ability of each response's data.

Caching response data can help to reduce client-perceived latency, increase the overall availability and reliability of an application, and control a web server's load. In a word, caching reduces the overall cost of the Web.

Stateless

The stateless constraint dictates that a web server is not required to memorize the state of its client applications. As a result, each client must include all of the contextual information that it considers relevant in each interaction with the web server.

Web servers ask clients to manage the complexity of communicating their application state so that the web server can service a much larger number of clients. This trade-off is a key contributor to the scalability of the Web's architectural style.

Code-on-Demand

The Web makes heavy use of code-on-demand, a constraint which enables web servers to temporarily transfer executable programs, such as scripts or plug-ins, to clients.

Code-on-demand tends to establish a technology coupling between web servers and their clients, since the client must be able to understand and execute the code that it downloads on-demand from the server. For this reason, code-on-demand is the only constraint of the Web's architectural style that is considered optional.

HATEOAS

The final principle of REST is the idea of using Hypermedia As The Engine Of Application State (HATEOAS). When developing a client-server solution using HATEOAS, the logic on the server side might change independently of the clients.

Hypermedia is a document-centric approach with the added support for embedding links to other services and information within the document format.

One of the uses of hypermedia and hyperlinks is composing complex sets of information from disparate sources. The information could be within a company private cloud or within a public cloud from disparate sources.

Example:

```
<podcast id="111">
   <customer>http://customers.myintranet.com/customers/1</customers>
   <link>http://podcast.com/myfirstpodcast</link>
   <description> This is my first podcast </description>
</podcast>
```

Each of these web architecture styles adds beneficial properties to the web system.

By adopting these constraints, teams can build simple, visible, usable, accessible, evolvable, flexible, maintainable, reliable, scalable and performant systems as shown in Table 1-2 below:

Table 1-2. *Constraint and system property*

By following the constraint	Gain the following system property
Client-server interactions	Simple, Evolvable, Scalable
Stateless communications	Simple, Visible, Maintainable, Evolvable, and Reliable
Cacheable data	Visible, Scalable, and Performant
Uniform Interfaces	Simple, Usable, Visible, Accessible, Evolvable, and Reliable
Layered system	Flexible, Scalable, Reliable, and Per formant
Code on demand	Evolvable

Security

We have not covered security in this chapter as part of REST fundamentals, but security is very important for rolling out RESTful APIs. This book has a complete chapter on securing RESTful APIs which has details on best practices for securing RESTful APIs and OAuth, which is a standard for REST APIs security.

What is REST?

We have briefly introduced REST with REST API fundamentals in the previous section. This section has further introductory details about REST concepts.

"REST" was coined by Roy Fielding in his Ph.D. dissertation to describe a design pattern for implementing networked systems. REST is Representational State Transfer, an architectural style for designing distributed systems. It's not a standard, but rather a set of constraints. It's not tied to HTTP, but is associated most commonly with it.

REST Basics

Unlike SOAP and XML-RPC, REST does not really require a new message format. The HTTP API is CRUD (Create, Retrieve, Update, and Delete)

- GET = "give me some info" (Retrieve)
- POST = "here's some update info" (Update)
- PUT = "here's some new info" (Create)
- DELETE = "delete some info" (Delete)
- And more….
- PATCH = The HTTP method PATCH can be used to update partial resources. For instance, when you only need to update one field of the resource, PUTting a complete resource representation might be cumbersome and utilizes more bandwidth.
- HEAD = The **HEAD** method is identical to the GET method, except that the server must not return a message body in the response. This method is often used for testing hypertext links for validity, accessibility, and recent modification.
- OPTIONS = This method allows the client to determine the options and/or requirements associated with a resource, or the capabilities of a server, without implying a resource action or initiating a resource retrieval.
- Notion of "Idempotency" – the idea that when sending a GET, DELETE, or PUT to the system, the effect should be the same whether the command is sent one or more times, but POST creates an entity in the collection and therefore is not idempotent.

REST Fundamentals

Just to remind you, about 8,356 APIs were written in REST by ProgrammableWeb.com in 2016. REST is resource-based architecture. A resource is accessed via a common interface based on the HTTP standard methods. REST asks developers to use HTTP methods explicitly and in a way that's consistent with the protocol definition. Each resource is identified by a URL. Every resource should support the HTTP common operations, and REST allows that resource to have different representations, e.g., text, xml, json, etc. The rest client can ask for specific representation via the HTTP protocol (Content Negotiation). Table 1-3 below describes data elements used in REST.

Table 1-3. *Structures of REST*

Data Element	Description
Resource	Conceptual target of a hypertext reference, e.g., customer/order
Resource Identifier	A uniform resource locator (URL) or uniform resource name (URN) identifying a specific resource, e.g., `http://myrest.com/customer/3435`
Resource Metadata	Information describing the resource, e.g., tag, author, source link, alternate location, alias names
Representation	The resource content—JSON Message, HTML Document, JPEG Image
Representation Metadata	Information describing how to process the representation, e.g., media type, last-modified time
Control Data	Information describing how to optimize response processing, e.g., if-modified-since, cache-control-expiry

Let's look at some examples.

Resources

First, a REST resource to GET a list of podcasts:

```
http://prorest/podcasts
```

Next, a REST resource to GET details of podcast id 1:

```
http://prorest/podcasts/1
```

Representations

Here is an XML respresentation of a response—GET customer for an id.

```
<Customer>
   <id>123</id>
   <name>John</name>
</Customer>
```

Next, a JSON respresentation of a response—GET customer for an id:

```
{"Customer":{"id":"123","name":"John"}}
```

Content Negotiation

HTTP natively supports a mechanism based on headers to tell the server about the content you expect and you're able to handle. Based on these hints, the server is responsible for returning the corresponding content in the correct format. Figure 1-1 shows an example.

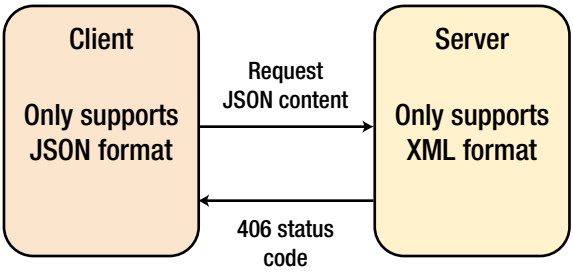


Figure 1-1. *Content negotiation*

If the server doesn't support the requested format, it will send back a 406 status code (Not Acceptable) to notify the client that made the request ("The requested resource is only capable of generating content not acceptable according to the Accept headers sent in the request") according to the specification.

Wrapping-Up

REST identifies the key architectural principles of why the Web is prevalent and scalable. The next step in the education of the Web is to apply these principles to the semantics Web and the world of web services. REST offers a simple, interoperable, and flexible way of writing web services that can be very different than the WS-* that so many of you had training in. In the next chapter we will cover these concepts in more detail.

CHAPTER 2

API Design and Modeling

This chapter starts with API design strategies and then goes into API creation process and modeling. Best practices for REST API design are discussed, followed by API solution architecture. In the excercises, a simple API is designed for podcasts suscription and then modeling using RAML.

API Design Strategies

As UI is to UX (User Experience), API is to APX (Application Programming Experience). In APX it is important o answer following questions:

- What should be exposed?
- What is the best way to expose the data?
- How should API be adjusted and improved?

In addition, let's discuss why we should develop a nice Application Programming Experience?

A nice API will encourage the developers to use it and share it with others, creating a virtuous cycle where each additional successful implementation leads to more engagement and more contributions from developers who add value to your service. I'll start by saying that API design is hard.

Also, a nice API will help to grow an ecosystem of employees, customers, and partners who can use and help to continue to evolve your API in ways that are mutually beneficial.

There are four strategies for API design:

- Bolt-on strategy: This is when you have an existing application and add an API after the fact. This takes advantage of existing code and systems (Figure 2-1).

S. Patni, *Pro RESTful APIs*, DOI 10.1007/978-1-4842-2665-0_2

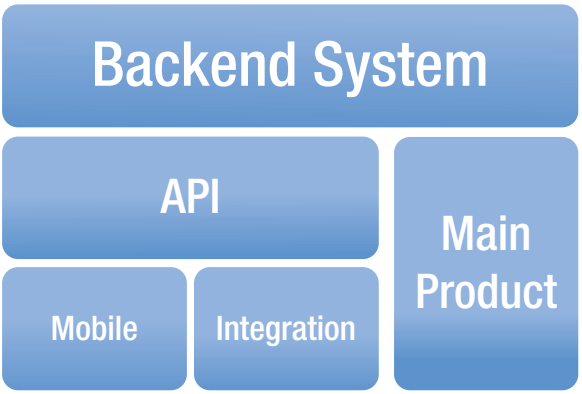


Figure 2-1. *Bolt-on strategy*

- Greenfield strategy: This is the other extreme. This is a strategy behind "API-first" or "Mobile first," and is the easiest scenario to develop an API. Since you're starting from scratch, you can make use of technologies and concepts that may not have been available before (Figure 2-2).

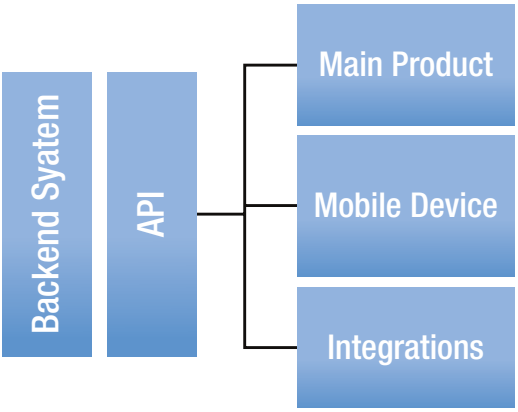


Figure 2-2. *Greenfield strategy*

Greanfield or API-first strategy is a simulation-based design implementation.

Simulation of a back-end system is development of a back-end system without needing fully implemented back-end systems. With simulation of APIs, consumers can start development of apps without fully developed APIs.

- Agile design strategy: Agility is based on the premise that you can start without a full set of specs. You can always adapt and change the specs later, as you go and learn more. Through multiple iterations, architectural design can converge to the right solution. Agile approach should only be applied until API is published.
- Finally, you have the facade strategy, which is the middle ground between Greenfield and bolt-on. In this case, you can take advantage of existing business systems, yet shape them to what you prefer and need. This gives them the ability to keep working systems in place while making the underlying architecture better.

API Creation Process and Methodology

In this section we are going to review API creation process and methodology. In order to deliver great APIs, the design must be a first-order concern. Like optimizing for UX (User Experience) has become a primary concern in UI development, also optimizing for APX (API User Experience) should be a primary concern in API development.

Process

First determine your business value. When thinking about business value, think of the "elevator pitch" about why you need an API. Developer engagement is not a great goal; you need a tangible goal: increase user engagement, move activity off the main product to the API, engage and retain partners, and so on.

Choose your metrics, e.g.:

- Number of developer keys in use
- Number of applications developed
- Number of users interacting via API
- Number of partner integrations
- How API is enhancing goals of the company as a whole rather than simply determining how many people have begun to integrate

API Methodology

Consists of 5 phases in the case of agile strategy:

- Domain analysis or API description
- Architecture design
- Prototyping
- Building API for production, then
- Publishing the API

Domain Analysis or API Description

Define your use cases for domain analysis. Who are the participants? Are they external or internal? Which API solutions do consumers want to build with the API? Which other API solutions would be possible with the API?

Activities participant takes on consumer view: What would the API that the consumer wants to use look like? What apps does the consumer want to build? What data or domain objects does the consumer want to use in his app?

Break activities into steps or write down the usage scenario.

- A dependent resource can not exist without another.
    - For example, the association of a podcast and its consumer can not be determined unless the podcast and its consumer are created.
- An independent resource can exist without another.
    - For example, a podcast resource can exist without any dependency.
- An associative resource exists independently but still has some kind of relation, i.e., it may be connected by reference.
    - As mentioned above

The next step is to identify possible transitions between resource states. Transitions between states provide an indicator of the HTTP method that needs to be supported. For the example of the podcast which could be added to a playlist, let's analyize different states:

Table 2-1. *Domain analysis example*

State	Operation	Domain Objet	Description
CREATE	POST	PODCASTS	Creates podcast
CREATE	POST	PLAYLISTS	Creates empty playlists
READ	GET/{podcast_id}	PODCAST	Reads podcast
UPDATE	PUT/{playlist_id}	PLAYLIST	Adds podcast to playlist

Also, verify by building a simple demo app. More than curl calls, this demo app provides a showcase for the API and can be reused in later stages.

Architecture Design

In this phase, API description or analysis phase is further redefined. Architecture design should make decisions about

- Protocol
- End points
- URI design
- Security
- Performance or availability

Detail design description:

- Resources
- Representations
- Content types
- Parameters
- HTTP methods
- HTTP status codes
- Consistent naming

In addition, look into resuability by looking at common APIs in the API Portfolio. Design decisions should be consistent with the API in the API Portfolio. The API Portfolio is a collection of APIs in an Enterprise, as discussed in Chapter 5.

As part of the design verfication, the demo app can be further extended here with design decisions. Issues to be verfied are that:

- the API is still easy to use;
- the API is simple and supports use cases; and,
- the API follows architectural style.

Prototyping

Prototyping is the preparation for the production implementation. Take complex use cases and implement end-to-end with high-fidelity. The prototype is incomplete and uses shortcuts. It can have a simulation of API if the back-end functionality is not available at the time of building the prototype. Once the prototype is made, then there is the acceptance test with pilot consumers as verification of the API. Pilot customers are internal customers from the API provider's team.

Implementation

The implementation needs to conform to the API description and needs to be delivered as soon as possible. In addition, the API is fully integrated into the back-end system and API Portfolio. This should have all the desired functionality as well as non-functional aspects of the API, like performance, security, and availability. At this stage, the API description should be stable since it has gone through multiple iterations. For verification, hand-picked API consumers could be identified at this stage.

Publish

Publishing of the API does not require a lot of work, but this is a big milestone for the API. From an organizational perspective, the responsibility of the API is transferred from development to the operational unit. After publishing, there is no agility in the development process. Any change requires traditional change management process. As part of the verfication, there is analysis on successful vs. failed API calls and documentation gaps which are supported by the maintenance team.

API Modeling

Modeling the schema for your API means creating a design document that can be shared with other teams, customers, or executives. A schema model is a contract between your organization and the clients who will be using it. A schema model is essentially a contract describing what the API is, how it works, and exactly what the endpoints are going to be. Think of it as a map of the API, a user-readable description of each endpoint, which can be used to discuss the API before any code is written. Figure 2-3 below shows the API Modeling framework where you have API specifications defined and generate API documentation. Also, generate server and client source code.

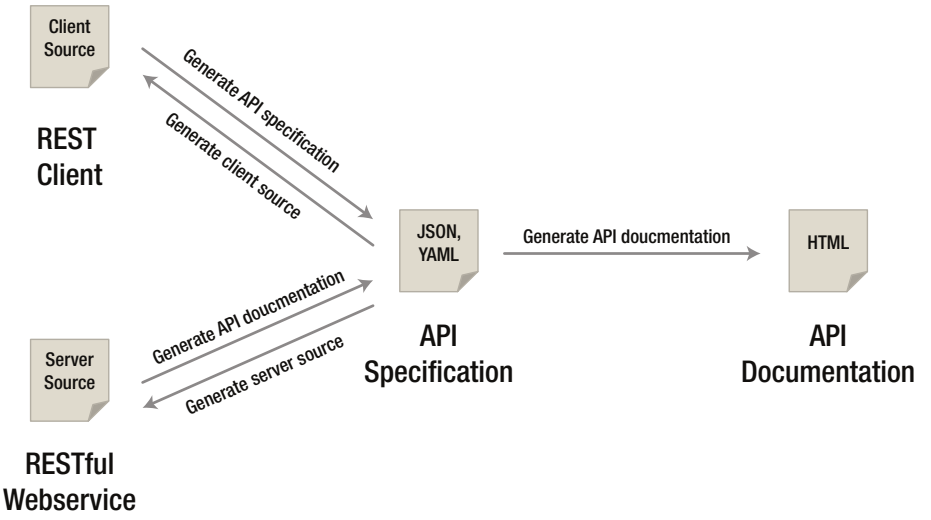


Figure 2-3. *API Modeling*

Creating this model before starting development helps you to ensure that the API you create will meet the needs described by the use cases you've identified. The three schema modeling systems and the markup languages they use are:

- RAML: markdown, relatively new. Good online modeling tool: RESTful API Modeling Language
- Swagger: JSON, large community
- Blueprint: markdown, low adoption

The RAML exercise in this chapter shows the modeling done for the podcast resource.

Each of the schema modeling languages has tools available to automate testing or code creation based on the schema model you've created, but even without this functionality the schema model helps you to have a solid understanding of the API before a single line of code is written.

Figure 2-4 below shows the API Modeling tool.

Figure 2-4. *API Modeling tool*

Comparison of API Modeling

Table 2-2. *Comparision of API Modeling tools*

Category	Property	RAML	API Blueprint	Swagger
What is behind name?	Format	YAML	Markdown (MOSN)	JSON
	Available at	Github	Github	Github
	Sponsored By	Mulesoft	Apiary	Reverb
	Current Version	1.0	1A3	2.0
	Initial Commit	Sep 2013	Apr 2013	Jul 2011
	Commercial Offering	Yes	Yes	Yes
How does it model REST?	Resources	X	X	X("api")
	Methods/ Actions	X("methods")	X("actions")	X("operations")

(*continued*)

Category	Property	RAML	API Blueprint	Swagger
	Query Parameters	X	X	X
	Path / URL Parameters	X	X	X
	Representation	X	X	X
	Header Parameters	X	X	X
	Documentation	X	X	X
	References	http://raml.org	https://apiblueprint.org	http://swagger.io
	Design	API-first	Design First	Existing API
	Code Generation	X		X
Who are customers?				APIGEE, Microsoft, Paypal

In summary:

- Swagger has a very strong modeling language for defining exactly what's expected of the system—very useful for testing and creating coding stubs for a set of APIs.
- RAML is designed to support a design-first development flow, and focuses on consistency.
- Apiary blueprint is more documentation-focused, with user-readable models and documentation as its first priority.

Each project brings different strengths and weaknesses to the table, and in the end it's really about what strengths you need and which weaknesses you cannot afford. Overall, RAML fared the best in these different categories and, while the developer community is not as large as the others, I think it's safe to say it will keep growing.

Overall Winner: RAML

Best Practices

REST is an architectural style and not a strict standard; it allows for a lot of flexibly. Because of that flexibility and freedom of structure, there is also a big appetite for design best practices. These best practices are discussed here in this section.

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. For Web API design, there should be only two base URLs per resource. Let's model an API around a simple object or resource (a customer) and create a Web API for it. The first URL is for a collection; the second is for a specific element in the collection:

- `/customers` - Collection
- `/customers/1` - Specific element

Boiling it down to this level will also force the verbs out of your base URLs. Keep verbs out of your URLs as shown in table below:

Table 2-3. *Nouns and verbs*

Resource	POST Create	GET Read	PUT Update	DELETE Delete
`/customers`	New customer	List customers	Bulk update	Delete all
`/customers/12`	-	Show customer 12	If exists update If not error	Delete customer 12

In summary:

- Use two base URLs per resource. Keep verbs out of your base URLs. Use HTTP verbs to operate on the collections and elements.
- The level of abstraction depends on your scenario. You also want to expose a manageable number of resources.
    - Aim for concrete naming and to keep the number of resources between 12 and 24.
- An intuitive API uses plural rather than singular nouns, and concrete rather than abstract nouns.
- 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 podcasts resource. Remember, we had two base URLs: /podcasts and /podcasts/1234. We're using HTTP verbs to operate on the resources and collections. Our podcasts belong to customers. To get all the podcasts belonging to a specific customer, or to create a new podcast for that customer, do a GET or a POST:
    - GET /customers/5678/podcasts
    - POST /customers/5678/podcasts

- Sweep complexity under the "?". Make it simple for developers to use the base URL by putting optional states and attributes behind the HTTP question mark. To get all customers in `sfo` city of `ca` state of `usa` country:
    - GET /customers?country=usa&state=ca&city=sfo

Error Handling

Many software developers, including myself, don't always like to think about exceptions and error handling, but it is a very important piece of the puzzle for any software developer, and especially for API designers. 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. Second, 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: Let's take a look at how three top APIs approach:

- Facebook

HTTP Status Code: 200

```
{"type" : "OauthException", "message":"(#803) Some of
the aliases you requested do not exist: foo.bar"}
```

- Twilio

HTTP Status Code: 401

```
{"status" : "401", "message":"Authenticate","code":
20003, "more info": "http://www.twilio.com/docs/
errors/20003"}
```

- Another example of error messaging from SimpleGeo

HTTP Status Code: 401

```
{"code" : 401, "message": "Authentication Required"}
```

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.

Error Code

Start by using the following 3 codes which should map to the 3 outcomes above. If you need more, add them. But you shouldn't need to go beyond:

- 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

Check out this good Wikipedia entry for all HTTP Status codes: `https://en.wikipedia.org/wiki/List_of_HTTP_status_codes`.

Versioning

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/dogs`).
- 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 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.
- There is a strong school of thought about putting format (xml or json) 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.

Partial Response

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 retweeted, 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:

- 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
```

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. As you see in this example, you can also put sub-objects in responses to pull in other information from additional resources.

Pagination

Make it easy for developers to paginate objects 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 and count.

To get records 50 through 75 from each system, you would use:

- Facebook - `offset` 50 and `limit` 2
- Twitter - `page` 3 and `rpp` 25 (records per page)
- LinkedIn - `start` 50 and `count` 25

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`

API Façade

Use the façade pattern when you want to provide a simple interface to a complex subsystem. Subsystems often get more complex as they evolve.

Implementing an API façade pattern involves three basic steps:

1. Design the ideal API—design the URLs, request parameters and responses, headers, query parameters, and so on. The API design should be self-consistent. This means you give the developers the information they need.
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.

API Solution Architecture

Developers and architects often think of APIs as a continuation of the integration-based architectures that have long been in use within enterprise IT. But this is a narrow view.

To understand the demands and requirements on APIs, let's discuss typical solutions that are enabled by APIs.

Figure 2-5 below shows API Solution Architecture.

API solutions typically consists of two components:

- Exposes API
- Exposed API resides server-side, e.g., in the cloud or on premise.
- Consumes API
- Web or mobile apps and embedded devices on IoT

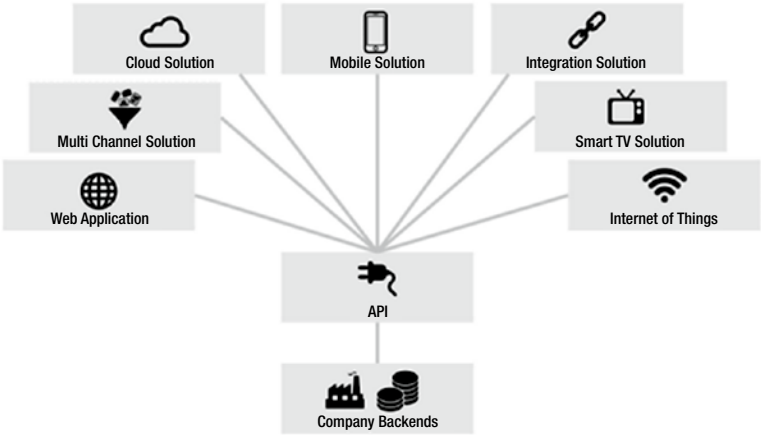


Figure 2-5. *API Solution Architecture*

Mobile Solutions

Mobile apps need to connect to the servers on the Internet to be useable at all or at least to be usable to their full potential—some business logic on the app and heavy duty processing logic on servers on the cloud. Functionality hosted on these servers can be reached by APIs calls. Data captured on mobile devices is sent to servers by APIs calls, which hands to services and then to databases. Data delivered by APIs needs to be lightweight. This ensures APIs can be consumed by devices with limited processing power. Typically, the mobile app provider provides the APIs for the mobile app.

Cloud Solutions

SaaS cloud solutions typically consist of a web application and APIs. The web application is visible for the consumers. Under the hood, cloud solutions usually offer an API as well. Examples: Dropbox, Salesforce, Workday, Oracle Cloud

Web Solutions

Web applications display dynamic web pages based upon user requests; web pages are created on the fly with data available from the back end. The web application pulls raw data from the APIs, processes the data (JSON, XML), and displays in HTML, e.g., podcast or customer API.

Integration Solutions

APIs provide capabilities which are essential for connecting, extending the integrating software. By integrating software APIs, businesses can connect with other businesses. The business of an enterprise can be expanded by linking business to a partner. Integration not only makes sense externally, but also internally for integrating internal systems.

Multi-channel Solutions

Today, an e-commerce system offers customers shopping on multiple platforms—mobile, web, tablet. It is required to provide a seamless experience when a consumer moves from one platform to another. This can be accomplished by providing a common API, which supports a multichannel maintaining state of user experience.

Smart TV Solutions

Smart TV offers not only TV channles, but provides interaction capablities. These are all implemented by API calls to the servers.

Internet-of-Things

The Internet of Things is made up of physical devices with an Internet connection. The device connects to smart functions (e.g., sensors, scanners, etc.) which are exposed on the Internet via APIs.

Stakeholders in API Solutions

In API Solutions, stakeholders are API Providers, API Consumers and End Users. We will discuss the roles of each here in this section.

API Providers

API providers develop, design, deploy, and manage APIs. API providers define the API portfolio, roadmap, and product mode. It is the responsiblity of an API provider to decide which functionality is exposed by the API. In the solution-driven approach, only those APIs are built which are required by the consumer. In the top-down approach, API providers provide APIs which are good from an internal perspective, e.g., from a reusability perspective.

API Consumers

Consumers need to know how to call API and build an API client. API providers should provide a demo app to consume their API for the consumers.

End users

End users do not call the API directly, but use the app developed by API consumers.

API DESIGN

Podcast

For the purposes of a Podcast API, these would be the use cases you want to support:

- Creating a new podcast
- Getting a list of podcasts, including matching particular podcast title
- Creating a new customer
- Customer subscribing to a podcast - Association

In summary, the resources and their methods will be as follows:

Detail design description:

- Resource - podcasts
  - Creating a podcast
    - Protocol - HTTP
    - End Points - `protocol:host:port/podcasts`
    - URI Design - `protocol:host:port/podcasts`
  - Searching a podcast
    - Protocol - HTTP
    - End Points - `protocol:host:port/podcasts`
    - URI Design - `protocol:host:port/podcasts?title=<title>`
- Representations
  - JSON

```
{ "podcasts" :
 [{ "id" : 1,
                        "title" : "itunes podcast",
                        "feed" : "http://www.itunes.com/",
                        ..
                        ..
                  },
                  {
                  }
            ]
           }
```

- Content types - `application/json`
- Query Parameters - `title`
- HTTP methods - GET, POST
- HTTP status codes - 200, 400, 500
- Resource - customers
  - Creating a customer
    - Protocol - HTTP
    - End Points - `protocol:host:port/customers`
    - URI Design - `protocol:host:port/customers`
  - Customer subscribing to a podcast
    - Protocol - HTTP
    - End Points - `protocol:host:port/customers`
    - URI Design - `protocol:host:port/customers/{id}/podcasts/{id}`
- Representations
  - JSON

```
{
  "customers" :
  [  { "id" : 1,
                "name" : "apple",
                "url" : http://www.apple.com/
                "podcasts" : { "podcasts" :
               [{"id" : 1,
                "title" : "itunes podcast",
                "feed" : "http://www.itunes.com/"
                    ..
                    ..
               }]
            }

           },

     ]
    }
```

- Content types - `application/json`
- Parameters - `name`
- HTTP methods - PUT or POST
- HTTP status codes - 200, 400, 500

API MODELING

RAML

To get started with the RAML API Designer, you will first need to create a (free) account on the Anypoint system, where Mulesoft maintains their RAML specific tools:

1. Sign up for Anypoint

   `https://anypoint.mulesoft.com/apiplatform`

2. From the API Administration board, select Add New API

   `https://anypoint.mulesoft.com/accounts/#/signin`

MODELING STEPS USING RAML TOOL

This tutorial walks you through modeling an API using RAML with a mock response at the end.

The Anypoint RAML editor features a bottom toolbar for adding sections to the model. This toolbar is context-sensitive; it will only offer appropriate sections based on where you are currently in the model.

Step 1: Enter the Root. Everything you enter in at the root (or top) of the spec applies to the rest of your API. This is going to come in very handy later as you discover patterns in how you build your API. The `baseURI` you choose will be used with every call made, so make sure it's as clean and concise as can be.

```
#%RAML 0.8
title: Podcast
version: v1
baseUri: http://api.podcast:8080/
```

Step 2: Enter the resources - Recalling how your API consumers will use your API, enter the following three resources under your root as per API design in previous exercise:

```
#%RAML 0.8
title: Podcast
version: v1
baseUri: http://api.podcast:8080/
/podcasts:
  /customers:
            /podcasts:
```

Step 3: Enter methods for the resources. You can add as many methods as you like to each resource.

```
#%RAML 0.8
title: Podcast
version: v1
baseUri: http://api.podcast:8080/
/podcasts:
  post:
  get:
/customers:
    post:
              get:
                    /podcasts
                    put:
```

Step 4: Enter URI parameters. The resources that we defined are collections of smaller, relevant objects. This is a URI parameter, denoted by surrounding curly brackets in RAML.

```
#%RAML 0.8
title: Podcast
version: v1
baseUri: http://api.podcast:8080/
/podcasts:
  post:
  get:
/customers:
    post:
              get:{id}:
                    /podcasts:{id}:
                    put:
```

Step 5: Enter query parameters. Start by adding query parameters under the GET method for podcasts. These can be specific characteristics, like the podcasts for a title.

```
#%RAML 0.8
title: Podcast
version: v1
baseUri: http://api.podcast:8080/
/podcasts:
  post:
  get:
   queryParameters:
     title

/customers:
    post:
              get:{id}:
                    /podcasts:{id}:
                    put:
```

Step 6: Enter responses. Responses MUST be a map of one or more HTTP status codes, and each response may include descriptions and examples.

```
#%RAML 0.8
title: Podcast
version: v1
baseUri: http://api.podcast:8080/
/podcasts:
  post:
  get:
   queryParameters:
     title:
            responses:
                    200:
                           body:
                                  application/json:
            { "podcasts" :
                    [{"id" : 1,
                       "title" : "itunes podcast",
                       "feed" : "http://www.itunes.com/",
                       ..
                       ..
                  },
                  {
                  }
             ]
            }
```

```
/customers:
    post:
              get:{id}:
                    /podcasts:{id}:
                    put:
```

Once you have modeled API, you can generate a document which could be shared with API consumers.

Wrapping Up

In this chapter we started with API design strategies and then looked into API creation process and modeling. Best practices for REST API design are discussed, followed by API solution architecture. We compared API modeling tools, designed an API for the podcasts suscription, and then modeled that using RAML.

CHAPTER 3

Introduction - XML, JSON

This chapter introduces basic concepts about XML and JSON. At the end of this chapter there is an exercise for environment setup.

What is XML?

e**X**tensible **M**arkup **L**anguage - XML is a text-based markup language which is standard for data interchange on the Web. As with HTML, you identify data using tags (identifiers enclosed in angle brackets, like this: `<...>`). Collectively, the tags are known as "markup." It puts a label on a piece of data that identifies it (for example: `<message>...</message>`). In the same way that you define the field names for a data structure, you are free to use any XML tags that make sense for a given application. Naturally, though, for multiple applications to use the same XML data, they have to agree on the tag names they intend to use. Here is an example of some XML data you might use for a messaging application:

```
<message>
 <to>you@yourAddress.com</to>
 <from>me@myAddress.com</from>
 <subject>XML Is Really Cool></subject>
 <text>
    How many ways is XML cool? Let me count the ways...
 </text>
</message>
```

Tags can also contain attributes (additional information included as part of the tag itself) within the tag's angle brackets. If you consider the information in question to be part of the essential material that is being expressed or communicated in the XML, put it in an element. For human-readable documents, this generally means the core content that is being communicated to the reader. For machine-oriented records formats, this generally means the data that comes directly from the problem domain. If you consider

S. Patni, *Pro RESTful APIs*, DOI 10.1007/978-1-4842-2665-0_3

the information to be peripheral or incidental to the main communication, or purely intended to help applications process the main communication, use attributes. The following example shows an email message structure that uses attributes for the `to`, `from`, and `subject` fields:

```
<message to=you@yourAddress.com from=me@myAddress.com
    subject="XML Is Really Cool">
  <text>
   How many ways is XML cool? Let me count the ways...
  </text>
</message>
```

One really big difference between XML and HTML is that an XML document is always constrained to be well-formed. There are several rules that determine when a document is well-formed, but one of the most important is that every tag has a closing tag. So, in XML, the `</to>` tag is not optional. The `<to>` element is never terminated by any tag other than `</to>`.

■ **Note** Another important aspect of a well-formed document is that all tags are completely nested. So you can have `<message>..<to>..</to>..</message>`, but never `<message>..<to>..</message>..</to>`.

An XML Schema is a language for expressing constraints about XML documents. There are several different schema languages in widespread use, but the main ones are Document Type Definitions (DTDs). It defines the legal building blocks of an XML document. It also defines the document structure with a list of legal elements and attributes.

XML Comments

XML comments look just like HTML comments:

```
<message to=you@yourAddress.com from=me@myAddress.com
    subject="XML Is Really Cool">
<!-- This is comment -->
<text>
   How many ways is XML cool? Let me count the ways...
</text>
</message>
```

To complete this introduction to XML, note that an XML file always starts with a prolog. The minimal prolog contains a declaration that identifies the document as an XML document, like this:

```
   <?xml version="1.0"?>
```

The declaration may also contain additional information, like this:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
```

- **version**: Identifies the version of the XML markup language used in the data. This attribute is not optional.
- **encoding**: Identifies the character set used to encode the data. "ISO-8859-1" is "Latin-1", the Western European and English language character set. (The default is compressed Unicode: UTF-8.)
- **standalone**: Tells whether or not this document references an external entity or an external data type specification. If there are no external references, then "yes" is appropriate.

Why is XML Important?

It is important because it allows the flexible development of user-defined document types, which means that it provides a persistent, robust, non-proprietary, and verifiable file format which can be used for the storage and transmission of data for both on and off the Web. In addition, XML:

- provides plain text: plain text makes it readable;
- provides data identification: by use of tags, data can be identified;
- provides styleability: using XSLT (eXtensible StyLe Sheet), data can be made in a presentable form;
- is easily processed (XML parsers, as well as well-formed parsers);
- is hierarchical (through nested tags).

How can you use XML?

There are several basic ways to make use of XML:

- Document-driven programming, where XML documents are containers that build interfaces and applications from existing components
- Archiving: the foundation for document-driven programming, where the customized version of a component is saved (archived) so it can be used later
- Binding, where the DTD or schema that defines an XML data structure is used to automatically generate a significant portion of the application that will eventually process that data

Pros and Cons of XML

Some of the pros and cons of XML are explained below.

- Pros
  - Readable and editable by developers
  - Error checking by means of schema and DTDs
  - Can represent complex hierarchies of data
  - Unicode gives flexibility for international operation
  - Plenty of tools in all computer languages for both creation and parsing
- Cons
  - Bulky text with low payload/formatting ratio (but can be compressed)
  - Both creation and client-side parsing are CPU intensive
  - Common word processing characters are illegal (MS Word "smart" punctuation, for example)
  - Images and other binary data require extra encoding

What is JSON?

JSON or JavaScript Object Notation is a lightweight text-based open standard designed for human-readable data interchange. Conventions used by JSON are known to programmers, which include those with knowledge of C, C++, Java, Python, Perl, etc.

- The format was specified by Douglas Crockford.
- It was designed for human-readable data interchange.
- It has been extended from the JavaScript scripting language.
- The filename extension is `.json`.
- JSON Internet media type is `application/json`.
- JSON is easy to read and write.
- JSON is language-independent.

JSON Syntax

In this section we will discuss what JSON's basic data types are and syntax. Figure 3-1 shows basic data types of the JSON.

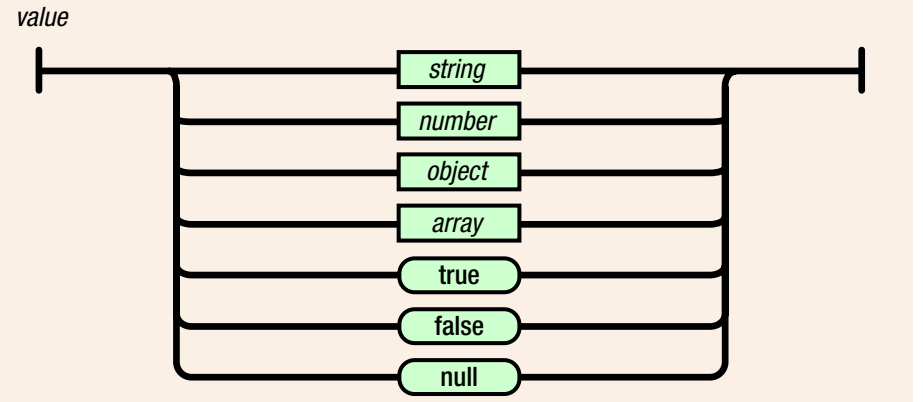


Figure 3-1. *Basic data types*

Strings

Strings are enclosed in double quotes, and can contain the usual assortment of escaped characters.

Numbers

Numbers have the usual C/C++/Java syntax, including exponential (E) notation. All numbers are decimal—no octal or hexadecimal.

Objects

An object is an unordered set of a name/value pair. The pairs are enclosed within braces ({ }).

Example:

```
{ "name": "html", "years": 5 }
```

Pairs are separated by commas. There is a colon between the name and the value. The syntax of a JSON object is shown in Figure 3-2.

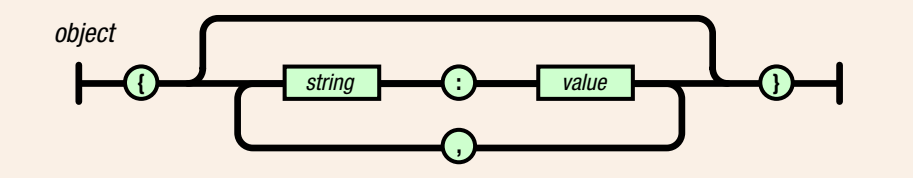


Figure 3-2. *JSON object*

Arrays

An array is an ordered collection of values. The values are enclosed within brackets. The syntax of JSON arrays is shown in Figure 3-3.

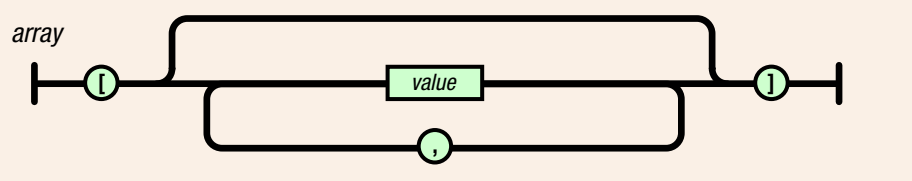


Figure 3-3. *JSON arrays*

Booleans

It can have either true or false values.

Null

The value is that it's empty.

Why is JSON Important?

There is a reason why JSON is becoming very popular as a data exchange format (more important than it being less verbose than XML): programmers are sick of writing parsers! But "wait," you say. "Surely there are XML parsers available for you to use so that you don't have to roll your own." Yes, there are. But while XML parsers handle the low-level syntactic parsing of XML tags, attributes, etc., you still need to walk the DOM tree or, worse, build one yourself with nothing but a SAX parser (Objective-C iPhone SDK I'm looking at you!). And that code you write will of course depend on whether the XML you need to make sense of looks like this:

```
1   <person first-name="John" last-name="Smith"/>
```

or this:

```
1   <person>
2   <first-name>John</first-name>
3   <last-name>Smith</last-name>
4   </person>
```

or this:

```
<object type="Person">
<property name="first-name">John</property>
<property name="last-name">Smith</property>
</object>
```

or any of the myriad of other ways one can conceive of expressing the same concept (and there are many). The standard XML parser does not help you in this regard. You still need to do some work with the parse tree.

Working with JSON is a different, and superior, experience. First, the simpler syntax helps you avoid the need to decide between many different ways of representing your data (as we saw above with XML), much less which rope to hang yourself with. Usually there is only one straightforward way to represent something:

```
{ "first-name" : "John",
  "last-name" : "Smith" }
```

How can you use JSON?

The following discusses how you can use JSON.

- It is used while writing JavaScript-based applications that include browser extensions and websites.
- JSON format is used for serializing and transmitting structured data over a network connection. It is primarily used to transmit data between a server and web applications.
- Web services and APIs use JSON format to provide public data.

Pros and Cons of JSON

The following are pros and cons of JSON:

Pros:

- Easy to read/write/parse
- Reasonably succinct (compared with XML, for instance)
- Common “standard” with many libraries available

Cons:

- Not as light as binary formats
- Can’t use comments

- It's "encapsulated," meaning that you can't readily stream/append data, but have to break it up into individual objects. XML has the same problem, whereas CSV does not.
- Difficult to describe the data you're presenting (easier with XML)
- Unable to enforce, or validate against, a structure/schema

XML - JSON Comparison

This section compares XML and JSON based upon different properties.

Table 3-1. *XML - JSON comparison*

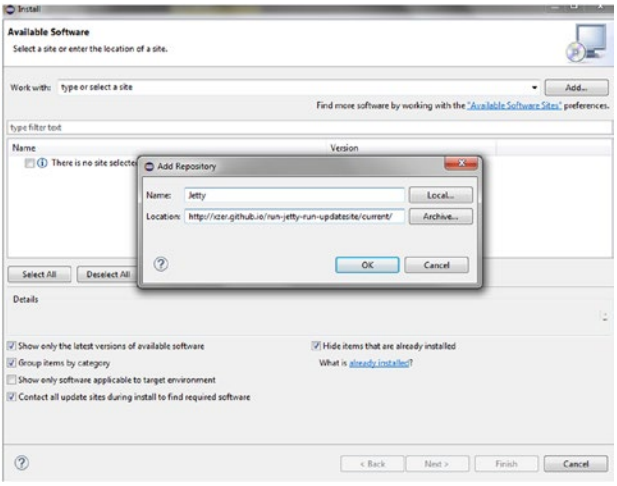
Property	XML	JSON
Simplicity	XML is simple, human-readable	But JSON is much simpler than XML as well as human-readable
Self- Describing	Yes	Yes
Processing	XML is processed easily.	JSON is processed more easily because its structure is simpler.
Performance	Not optimized for performance due to tags	Faster than XML because of size
Openness	XML is open.	JSON is at least as open as XML, perhaps more so because it is not in the center of a corporate/political standardization struggle.
Object- Oriented	XML is document-oriented	JSON is data-oriented. JSON can be mapped more easily to object-oriented systems.
Interoperability	XML is interoperable.	JSON has the same interoperability potential as XML.
Internationalization	Supports unicode	Supports unicode
Extendability	XML is extensible.	JSON is not extensible because it does not need to be. JSON is not a document markup language, so it is not necessary to define new tags or attributes to represent data in it.
Adoption	XML is widely adopted by industry.	JSON is just beginning to become known. Its simplicity and the ease of converting XML to JSON makes JSON ultimately more adoptable.

ENVIRONMENT SETUP AND HELLO FROM REST EXERCISE

Environment

Table 3-2 lists the software needed for the labs. The lab in this chapter sets you up with the environment and helps you validate the initial environment for "hello from REST".

Table 3-2. *Software for installation*

Software	Usage	Installation Notes
JDK 8	Required for lab projects	Download binary for your machine. Execute binary and follow the instructions.
Eclipse - Mars	Development IDE	
Jetty or Tomcat	Running labs in IDE	Set up JETTY in Eclipse. Follow the instructions below. 1. Go to Eclipse ➤ Install New Software menu. 2. Click on Add and type Jetty for Name and `http://eclipse-jetty.github.io/update/` for Location. 3. Pressing OK will prompt with a screen for "Terms and Conditions." Review these and select accept to install JETTY plug-in in Eclipse: 

(*continued*)

Table 3-2. (*continued*)

Software	Usage	Installation Notes
Community MySQL or MS SQL	Database - RDBMS	Download binary for your machine. Execute binary and follow the instructions to install it.
Maven Or Ant	Build tool	
Curl	Running REST API from command-line	
Postman	Running REST API from browser	

Hello from REST

This lab objective is:

- validation of the setup of the development environment;
- running “Hello from REST” in Jetty configured in Eclipse.

Prerequisite

You have JDK installed and Eclipse Mars setup with JETTY plug-in.

The image shown below depicts how REST URI is processed step by step from client to server and then to client.

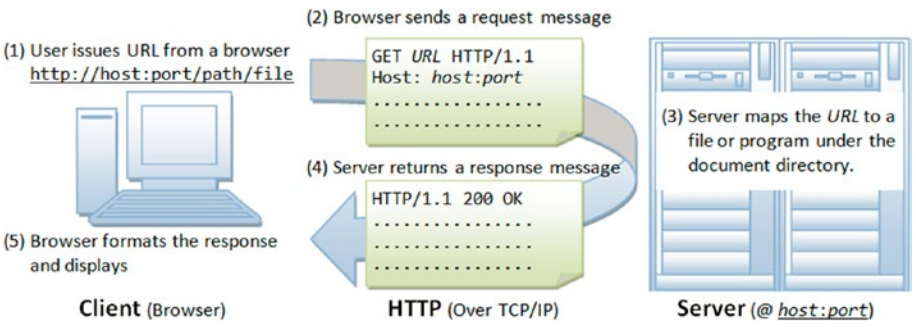
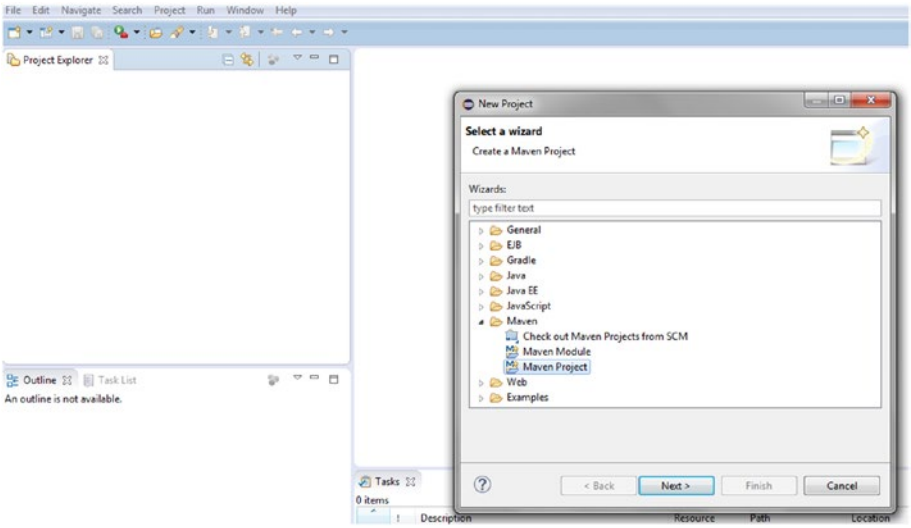


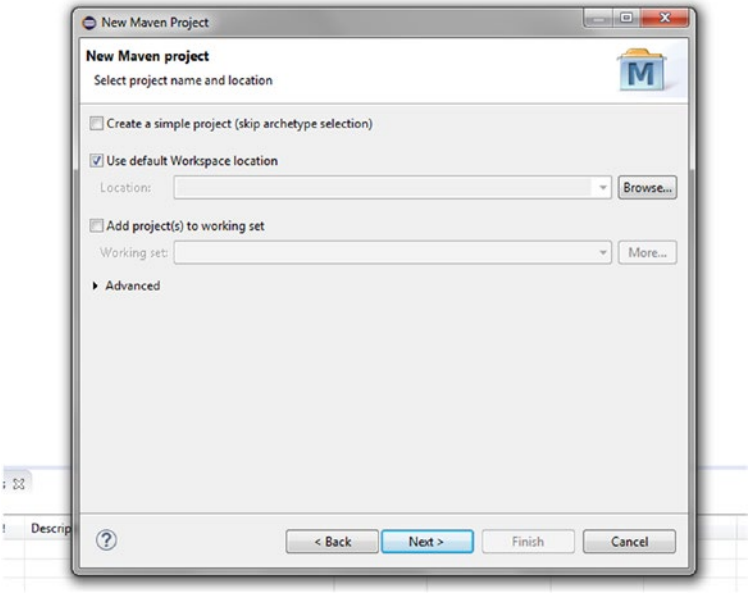
Figure 3-4. *REST URI processing*

Instructions

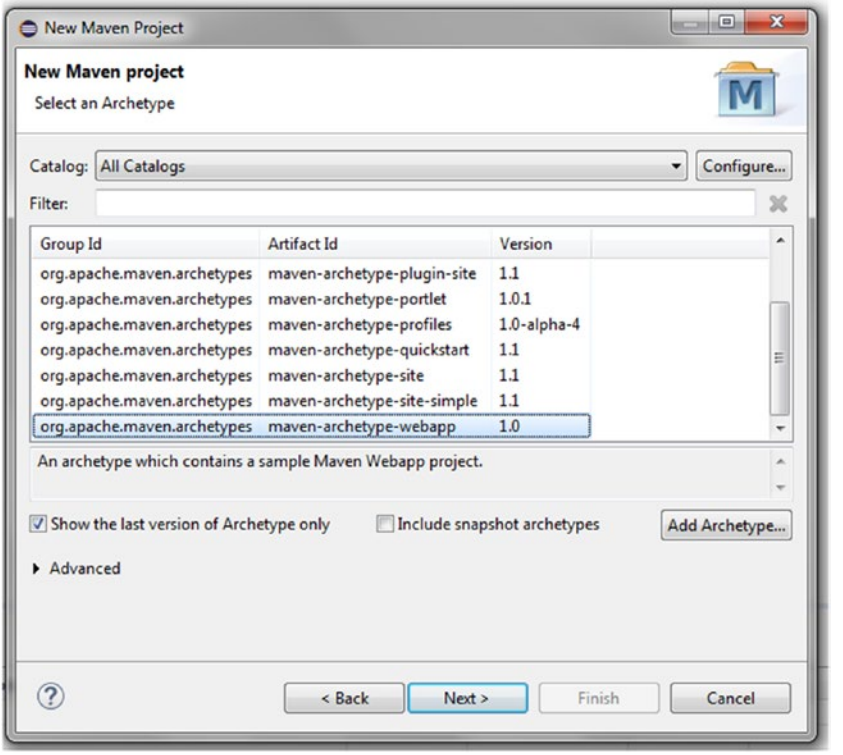
1. Create new project - File ➤ New ➤ Project ➤ Maven ➤ Maven Project



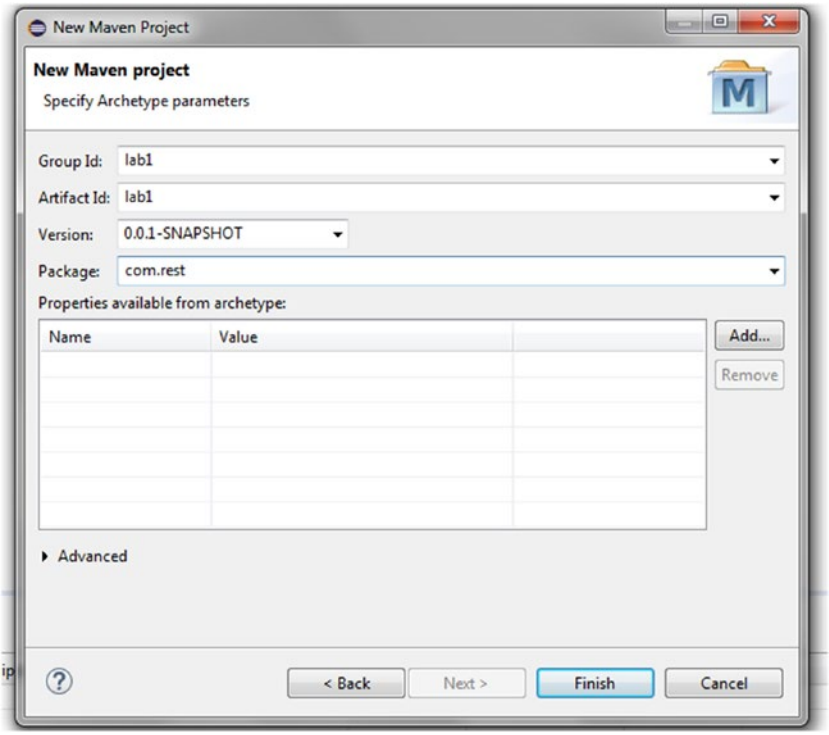
2. Press Next. It will show following screen. Skip this screen by pressing Next.



3. Select archetype as maven-archtype-webapp as below.



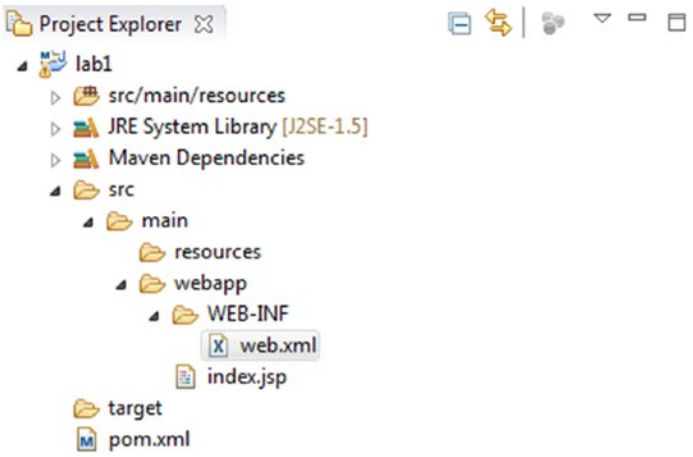
4. Fill information as below in the next screen for the Maven project and press Finish.



5. Add Maven dependencies in `pom.xml` for Jersey and servlet.

```
<dependency>
<groupId>com.sun.jersey</groupId>
<artifactId>jersey-server</artifactId>
<version>1.19</version>
<scope>compile</scope>
</dependency>
<dependency>
<groupId>com.sun.jersey</groupId>
<artifactId>jersey-servlet</artifactId>
<version>1.19</version>
<scope>compile</scope>
</dependency>
```

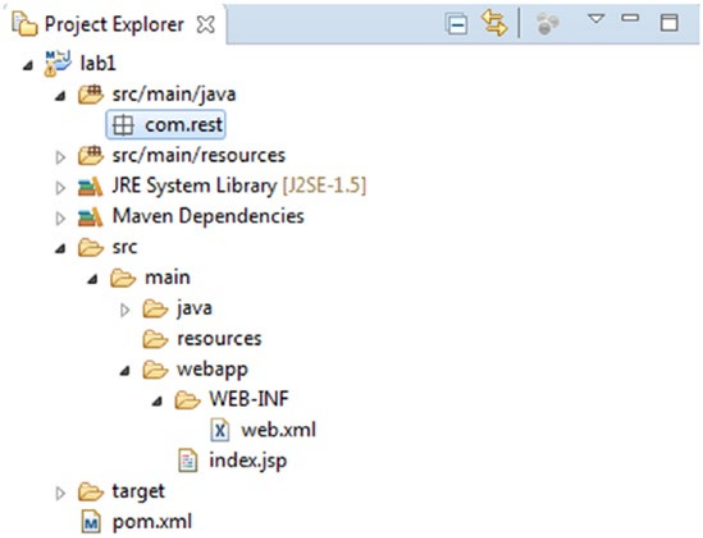
6. Add Jersey servlet configuration in `web.xml`.



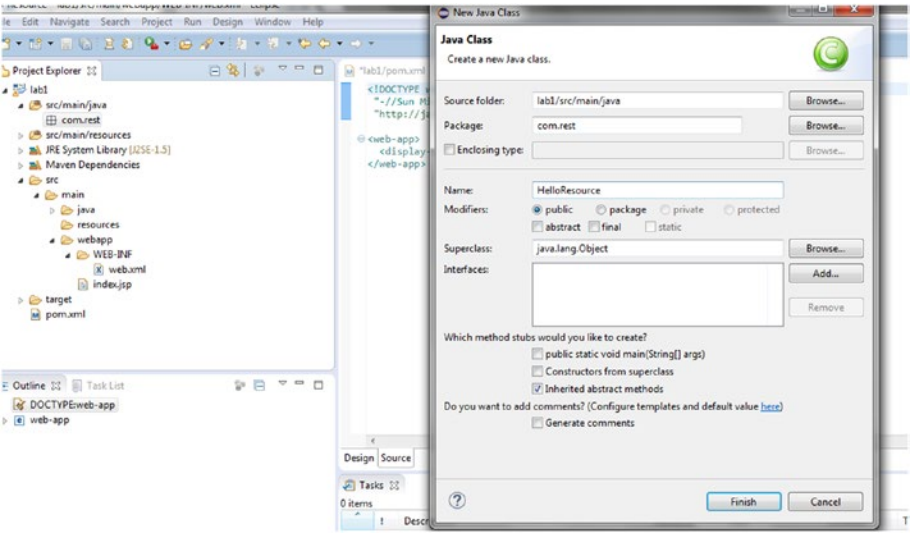

```
<web-app>
<display-name>Archetype Created Web Application</display-name>
<servlet>
<servlet-name>Jersey REST Service</servlet-name>
<servlet-class>com.sun.jersey.spi.container.servlet.
ServletContainer</servlet-class>
<init-param>
<param-name>com.sun.jersey.config.property.packages</param-name>
<param-value>com.rest</param-value>
</init-param>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>Jersey REST Service</servlet-name>
```

```
<url-pattern>/rest/*</url-pattern>
</servlet-mapping>
</web-app>
```

7. Select main ➤ New ➤ Folder ➤ java
8. Select `src/main/java`. Then New Package ➤ com.rest. You should see following:



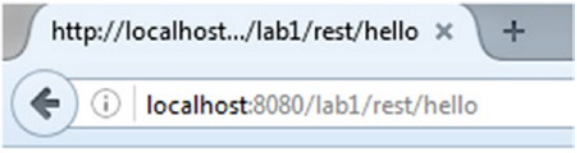
9. Create a new class HelloResource in com.rest package:



```
package com.rest;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
@Path("hello")
public class HelloResource {
  @GET
  @Produces(MediaType.TEXT_HTML)
  public String sayHtmlHello() {
  return "Hello from REST;
  }
}
```

10. Select `pom.xml`, right-click and then Run As ➤ Run Jetty.

11. Open browser and type in URL `http://localhost:8080/lab1/rest/hello` to see the results as below.



Hello from REST

■ **Note** If you are using NEO or later version of Eclipse IDE in the URI you do need to to specify lab1. URI will look like `localhots:8080/rest/hello`. This is true with all other excercises.

CHAPTER 4

Introduction to JAX-RS

This chapter introduces basic concepts about JAX-RS. At the end there are exercises using XML and JSON representations with JAX-RS.

JAX-RS Introduction

Java API for RESTful Web Services (JAX-RS) is a Java programming language API spec that provides support in creating web services according to the REST architectural pattern. JAX-RS is a collection of interfaces and Java annotations that simplifies development of server-side REST applications. By using JAX-RS technology, REST applications are easier to develop and easier to consume when compared to other types of distributed systems. The following are salient features provided by JAX-RS:

- POJO-based resource classes
- HTTP-centric programming model
- Entity format independence
- Container independence
- Included in Java EE
- Is a standardized API for building and consuming RESTful web services
- Application: specifies which Java classes service which requests
- Resources: base Java code and annotations for building RESTful services
- Providers: facilitates for marshaling and un-marshaling of data (POJO Customer)
- Client API: base Java code and annotations for building RESTful service clients
- Filters and interceptors: facilities for inserting code to execute throughout the request-response invocation chain

S. Patni, *Pro RESTful APIs*, DOI 10.1007/978-1-4842-2665-0_4

- Validation: provides annotations-based validation for incoming data. Example:

```
@NotNull  @FormParam ( "firstname" ) String firstname,
        @NotNull  @FormParam ( "lastname" ) String lastname,
                    @Email  @FormParam ( "email" ) String
                    email)
```

- Asynchronous processing: facilities for client and provider-side asynchronous requests for long-running requests (@Suspended)
- @Context: provides client and provider implementation classes with access to useful objects from the runtime environment, e.g., header (less servlet)
- Environment: provides provider implementation classes with servlet information

Figure 4-1 below has an example of JAX-RS API for getting balance for an ATM account. It shows resources, HTTP method, built-in serialization, and URI parameter injection.

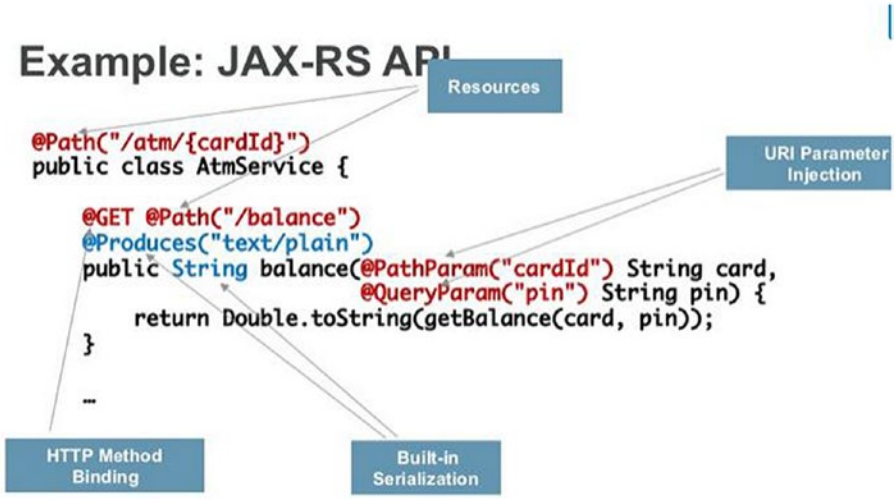


Figure 4-1. *Example of JAX-RS*

Input and Output Content Type

The following are media types supported by JAX-RS. These should be used in request and response annotations for contents produced or consumed by a REST resource.

- `APPLICATION_JSON: application/json`
- `APPLICATION_XML: application/xml` (encoding is used)
- `TEXT_HTML: text/html`
- `TEXT_PLAIN: text/plain`
- `TEXT_XML: text/xml`

The difference between application and text media types is the internationalization. The application supports encoding for internationalization.

Examples:

```
@Produces("application/json")
@Consumes("text/xml)
```

JAX-RS Injection

A lot of JAX-RS involves pulling information from an HTTP request and injecting it into a Java method. You may be interested in only a fragment of the incoming URI. You might be interested in a URI query string value. The client might be sending critical HTTP headers or cookie values that your service needs to process the request. JAX-RS lets you grab this information as you need it through a set of injection annotations and APIs.

There are a lot of different things JAX-RS annotations can inject. Here is a list of those provided by the specification:

- You need to use the Path annotation in JAX-RS to define a URI matching pattern for incoming HTTP requests. You can place it on a class or on one or more methods. If you want a class to receive HTTP requests, you must annotate it with at least `@Path("/")`. This annotated class is then called a JAX-RS root resource.
- To use the `@Path` annotation you provide a URI expression that is relative to the context root of your JAX-RS application.

  ```
  @javax.ws.rs.Path
  ```

- `PathParam` allows you to extract values from URI template parameters.

  ```
  @javax.ws.rs.PathParam
  ```

- `QueryParam` allows you to extract values from URI query parameters.

  `@javax.ws.rs.QueryParam`

- `FormParam` allows you to extract values from posted form data.

  `@javax.ws.rs.FormParam`

- `HeaderParam` allows you to extract values from HTTP request headers.

  `@javax.ws.rs.HeaderParam`

- `CookieParam` allows you to extract values from HTTP cookies set by the client.

  `@javax.ws.rs.CookieParam`

- `MatrixParam` allows you to extract values from URI matrix parameters.

  `@javax.ws.rs.MatrixParam`

- The `Context` class is the all-purpose injection annotation. It allows you to inject various helper and informational objects that are provided by the JAX-RS API.

  `@javax.ws.rs.core.Context`

The following are a few examples of JAX-RS injections.

Path Parameter

One parameter:

```
@Path("/customers/{id}")
public String getCustomer(@PathParam("id") int id);
```

Multiple parameters:

```
@Path("/products/{name}-{version}")
public String getProduct(@PathParam("name) String name, @
PathParam("version") String version)
```

Query Parameter

Requesting subset by qualifying with query parameter:

```
http://restcalss/products?start=0&count=10
@GET
@Produces("application/json")
public String getProducts(@QueryParam("start") int start, @
QueryParam("count") int count);

public String getProducts(@Context Uriinfo info)
String info.getQueryParameters().getFirst("start");
```

Cookie Parameter

Use `@CookieParam` to retrieve individual value:

```
public string get(@CookieParm("userId" String userId)
```

Header Parameter

Used for injecting HTTP header values:

```
public String get(@HeaderParam("Accept") String accept)
```

Form Parameter

`@FormParam` is used to retrieve information from the request body of HTML:

```
@Path("/product")
@POST
createProduct( @FormParm("name") String, productName, @
FormParm("description") String description,..
```

Matrix Parameter

Matrix parameters are used to qualify individual path segments, not the complete URI:

```
Response getBooks(@PathParam("year") String year,
@MatrixParam("author") String author,
@MatrixParam("country") String country) {
```

Consider the following request that is asking for all the books published in year 2016 by author bill in country usa.

```
"/books/2016;author=bill;country=usa"
```

getBooks is called with year set to 2016, author set to bill, and country set to usa.

REST Implementation

Now it's time to actually write a REST implementation with JAX-RS.

JAX-RS WITH XML

This exercise uses JAX-RS to implement CRUD (create, read, update, delete) operations using XML as the data representation. Create is implemented with HTTP POST. It will add a customer object to in-memory array. Read will GET customer object for a customer id. Update will update customer object fields via HTTP PUT. Delete will delete a customer for an id.

This exercise uses the configuration you have completed in the previous chapter, including the same `pom.xml` and `web.xml` files.

Customer Object - XML

The following is a representation of customer in XML.

```
<customer>
  <firstname>Bill</firstname>
  <lastname>Clark</lastname>
  <email>bill.clark@gmail.com</email>
</customer>
```

Customer Object - JAVA

This is a representation of customer domain object in Java.

```
package com.rest.domain;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlElement;
@XmlRootElement(name="customer")
public class Customer   {
  // Maps a object property to a XML element derived from property name.
```

```
  @XmlElement public int id;
  @XmlElement public String firstname;
  @XmlElement public String lastname;
  @XmlElement public String email;
}
```

Customer Resource - JAX-RS

Here is the customer resource implementing CRUD operations using JAX-RS injections.

```
package com.rest.resource;

import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.atomic.AtomicInteger;
import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.Response;

@Path("customers")
public class CustomerResource {
// ConcurrentHashMap - A hash table supporting full concurrency of
retrievals and adjustable expected concurrency for updates.
static private Map<Integer, Customer> customerDB = new
ConcurrentHashMap<Integer, Customer>();

// An AtomicInteger is used in applications such as atomically
incremented counters
static private AtomicInteger idCounter = new AtomicInteger();

@POST
@Consumes("application/xml")
public Customer createCustomer(Customer customer) {
  customer.id = idCounter.incrementAndGet();
  customerDB.put(customer.id, customer);
  return customer;
}
```

```
@GET
@Path("{id}")
@Produces("application/xml")
public Customer getCustomer(@PathParam("id") int id) {
 Customer customer = customerDB.get(id);
 return customer;
}

@PUT
@Path("{id}")
@Consumes("application/xml")
public void updateCustomer(@PathParam("id") int id, Customer update)  {
   Customer current = customerDB.get(id);
   current.firstname = update.firstname;
   current.lastname = update.lastname;
   current.email = update.email;
   customerDB.put(current.id, current);
}

@DELETE
@Path("{id}")
public void deleteCustomer(@PathParam("id") int id) {
   Customer current = customerDB.remove(id);
   if (current == null) throw new WebApplicationException(Response.
Status.NOT_FOUND);
  }
}
```

CRUD Operations with CURL

CURL is a command line tool you can use to invoke REST URIs. Below are CURL requests for each CRUD operation.

```
curl -H "Content-Type: application/xml" -X POST
-d  "<customer><firstname>Bill</firstname><lastname>Burke</
lastname><email>bill.burke@gmail.com</email></customer>" http://
localhost:8080/lab2/rest/customers

curl -H "Content-Type: application/xml" -X GET http://localhost:8080/
lab2/rest/customers/1

curl -H "Content-Type: application/xml" -X PUT -d
"<customer><firstname>Ed</firstname><lastname>Burke</lastname><email>ed.
burke@gmail.com</email></customer>" http://localhost:8080/lab2/rest/
customers/1

curl -H "Content-Type: application/xml" -X DELETE  http://
localhost:8080/lab2/rest/customers/1
```

The screenshot below shows results of CURL command execution.

JAX-RS WITH JSON

This exercise uses JAX-RS to implement CRUD (create, read, update, delete) operations using JSON as the data representation mechanism. Create is implemented with HTTP POST. It will add a customer object to in-memory array. Read will GET customer object for a customer id. Update will update customer object fields via HTTP PUT. Delete will delete a customer for an id. In addition, get all customers will get a list of customers in the collection. Also, if a customer id does not exist, a JSON representation of error messages based upon best practices will be returned in response.

Customer Object - JSON

The following is a representation of customer sample data in JSON.

```
{ "customer" : [ {
  "firstname" : "Bill",
  "lastname" : "Clark",
  "email : "bill.clark@gmail.com"
}]}
```

For this exercise there are changes to `pom.xml` and `web.xml` files required to support JSON representation.

POM.XML Updates

`pom.xml` updates for Glassfish and Jackson for JSON

```
<dependency>
<groupId>org.glassfish.jersey.containers</groupId>
<artifactId>jersey-container-servlet</artifactId>
<version>2.2</version>
</dependency>
```

```
<dependency>
<groupId>org.glassfish.jersey.core</groupId>
<artifactId>jersey-client</artifactId>
<version>2.2</version>
</dependency>
<dependency>
<groupId>com.fasterxml.jackson.jaxrs</groupId>
<artifactId>jackson-jaxrs-json-provider</artifactId>
<version>2.4.1</version>
</dependency>
```

web.xml Updates

`web.xml` updates for Glassfish

```
<web-app>
<display-name>Archetype Created Web Application</display-name>
<servlet>
<servlet-name>Jersey REST Service</servlet-name>
<servlet-class>org.glassfish.jersey.servlet.ServletContainer
</servlet-class>
<init-param>
<param-name>jersey.config.server.provider.packages</param-name>
<param-value>com.rest</param-value>
</init-param>

<load-on-startup>1</load-on-startup>

</servlet>
<servlet-mapping>
<servlet-name>Jersey REST Service</servlet-name>
<url-pattern>/rest/*</url-pattern>
</servlet-mapping>
</web-app>
```

CustomerResource Updates

The following are the `CustomerResource` updates for JSON. We'll add functionality to get all customers from the collection and add error handling.

Add the following to all methods to allow them to work with the JSON version of our API:

```
@Consumes({ "application/xml", "application/json" })
@Produces({ "application/xml", "application/json" })
```

Now we can get all customers. This request is routed through `@Path("customers")` to `getAll()` method to get a collection of customers. `getAll()` is a Java method which returns a list of customers.

```
@GET
@Produces({ "application/xml", "application/json" })
public Collection<Customer> getAll() {

  List<Customer> customerList = new ArrayList<Customer>(customerDB.
values());
  return customerList;
}
```

Add error handling:

```
final Customer customer = customerDB.get(id);
if (customer == null) {
  ErrorMessage errorMessage = new ErrorMessage("1001", "Customer not
found!", "http://localhost:8080/lab3/error1001.jsp", Response.Status.
NOT_FOUND);

  throw new NotFoundException(errorMessage);
}
```

ErrorMessage Object

Here is the `ErrorMessage` object:

```
package com.rest.exception;

import javax.xml.bind.annotation.XmlRootElement;
import javax.ws.rs.core.Response;
import javax.xml.bind.annotation.XmlElement;
@XmlRootElement(name="message")
public class ErrorMessage {
@XmlElement public String code;
@XmlElement public String description;
@XmlElement public String link;
@XmlElement public Response.Status status;
   public ErrorMessage(String code,String description, String link,
Response.Status status) {
        this.code = code;
        this.description = description;
        this.link = link;
        this.status = status;
   }
}
```

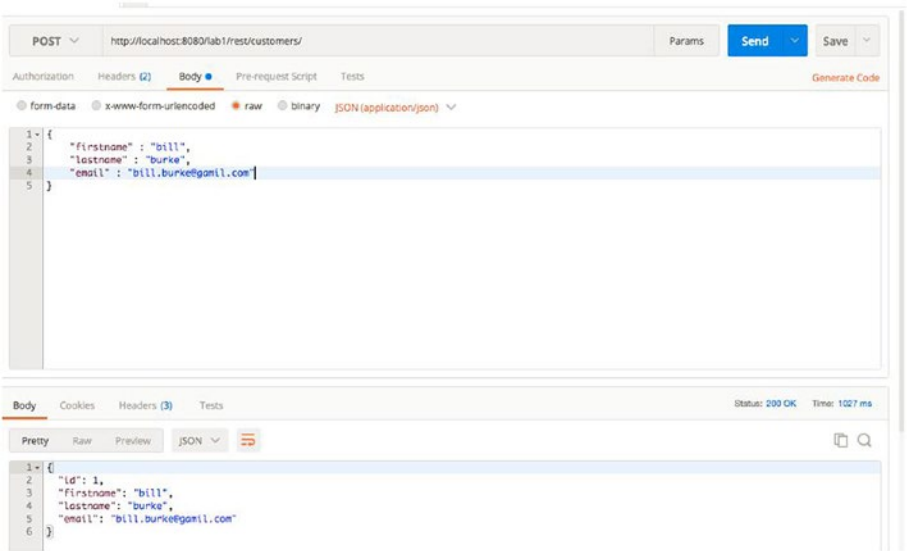
NotFoundException Class

And the `NotFoundException` class:

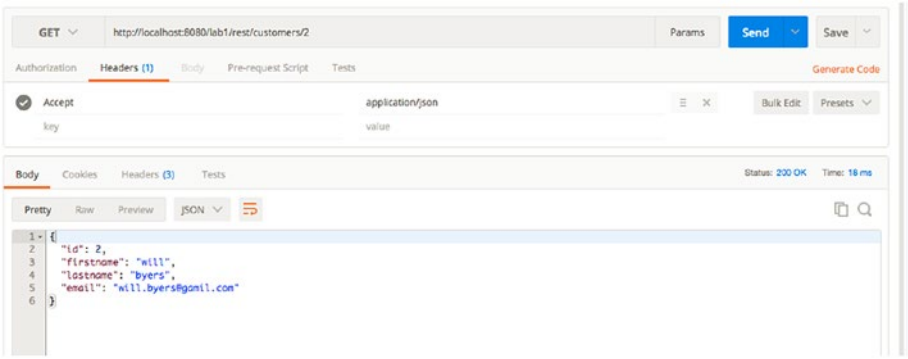
```
package com.rest.exception;

import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.Response;
public class NotFoundException extends WebApplicationException {
/**
*
*/
private static final long serialVersionUID = 1L;
/**
* Create a HTTP 404 (Not Found) exception.
* @param message the String that is the entity of the 404 response.
*/
public NotFoundException(ErrorMessage message) {
      super(Response.status(Response.Status.NOT_FOUND).
      entity(message).type("application/json").build());
    }
}
```

RESULTS IN POSTMAN

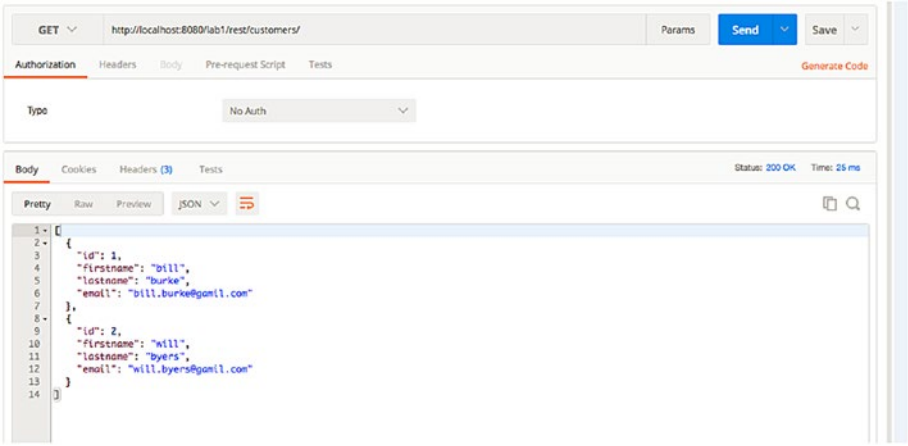
First, creating a customer:



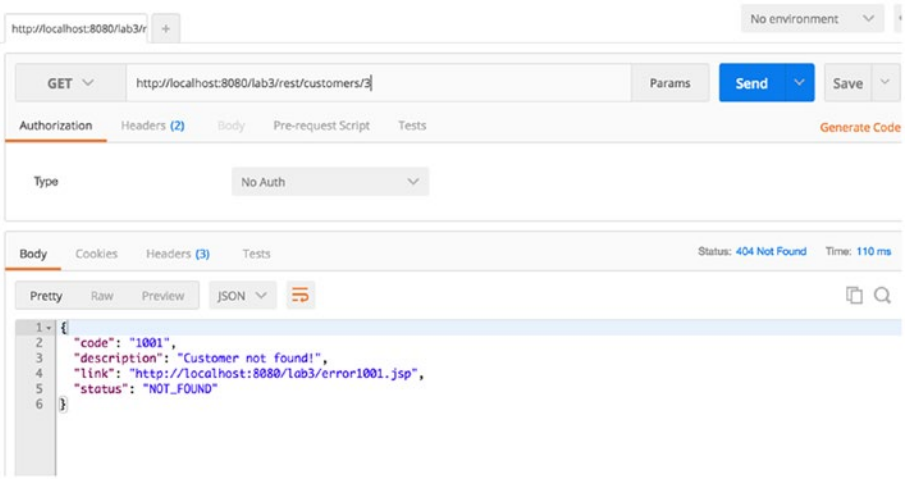
Get a customer for an id:



Get all customers:



Get a customer for non-existing id:



CHAPTER 5

API Portfolio and Framework

This chapter starts with API Portfolio Architecture and then gets into framework for API development. An overview of API framework starting from client to data is discussed and then focus is shifted to review services layer with an exercise implementing services layer.

API Portfolio Architecture

Usually, an organization does not have one API but several APIs. All the APIs in the portfolio need to be consistent with each other, reusable, discoverable, and customizable.

Requirements

API portfolio design is a concern for different API stakeholders. Both API consumers and producers have significant advantages over properly designed API portfolio and both parties formulate requirements for API portfolio regarding consistency, reuse, customization, discoverability, and longevity.

Consistency

An API solution, such as mobile app may use several APIs from the portfolio and the output of one API is the input of another. So consistency is required about data structures, representations, URIs, error messages, and behavior of the APIs. API consumers find it easier to work with if it behaves similar to the last one and delivers similar error messages.

Reuse

A consistent portfolio consists of many commonalities among the APIs. These commonalities can be factored out, shared, and reused. Reuse leads to a speed-up in the development. By reusing common elements, the wheel is not reinvented each time an API is built. Instead, a common library of patterns and know-how is shared and reused. Reuse can be realized in several ways.

S. Patni, *Pro RESTful APIs*, DOI 10.1007/978-1-4842-2665-0_5

- Reuse of API by several apps
- Reuse of API by multiple APIs
- Reuse of parts of API

APIs should not be developed for a specific consumer. APIs should always be used by several consumers, solutions, or projects.

Customization

There might be consumers who might have specific requirements from the APIs. if the consumers of APIs are not a homogenous group. In such a scenario, customizations are required to the APIs to meet a consumer's individual needs. This contradicts with Reuse requirements, but both can be realized at the same time.

Discoverability

To expand the usage of APIs, it should be easy for the APIs consumer to find and discover all APIs in an API portfolio. An API portfolio design needs to ensure that APIs can be found and all the information necessary for proper usage is available.

Longevity

This means that important aspects of the API do not change and stay stable for a long time. What needs to be stable is the signature of the API, the client-facing interface. A change in signature will break the apps built by the API consumer. For example, with IoT on "h/w devices" it is not easy to change.

How do we enforce these requirements—governance?

An API initiative is often regarded as an innovation lab of an enterprise. Imposing governance can contradict innovation. So to manage these conflicting requirements, an API portfolio may be split in two portfolios. One portfolio is dedicated to innovation and experiment. This portfolio requires light-weight governance processes. Another portfolio is dedicated to stable, productive APIs, which are offered to external API consumers.

Consistency

Each enterprise may implement its own set of consistency rules. When consistency rules are defined, consistency checks can be realized as manual or automated. Lightweight consistency checks can be realized by manual quality checks or review by a colleague. A complementary approach is by automated code generation based upon API description.

Reuse

There are two types of building blocks that are offered by an API Platform like Security, Logging and Error Handler. Any other functional commonality or reusable solution pattern can be realized as a composition of building blocks. You could have your "own" API or third-party APIs. Third party APIs could be integrated in an API Platform by creating an API Proxy on its "own" platform. This helps the consumer with homogenous security. API Proxy and API Platform architectures are discussed in the next chapter.

Customization

An API consumer is interested in data formatting and data delivery. Data gathering is, however, no concern to the API consumer. So these could be separated into two parts: one API we call "utility API" covers the data gathering; other API, which delivers data and formats to the consumer requirements, is called "consumer API." Utility APIs cannot be called directly by a consumer; only consumer APIs can call these.

Discoverability

This could be Manual or Automated. Manual: Discover by API catalog or yellow pages. Automated: SOAP-based thru UDDI and WSDL. REST: Limited with OPTIONS verb of HTTP.

Change Management

From an innovation or business perspective, there are forces to publish API as early as possible. From an IT Governance perspective, as late as possible. In a compromise solution, APIs are published early but only to pilot consumers, with the expectation that there will be changes and API will break the app. Changes are classified into three groups: backward compatible, forward compatible, and not compatible. Backward compatibility is given if the old client can interact with the new API (adding query, header or form parameter as long as they are optional; adding new fields in JSON or XML as long as they are optional; adding endpoint, e.g., new REST Resource; adding new operations to existing endpoints, e.g., in SOAP; adding optional fields to request interface; changing mandatory fields to optional fields in a existing API). Forward compatibility is given if a new client can interact with an old API. It's hard to achieve and generally it is nice to have it.

- Incompatible changes: If a change in API breaks the client, the change was incompatible.
- Removing: Renaming fields in data structures or parameters in request or response.
- Changing URI: e.g., hostname, port.
- Changing data structure: making a field the child of some other. Adding a new mandatory field in a data structure.

API Framework

As we have discussed, there are multiple solutions to an API, e.g., Web applications, Mobile Applications, etc. Each of these solutions talks to an API which is implemented through a multilayered architecture using design patterns. A **design pattern** is a general reusable solution to a commonly occurring problem within a given context in software **design.** A **design pattern** is not a finished **design** that can be transformed directly into source or machine code.

As shown, the Figure 5-1 multilayer framework consists of:

- Process APIs implemented by Services design pattern;
- System APIs implemented by Data Access Object design pattern;
- Experience APIs implemented by API Façade Layer design pattern.

Each layer is implemented using software engineeing design patterns.

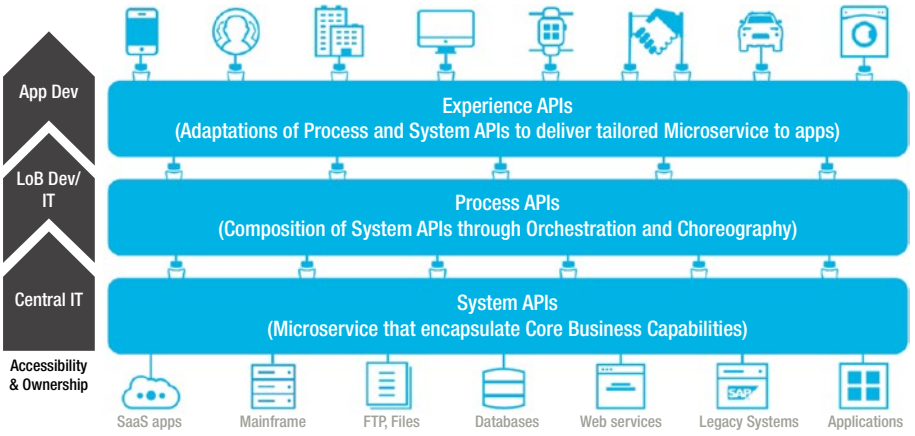


Figure 5-1. *APIs multilayered framework*

Process APIs - Services Layer

Services layer implements business logic of the application: The reusable logic: process-specific logic and the logic that interfaces with System APIs through Orchestration and Choreography. Orchestration (direct calls) in this sense is about aligning the Line of Business Dev/IT request with the applications, data, and infrastructure. Choreography, in contrast, does not rely on a central coordinator. Rather, each API involved in the choreography knows exactly when to execute its operations and with whom to interact.

System APIs - Data Access Object

These system APIs or system-level services are in line with the concept of an autonomous service which has been designed with enough abstraction to hide the underlying systems of record, e.g., databases, legacy systems, SaaS applications.

Typically, a data access object (DAO) is an object that provides an abstract interface to some type of database or other persistence mechanism. By mapping application calls to the persistence layer, DAO provides some specific data operations without exposing details of the system.

Experience APIs - API Facade

Both process and system APIs should be tailored and exposed to suit the needs of each business channel and digital touchpoint of solution architectures. The adaption is shaped by the desired digital experience and is what we call the Experience API. This is implemented by API Façade. The goal of an API Facade Pattern is to articulate internal systems and make them useful for the app developer providing a good APX (API experience).

We will review Servives layer in this chapter. Data Access Object and API Facade will be reviewed in Chapter 6 and Chapter 7.

Services Layer Implementation

Services layer implements the business logic of the application: the reusable logic, process-specific logic, and logic that interfaces with the legacy system. In the implementation of services layer, a design pattern dependency injection is used. The general concept between dependency injections is called Inversion of Control. A class `A` has a dependency to class `B` if class A uses class `B` as a variable. If dependency injection is used, then the class `B` is given to class `A` via the constructor of the class `A`. This is then called “construction injection.” If a setter is used, this is then called “setter injection.”

A class should not configure itself but should be configured from outside. A design based on independent classes/components increases the reusability. A software design based on dependency injection is possible with standard Java. Spring framework, which is used for the implemenation in the excercise, just simplifies the use of dependency injection by providing a standard way of providing the configuration and by managing the reference to the created objects. The fundamental functionality provided by the Spring Container is dependency injection. Spring provides a lightweight container, e.g., the Spring core container, for dependency injection (DI). This container lets you inject required objects into other objects. This results in a design in which the Java classes are not hard-coupled. The injection in Spring is either done via setter injection or via construction injection.

Figure 5-2 shows the spring frameworks. The Spring core container will be used for Dependency Injection from spring framework, which handles the configuration generally based on annotation.

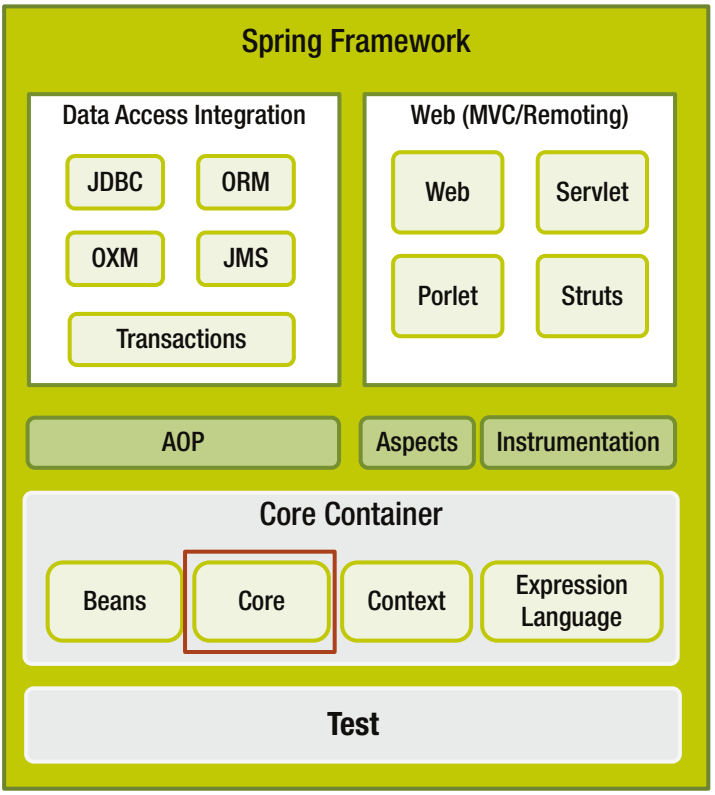


Figure 5-2. *Spring framework*

FRAMEWORK - SERVICES

This exercise uses podcast domain object to implement CRUD operations as well as search functionality on a field of podcast. Podcast domain object structure is pretty simple. There is an id, which identifies a podcast, and several other fields that we can see in the JSON representation below:

```
{
    "id":1,
    "title":"Quarks & Co - zum Mitnehmen-modified",
    "link":"http://itunes.com /podcasts/1/Quarks-Co-zum-Mitnehmen",
    "feed":"http://itunes.com /quarks.xml",
    "description":"Quarks & Co: Das Wissenschaftsmagazin",
    "insertionDate":1388213547000
}
```

POM.XML Updates

Table 5-1. *Maven project dependencies*

Project	Dependencies	Version	Description
org.glassfish.jersey.ext	jersey-spring3	2.14	Jersey extension module providing support for Bean Validation
org.glassfish.jersey.media	jersey-media-json-jackson	2.14	Jersey JSON Jackson
org.springframework	spring-core	4.1.4.RELEASE	Spring core
org.springframework	spring-context	4.1.4.RELEASE	Spring context
org.springframework	spring-web	4.1.4.RELEASE	Spring web

```
<properties>
    <spring.version>4.1.4.RELEASE</spring.version>
    <jersey.version>2.14</jersey.version>
</properties>
<dependencies>
    <!-- Jersey + Spring -->
    <dependency>
        <groupId>org.glassfish.jersey.ext</groupId>
        <artifactId>jersey-spring3</artifactId>
        <version>${jersey.version}</version>
        <exclusions>
            <exclusion>
                <groupId>org.springframework</groupId>
                <artifactId>spring-core</artifactId>
            </exclusion>
            <exclusion>
                <groupId>org.springframework</groupId>
                <artifactId>spring-web</artifactId>
            </exclusion>
            <exclusion>
                <groupId>org.springframework</groupId>
                <artifactId>spring-beans</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
```

```
        <!-- JSON -->
        <dependency>
            <groupId>org.glassfish.jersey.media</groupId>
            <artifactId>jersey-media-json-jackson</artifactId>
            <version>${jersey.version}</version>
        </dependency>

        <!-- Spring framework-->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>${spring.version}</version>
            <exclusions>
                <exclusion>
                    <groupId>commons-logging</groupId>
                    <artifactId>commons-logging</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-web</artifactId>
            <version>${spring.version}</version>
        </dependency>
    </dependencies>
```

web.xml Updates

`web.xml` updates are completed to include `applicationContext.xml` file of the Spring Framework.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
    <display-name>Demo - Restful Web Application</display-name>

    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
```

```
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            classpath:spring/applicationContext.xml
        </param-value>
    </context-param>
  <servlet>
          <servlet-name>Jersey REST Service</servlet-name>
          <servlet-class>org.glassfish.jersey.servlet.
          ServletContainer</servlet-class>
          <init-param>
              <param-name>jersey.config.server.provider.packages
              </param-name>
              <param-value>com.rest</param-value>
          </init-param>

         <load-on-startup>1</load-on-startup>

 </servlet>

<servlet-mapping>
    <servlet-name>Jersey REST Service</servlet-name>
    <url-pattern>/rest/*</url-pattern>
</servlet-mapping>

</web-app>
```

applicationContext

applicationContext.xml injects PodcastService in PodcastResource.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/
    spring-context-3.0.xsd">

    <context:component-scan base-package="com.rest.*" />

        <bean id="podcastService"
                  class="com.rest.service.PodcastServiceImpl" />

</beans>
```

Podcast Domain Object

Here is a POJO defining properties of podcast.

```
package com.rest.domain;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlElement;

@SuppressWarnings("restriction")
@XmlRootElement(name="podcast")
public class Podcast {
    @XmlElement public int id;
    @XmlElement public String link;
    @XmlElement public String feed;
    @XmlElement public String title;
    @XmlElement public String description;
    @XmlElement public String insertionDate;
}
```

PodcastResource

In the podcast resource we have CRUD operations for podcast, as well a new method for searching podcasts on the title field. Instance of `PodcastService`, which implements the logic for search, is injected in this class using `@Autowired` annotation.

```
@Path("podcasts")
public class PodcastResource {

    @Autowired
    PodcastService podcastService;

    @POST
    @Consumes("application/json")
    @Produces("application/json")
    public Podcast createPodcast(Podcast podcast) {

        podcastService.createPodcast(podcast);
        return podcast;
    }

    @GET
    @Path("{id}")
    @Consumes("application/json")
    @Produces("application/json")
    public Podcast getPodcast(@PathParam("id") int id) throws
    NotFoundException {
      Podcast podcast = podcastService.getPodcast(id);
       return podcast;
    }
```

```
    @GET
    @Consumes("application/json")
    @Produces("application/json")
    public List<Podcast> getPodcasts(@QueryParam("title")
    String title) {
      List<Podcast> podcasts = podcastService.getPodcasts(title);
       return podcasts;
    }

    @PUT
    @Path("{id}")
    @Consumes("application/json")
    @Produces("application/json")
    public void updatePodcast(@PathParam("id") int id,
    Podcast update) {
       podcastService.updatePodcast(id, update);
    }

    @DELETE
    @Path("{id}")
    @Consumes("application/json")
    @Produces("application/json")
    public void deletePodcast(@PathParam("id") int id)  {
      podcastService.deletePodcast(id);
    }
```

PodcastService

This is an interface having signatures of all the methods supported for podcast domain object.

```
public interface PodcastService {
    public Podcast getPodcast(int id) throws NotFoundException;
    public void createPodcast(Podcast podcast);
    public void updatePodcast(int id, Podcast update);
    public void deletePodcast(int id);
    public List<Podcast> getPodcasts(String title);
}
```

PodcastServiceImpl

This implements logic for searching podcasts on title. Also, all the methods for CRUD operations which have operations on in memory DB of podcasts are moved here.

```
public class PodcastServiceImpl implements PodcastService {

     static private Map<Integer, Podcast> podcastDB = new
     ConcurrentHashMap<Integer, Podcast>();
     static private AtomicInteger idCounter = new AtomicInteger();
```

```
// get podcast by id
public Podcast getPodcast(int id) throws NotFoundException {
    Podcast podcast = podcastDB.get(id);
    if (podcast == null) {
         ErrorMessage errorMessage = new ErrorMessage
         ("1001", "Podcast not found!", "http://localhost:8080/
         lab3/error.jsp", Response.Status.NOT_FOUND);

         throw new NotFoundException(errorMessage);
      }

    return podcast;
}

// add podcast
public void createPodcast(Podcast podcast) {

    podcast.setId(idCounter.incrementAndGet());
    podcastDB.put(podcast.getId(), podcast);

}

// update podcast
public void updatePodcast(int id, Podcast update)  {

    Podcast current = podcastDB.get(id);

      current.setDescription(update.getDescription());
      current.setTitle(update.getTitle());
      current.setFeed(update.getFeed());
      current.setLink(update.getLink());

      podcastDB.put(current.getId(), current);

}

// Delete podcast
public void deletePodcast(int id) {

    Podcast current = podcastDB.remove(id);

}

// Search podcast
public List<Podcast> getPodcasts(String title) {
```

```
        List<Podcast> podcastList = new ArrayList<Podcast>
                                    (podcastDB.values());
        List<Podcast> titleMatchedList = new ArrayList<Podcast>();

        for(int i = 0; i < podcastList.size(); i++)
        {
            Podcast podcast = podcastList.get(i);
            if ( podcast.getTitle().contains(title) )
                titleMatchedList.add(podcast);

        }

        return titleMatchedList;

    }

}
```

RESULTS IN POSTMAN

Figure 5-3 shows all podcasts URL and results in POSTMANN.

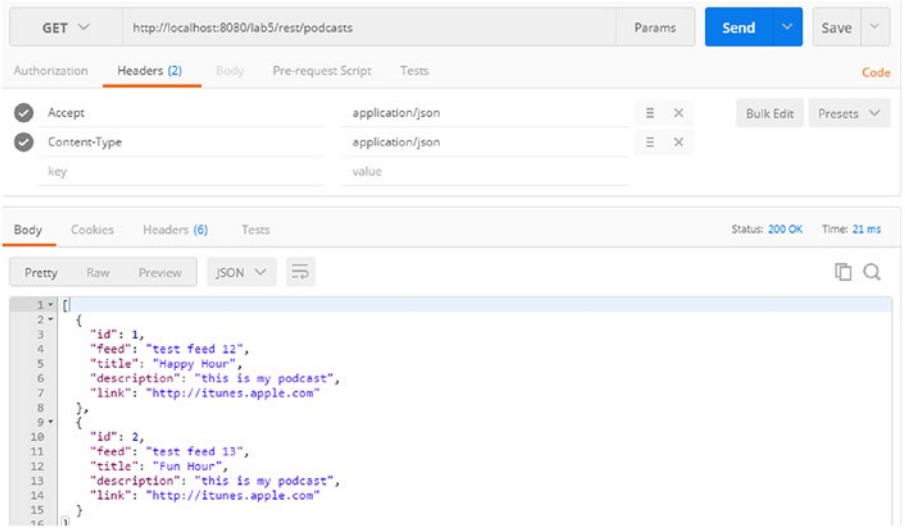


Figure 5-3. *All podcasts*

Now let us search podcasts by title, as shown in Figure 5-4.

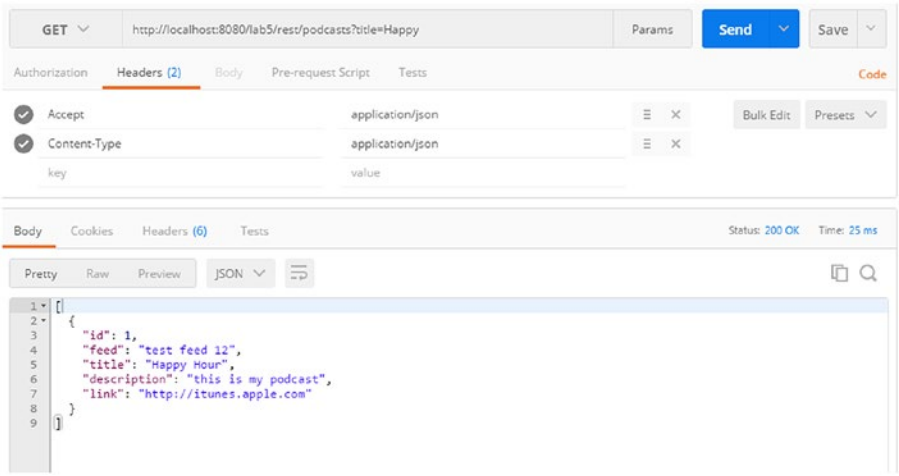


Figure 5-4. *Podcasts searched by title*

CHAPTER 6

API Platform and Data Handler

This chapter starts with API Platform Architecture and then gets into data handler pattern for integration of RESTful APIs with actual data sources within enterprise to make it more meaningful to the consumers through APIs.

API Platform Architecture

API Platforms are used by API Providers to realize APIs efficiently. We will review the following:

- Why do we need API Platform?
- What is an API Platform?
- Which capabilities does an API Platform have?
- How is API Platform organized? What is the architecture of API Platform?
- How does API architecture fit in the surrounding technical architecture of an Enterprise?

Why do we need API Platform?

It is certainly technically feasible to build APIs without any platform or framework. But why would you? For a moment, let's think about databases, which provide a platform for building applications. You could certainly build your application without a database and write your own data storage library. But we typically do not do that. We use an existing database as a platform. And this is the best practice for good reasons. It allows us to focus on building an application that serves the business case, because we can reuse existing, proven components and build the application quicker. The same augmentation applies to API platforms: API platforms allow us to focus on building APIs that consumers love, since we can reuse existing, proven API building blocks and build APIs quicker.

S. Patni, *Pro RESTful APIs*, DOI 10.1007/978-1-4842-2665-0_6

So what is an API Platform?

An API Platform consists of one of the following three components:

- API development platform
  - It offers tools to design and development of APIs quicker.
  - It offers building blocks, which are proven, reusable, and configurable.
- API runtime platform
  - This primarily executes API.
  - It serves API responses for incoming API requests of the consumers with non-functional properties like high throughput and low latency.
- API engagement platform
  - This platform allows API providers to manage their interaction with API consumers. It offers API Documentation, Credentials, and Rate plans for the consumers.

So which capabilities does the API platform have?

The following are the capabilities offered by three components of API Platform:

API Development Platform

API development platform offers a toolbox for API design and development targeted for API developers who works for API Providers. Toolbox contains API building blocks, which are proven, reusable, and configurable. When building APIs, certain functionality is needed over and over again. This can be accomplished by building blocks. Building blocks can be reused. Building blocks are tested so bugs are not there and these are configurable so they can be adopted for many purposes. The building blocks offered by API Development Platform span the following features at the minimum:

- Processing of HTTP requests and responses
- Header
- Query
- HTTP: status code
- Methods
- Security: IP-based access limitation, location-based access limitation, time-based access limitation, front-end authentication and authorization, OAuth, basic authorization, API key, back-end authentication and authorization (with LDAP, SAML)

- Front-end protocols: HTTP (REST), SOAP, RPC, RMI
- Data format transformation: XML to JSON and JSON to XML
- Structural transformation: XLST, XPATH
- Data integrity and protection: encryption
- Routing to one or more back ends
- Aggregation of multiple APIs and or multiple back ends
- Throttling to protect back-end rate limitation and throughput limitation
- Load balancing for incoming requests to the API platform and outgoing requests to the back ends
- Hooks for logging
- Hooks for analytics
- Monetization capabilities
- Language for implementing APIs: Java, JavaScript, etc. (Jersey, Restlet, Spring)
- IDE for API development with editor, debugger, and deployment tools: Eclipse, JDeveloper, NetBeans
- Language for designing APIs: YAML, RAML, etc.
- Design tools for creating API interface designs: RAML, Swagger, Blueprint
- Tools for generating documentation and API code skeletons based upon design: RAML, Swagger

API Runtime Platform: API Runtime platform primarily executes APIs. It enables the APIs to accept incoming requests from API consumers and serve responses.

- It should deliver non-functional properties like:
    - High availability, high security, high throughput
        - To meet above these properties platform offers:
            - Load balancing
            - Connection pooling
            - Caching
- It should also offer capabilities for Monitoring of API, Logging, and Analytics to check desired non-functional properties are met.

API Engagement Platform

API Engagement Platform is used by API providers to interact with its community of API consumers. API Providers use the following capabilities of API Engagement Platform:

- API management: configuration and reconfiguration of APIs without need for deployment
- API discovery: a mechanism for clients to obtain information about APIs.
- Consumer onboarding: app key generation, API Console
- Community management: blogs
- Documentation
- Version management
- Management of monetization and service-level SLAs

API Consumers use engagement platform for:

- Overview of API portfolio
- Documentation of APIs
- Possibility of trying API interactively
- Example source code for integration
    - Self service to get access to APIs
    - Client tooling, such as code generation for client

How is API Platform organized? What is architecture of API Platform?

Usually, APIs are not only deployed on the production system, but need to be deployed on different stages of increasing maturity. The stages are also sometimes called environments. Each of the stages has a specific purpose and is separated from the other stages to isolate potential errors.

- Simulation: used for playing with interface design, provides mocks or simulation of an API
- Development: used for development, which will eventually go to production
- Testing: used for manual black box testing and integration testing
- Pre-production: used as a practice for production and for acceptance testing
- Production: used as a real system for consumers

As shown in Figure 6-1, API Development Platform is used for Design and Development. API Runtime Platform is used for deployment. API Engagement Platform is used for publishing the API.

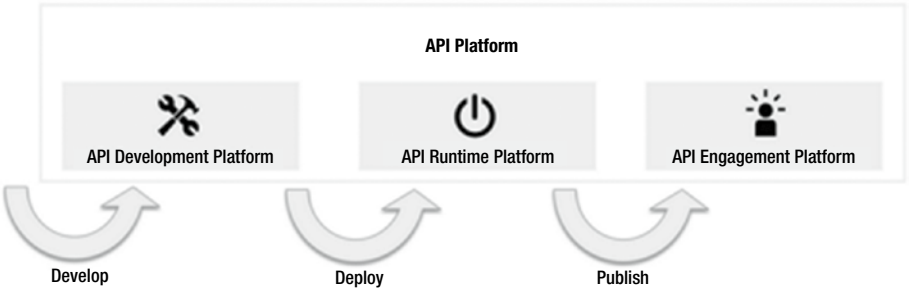


Figure 6-1. *API Platform architecture*

How does API architecture fit in surrounding technical architecture of an Enterprise?

API Platform is not isolated but it needs to be integrated in existing architecture in the enterprise. Firewall is used to improve security. Load balancers are used to improve performance, and are usually placed between the Internet and the API platform. IAM (Identity and Access Management) systems are for managing identity information and LDAP or Active Directory as shown in Figure 6-2.

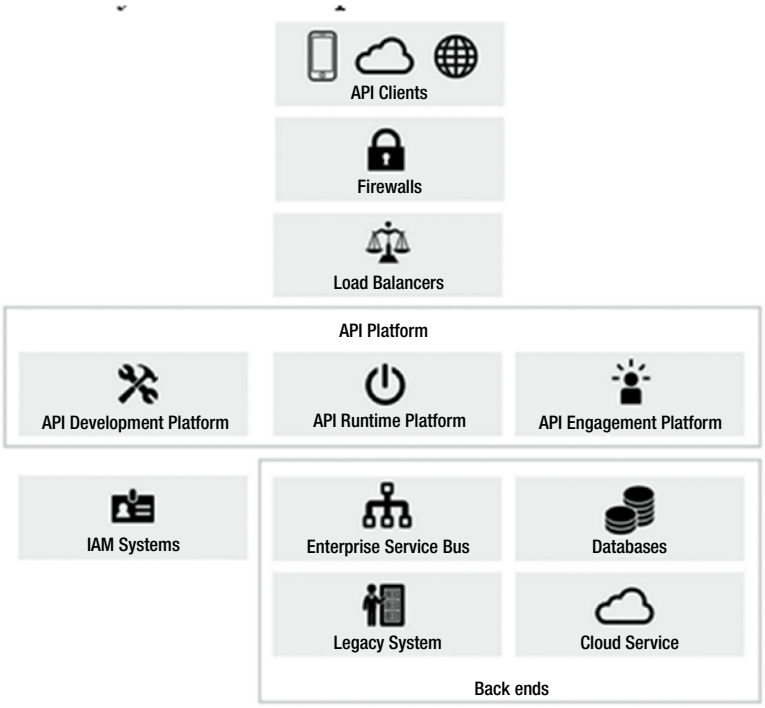


Figure 6-2. *API Architecture in Enterprise*

Back-end systems for providing the core functionality of the enterprise: Back-end systems form the heart of the enterprise data and services typically reside in the back-end system. Back ends may be databases, applications, enterprise service buses, web services using SOAP, message queues, and REST services.

Data Handler

As mentioned in previous section, we use an existing database as a platform. A Data Handler, a Data Access Object (DAO), or Command Query Responsibilities segmentation (CQRS) all provide an abstract interface to some type of database or any other persistence mechanism. Data Handler is a layer which handles data in the framework. Data Access Object is a design pattern used to implement the access from the database inside data handler. CQRS pattern, on the other hand, provides a mechanism to segment query and transational data in the data handler.

Data Access Object

By mapping application calls to the persistence layer, a DAO provides some specific data operations without exposing details of the database. The advantage of using data access objects is the relative simplicity and it provides separation between two important parts

of an application that can but should not know anything about each other, and which can be expected to evolve frequently and independently. Changing business logic can rely on the same DAO interface, while changes to persistence logic do not affect DAO clients as long as the interface remains correctly implemented. All details of storage are hidden from the rest of the application (see information hiding). Thus, possible changes to the persistence mechanism can be implemented by just modifying one DAO implementation while the rest of the application isn't affected. DAOs act as an intermediary between the application and the database. DAOs move data back and forth between objects and database records.

For accessing databases, there are different APIs available (for example, JPA, which will be used in class lab).

Command Query Responsibilities Segmentation - CQRS

New demands are being put on IT organizations every day to deliver agile, high-performance integrated mobile and web applications. In the meantime, the technology landscape is getting complex every day with the advent of new technologies like REST, NoSQL, and Cloud, while existing technologies like SOAP and SQL still rule everyday work. Rather than taking a religious side of the debate, NoSQL can successfully co-exist with SQL in this “polyglot” of data storage and formats. However, this integration also adds another layer of complexity both in architecture and implementation. We will talk about the following:

SQL Development Process

The application development lifecycle means changes to the database schema first, followed by the bindings, then internal schema mapping, and finally the SOAP or JSON services, and eventually the client code. This all costs the project time and money. It also means that the “code” (pick your language here) and the business logic would also need to be modified to handle the changes to the model. Figure 6-3 shows the traditional CRUD architecture.

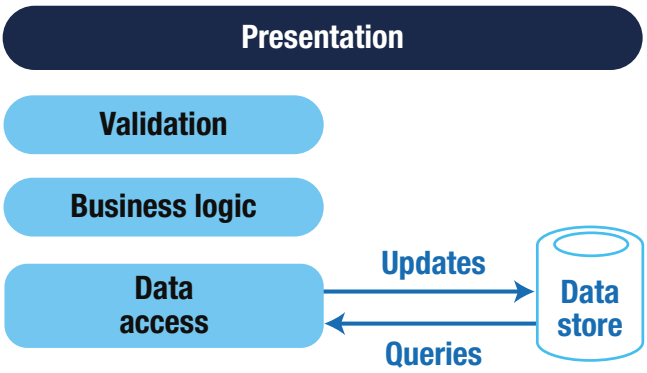


Figure 6-3. *Traditional CRUD architecture*

NoSQL Process

NoSQL is gaining supporters among many SQL shops for various reasons including low cost, the ability to handle unstructured data, scalability, and performance. The first thing database folks notice is that there is no schema. These document-style storage engines can handle huge volumes of structured, semi-structured, and unstructured data. The very nature of schema-less documents allows change to a document structure without having to go through the formal change management process (or data architect).

Do I have to choose between SQL and NoSQL?

The bottom line is both have their place and are suited for certain types of data—SQL for structured data and NoSQL for unstructured data. NoSQL databases are more scalable than SQL databases. So why not have the capability to mix and match this data depending on the application? This can be done by creating a single REST API across both SQL and NoSQL databases.

Why a single REST API?

The answer is simple—the new agile and mobile world demands this "mash-up" of data into a document-style JSON response.

Martin Fowler described the pattern called "CQRS" that is more relevant today in a "polyglot" of servers, data, services, and connections (Figure 6-4).

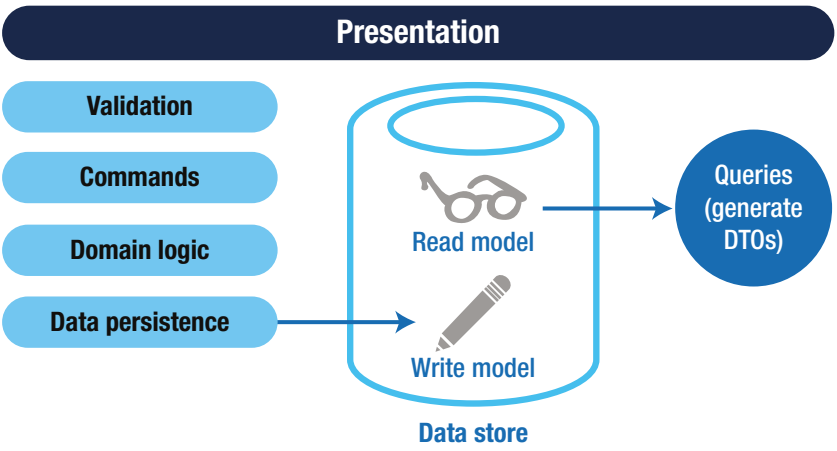


Figure 6-4. *Basic CQRS architecture*

In this design pattern, the REST API requests (GET) return documents from multiple sources (e.g., mash-ups). In the update process, the data is subject to business logic derivations, validations, event processing, and database transactions. This data may then be pushed back into the NoSQL using asynchronous events. The advantage of NoSQL databases over SQL for this purpose is that NoSQL has dynamic schema for unstructured data. Also, NoSQL databases are horizontally scalable, which means NoSQL databases are scaled by increasing the database servers in the pool of resources to reduce the load, whereas SQL databases are scaled by increasing horsepower of the server where the database is hosted. Figure 6-5 shows CQRS architecture with a separate read and write store. When you have a requirement of very, very large data volumes, you would choose separate stores.

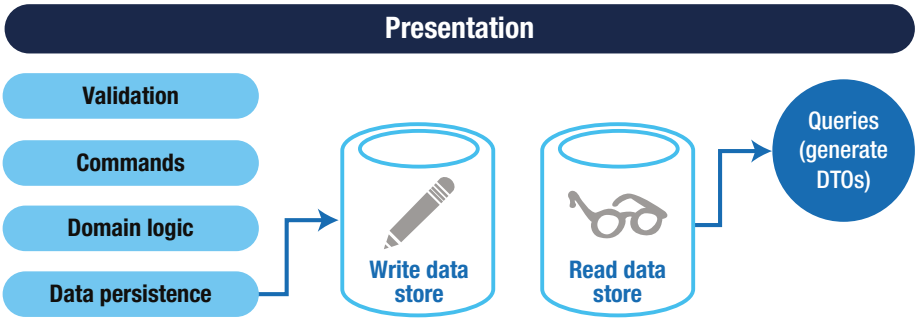


Figure 6-5. *CQRS architecture with separate read and write store*

FRAMEWORK - DATA HANDLER

This exercise will implement data handler or data access object for the podcast domain object using Java Persistence API (JPA) . The JPA is a Java specification for accessing, persisting, and managing data between Java objects/classes and a relational database. We will use our domain object podcast and implement CRUD operations using JPA in DAO.

```
Database Setup

Install MySql

mysql -u root -p
Password:

Use My SQL workbench if you do not like command-line interface
Create user rest_demo
    create user 'rest_demo'@'localhost' identified by 'rest_demo';
```

```
Grant priviledges

grant all privileges on *.* to 'rest_demo'@'localhost';
Create database
       create database rest_demo;
Create table
use rest_demo;

CREATE TABLE `podcasts`
(`id` bigint(20) NOT NULL AUTO_INCREMENT,
`title` varchar(145) NOT NULL,
`feed` varchar(145) NOT NULL,
`insertion_date` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP,
`description` varchar(500) DEFAULT NULL,
`link` varchar(145) DEFAULT NULL,
PRIMARY KEY (`id`), UNIQUE KEY `title_UNIQUE` (`title`)
)
```

POM.XML Updates

Table 6-1. *Maven project dependencies*

Project	Dependencies	Version	Description
javax.persistence	persistence-api	1.0	Java persistence API
org.eclipse.persistence	org.eclipse.persistence.jpa	2.1.1	Eclipse project for JPA

```
applicationContext.xml Updates

<bean id="podcastService"
          class="com.rest.service.PodcastServiceImpl" />
```

resources/META-INF/persistence.xml

The `persistence.xml` file is shown as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
     http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
     version="2.0" xmlns="http://java.sun.com/xml/ns/persistence">
     <persistence-unit name="Podcast" transaction-type=
     "RESOURCE_LOCAL">
          <class>com.rest.domain.Podcast</class>
          <properties>
               <property name="javax.persistence.jdbc.driver"
               value="com.mysql.jdbc.Driver" />
```

```
            <property name="javax.persistence.jdbc.url"
                 value="jdbc:mysql://localhost:3306/rest_demo" />
            <property name="javax.persistence.jdbc.user"
            value="rest_demo" />
            property name="javax.persistence.jdbc.password"
            value="rest_demo" />

        </properties>

    </persistence-unit>

</persistence>
```

Podcast Domain Object

Java Persistence API is a source to store business entities as relational entities. It shows how to define a Plain Object Oriented Java Object (POJO) as an entity and how to manage entities with relations.

```
package com.rest.domain;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

import javax.persistence.Table;

@Entity
@Table(name="rest_demo.Podcasts")
public class Podcast {
    // Maps a object property to a XML element derived from
       property name.
    private int id;
    String title = null;
    String feed = null ;
    String link = null ;
    String description = null;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public int getId() {
            return id;
    }
    public void setId(int id) {
            this.id = id;
     }
```

```
        public String getTitle() {
                return title;
        }
        public void setTitle(String title) {
                this.title = title;
        }

        public String getFeed() {
                return feed;
        }
        public void setFeed(String feed) {
                this.feed = feed;
        }
        public String getLink() {
                return link;
        }
        public void setLink(String link) {
                this.link = link;
        }
        public String getDescription() {
                return description;
        }
        public void setDescription(String description) {
                this.description = description;
        }

};
```

PodcastResource

Resource implementing CRUD operations:

```
package com.rest.resource;

import java.util.List;

import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.Response;
```

```
import org.springframework.beans.factory.annotation.Autowired;

import com.rest.domain.Podcast;
import com.rest.exception.NotFoundException;
import com.rest.service.PodcastService;

@Path("podcasts")
public class PodcastResource {

    @Autowired
    PodcastService podcastService;

    @POST
    @Consumes({"application/json"})
    @Produces({"application/json"})
    public void createPodcast(Podcast podcast) {

        podcastService.createPodcast(podcast);
    }

    @GET
    @Path("{id}")
    @Consumes({"application/json"})
    @Produces({"application/json"})
    public Podcast getPodcast(@PathParam("id") int id) throws
    NotFoundException {
       Podcast podcast = podcastService.getPodcast(id);
       Return podcast;
    }

    @GET
    @Path("/search")
    @Consumes({"application/json"})
    @Produces({"application/json"})
    public List<Podcast> getPodcasts(@QueryParam("title")
    String title) {
         List<Podcast> podcastList = podcastService.
         getPodcastByTitle(title);

         return podcastList;

    }

    @DELETE
    @Path("{id}")
    @Consumes({"application/json"})
    @Produces({"application/json"})
```

```
    public Podcast deletePodcast(@PathParam("id") int id) {
       Podcast current = podcastService.deletePodcast(id);
       return current;
    }

    @PUT
    @Path("{id}")
    @Consumes({"application/json"})
    @Produces({"application/json"})
    public Podcast updatePodcast(@PathParam("id") int id,
    Podcast update) {
       Podcast current = podcastService.updatePodcast(id, update);
       return current;
    }

    @GET
    @Consumes({"application/json"})
    @Produces({"application/json"})
    public List<Podcast> getAll() {
         List<Podcast> podcastList = podcastService.getAll();

         return podcastList;

    }
}
```

PodcastService

Podcast service interface:

```
package com.rest.service;

import java.util.List;

import com.rest.domain.Podcast;
import com.rest.exception.NotFoundException;

public interface PodcastService {
    public void createPodcast(Podcast podcast);
    public Podcast getPodcast(int id) throws NotFoundException;
    public Podcast deletePodcast(int id);
    public Podcast updatePodcast(int id, Podcast update);
    public List<Podcast> getAll();
     public List<Podcast> getPodcastByTitle(String title);
};
```

PodcastServiceImpl

Podcast service will use IoC to get an instance of `podcastDao` and implement any business logic here.

```
package com.rest.service;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import com.rest.dao.PodcastDAO;
import com.rest.domain.Podcast;
import com.rest.exception.NotFoundException;

public class PodcastServiceImpl implements PodcastService {

    @Autowired
    PodcastDAO podcastDao;

    public void createPodcast(Podcast podcast) {
            podcastDao.createPodcast(podcast);
    }

    public Podcast getPodcast(int id) throws NotFoundException {
            Podcast podcast = podcastDao.getPodcast(id);
            return podcast;
    }

    public Podcast deletePodcast(int id) {
        Podcast current = podcastDao.deletePodcast(id);
        return current;
    }

    public Podcast updatePodcast(int id, Podcast update) {
        Podcast current = podcastDao.updatePodcast(id, update);
        return current;
    }

    public List<Podcast> getAll() {
            List<Podcast> podcastList = podcastDao.getAll();
            return podcastList;
    }

    public List<Podcast> getPodcastByTitle(String title) {
            List<Podcast> podcastList = podcastDao.getPodcast
            ByTitle(title);
            return podcastList;
    }

}
```

PodcastDAO

Podcast DAO interface:

```
package com.rest.dao;

import java.util.List;

import com.rest.domain.Podcast;
import com.rest.exception.NotFoundException;

public interface PodcastDAO {
    public void createPodcast(Podcast podcast);
    public Podcast getPodcast(int id) throws NotFoundException;
    public Podcast deletePodcast(int id);
    public Podcast updatePodcast(int id, Podcast update);
    public List<Podcast> getAll();
    public List<Podcast> getPodcastByTitle(String title);
}
```

PodcastDAOImpl

`PodcastDAOImpl` object implements CRUD operations to support persistence using JPA. `EntityManagerFactory` is a factory class of `EntityManager`. It creates and manages multiple `EntityManager` instances. `EnitityManager` is an interface; it manages the persistence operations on objects. It works like factory for `Query` instance. `Query` interface is implemented by each JPA vendor to obtain relational objects that meet the criteria.

Below is `PodacstDAO` implementing CRUD operations and search by title on a podcast entity.

```
package com.rest.dao;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;

import javax.ws.rs.core.Response;

import com.rest.domain.Podcast;

import com.rest.exception.ErrorMessage;
import com.rest.exception.NotFoundException;
```

```
public class PodcastDAOImpl implements PodcastDAO {
      private static final String PERSISTENCE_UNIT_NAME = "Podcast";
     private static EntityManagerFactory factory;

    public void createPodcast(Podcast podcastnew) {
          factory = Persistence.createEntityManagerFactory
          (PERSISTENCE_UNIT_NAME);
          EntityManager em = factory.createEntityManager();
          // Read the existing entries and write to console
          Query q = em.createQuery("SELECT p FROM Podcast p");
          @SuppressWarnings("unchecked")
              List<Podcast> podcastList = q.getResultList();
          for (Podcast podcast : podcastList) {
               System.out.println(podcast.getTitle());
          }

          // Create new user
          em.getTransaction().begin();
          Podcast podcast = new Podcast();
          podcast.setTitle(podcastnew.getTitle());
          podcast.setDescription(podcastnew.getDescription());
          podcast.setFeed(podcastnew.getFeed());
          podcast.setLink(podcastnew.getLink());
          em.persist(podcast);
          em.getTransaction().commit();
          em.close();
    }

     public Podcast getPodcast(int id) throws NotFoundException {
              factory = Persistence.createEntityManagerFactory
              (PERSISTENCE_UNIT_NAME);
        EntityManager em = factory.createEntityManager();
     Podcast podcast = em.getReference(Podcast.class, id);
     if (podcast == null ) {
            ErrorMessage errorMessage = new ErrorMessage("1001",
            "Podcast not found!", "http://localhost:8080/lab6/
            error.jsp", Response.Status.NOT_FOUND);
                    throw new NotFoundException(errorMessage);
    }
    return podcast;
     }

    public Podcast deletePodcast(int id) {
            factory = Persistence.createEntityManagerFactory
            (PERSISTENCE_UNIT_NAME);
        EntityManager em = factory.createEntityManager();
        Podcast podcast = em.find(Podcast.class, id);
```

```
        em.getTransaction().begin();
        em.remove(podcast);
        em.getTransaction().commit();

        return podcast;
     }

    public Podcast updatePodcast(int id, Podcast update) {
            factory = Persistence.createEntityManagerFactory
            (PERSISTENCE_UNIT_NAME);
        EntityManager em = factory.createEntityManager();
        Podcast podcast = em.find(Podcast.class, id);

        em.getTransaction().begin();
        podcast.setTitle(update.getTitle());
        podcast.setDescription(update.getDescription());
        podcast.setFeed(update.getFeed());;
        podcast.setLink(update.getLink());;
        em.getTransaction().commit();
        return podcast;
    }

    // get all
            @SuppressWarnings("unchecked")
            public List<Podcast> getAll() {
                    factory = Persistence.createEntityManagerFactory
                    (PERSISTENCE_UNIT_NAME);
             EntityManager em = factory.createEntityManager();
                Query query = em.createQuery("SELECT e FROM
                Podcast e");
                return (List<Podcast>) query.getResultList();
              }

    // get all

            @SuppressWarnings("unchecked")
            public List<Podcast> getPodcastByTitle(String title)
            throws NotFoundException {
                    factory = Persistence.createEntityManagerFactory
                    (PERSISTENCE_UNIT_NAME);
             EntityManager em = factory.createEntityManager();

                    return (List<Podcast>) em.createQuery(
                        "SELECT p FROM Podcast p  WHERE p.title
                        LIKE :title")
                        .setParameter("title", title)
                        .setMaxResults(10)
                        .getResultList();

            }
```

Now let's look into podcasts in MySQL and fetched by REST in POSTMAN as shown in Figure 6-6 and Figure 6-7.

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 22
Server version: 5.7.16 MySQL Community Server (GPL)

Copyright (c) 2000, 2016, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use rest_demo
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> select * from Podcasts;
+----+---------+--------+---------+---------------------+-------------+--------+
| id | login   | title  | feed    | insertion_date      | description | link   |
+----+---------+--------+---------+---------------------+-------------+--------+
|  1 | sapatni | test 1 | bug.gif | 2016-11-20 16:21:58 | desc 1      | link 1 |
|  4 | sapatni | test 7 | feed 7  | 2016-12-18 18:07:10 | desc 1      | link 1 |
+----+---------+--------+---------+---------------------+-------------+--------+
2 rows in set (0.00 sec)
```

Figure 6-6. *Podcasts in MySQL database*

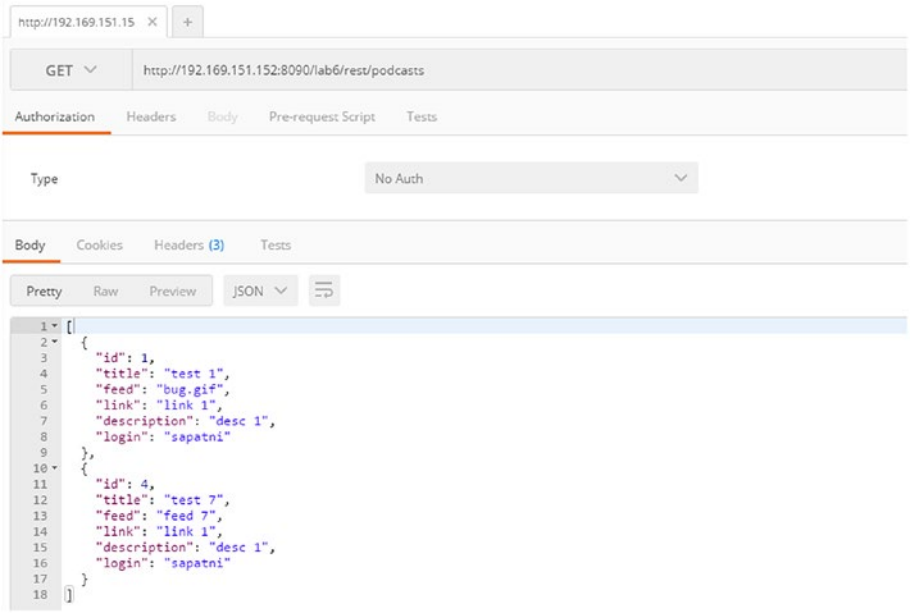


Figure 6-7. *Podcasts fetched from MySQL database*

Wrapping Up

In this chapter we started with API Platform Architecture and then got into data handler pattern for integration of RESTful APIs with actual data sources. In the exercise we demonstrated implementation of Data Handler using JPA.

■ **Note** If you have problem in running JPA in eclipse due to Eclipse persitence provider you can build it outside eclipse using maven and deploy in tomcat.

CHAPTER 7

API Management and API Client

In this chapter we will start with Façade and review API Management requirements/ solutions available. Then we will continue with the framework and build demo client calls of RESTful APIs for the podcast application, followed by how clients need to be supported by Cross Origin Resource Sharing (CORS).

Façade

In this section we will first review the Façade design pattern and then in the second part we will get into details about how Façade is applied to the APIs.

Façade Pattern

Before we discuss the Façade Pattern, let's consider what a façade is in the real world. The most obvious example is that of buildings, which all have an exterior to protect and decorate, hiding the internal workings of the interior. This exterior is the façade.

Now we can get closer to APIs by considering operating systems. Just like in buildings, an operating system provides an exterior shell to the interior functionality of a computer. This simplified interface makes an OS easier to use and protects the core from clumsy users.

This is where the definition of the Façade Pattern in On Design Patterns (Gamme, et al.) comes in handy:

> *Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.*

Consider Figure 7-1; you can see how the Façade Pattern puts an intermediate layer between the packages of the application and any client that wants to interact with them.

S. Patni, *Pro RESTful APIs*, DOI 10.1007/978-1-4842-2665-0_7

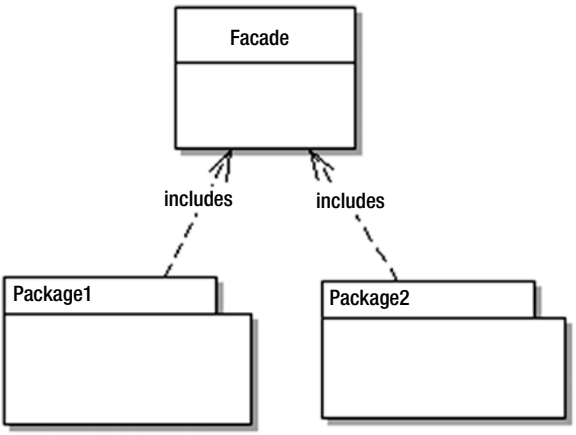


Figure 7-1. *Façade Pattern*

API Facade

Like all implementations of the Façade Pattern, an API Façade is a simple interface to a complex problem. Figure 7-2 shows internal subsystems in an enterprise. As shown, each internal subsystem is complex in itself: for example, JDBC hides the inner workings of database connectivity.

Figure 7-2. *Internal subsystems*

Figure 7-3 shows an API Façade layer on the top of internal subsystems of the enterprise, providing a unified interface to apps.

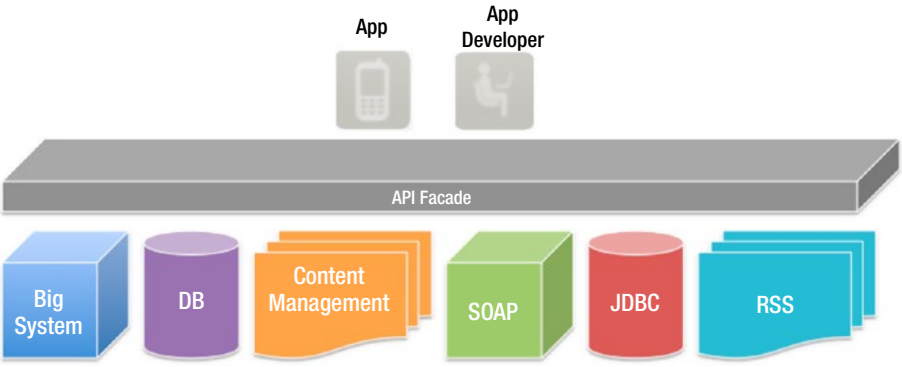


Figure 7-3. *API Facade*

Implementing an API façade pattern involves three basic steps.

1. Design the API: identify the URLs, request parameters and responses, payloads, headers, query parameters, and so on.
2. Implement the design with mock data. App developers can then test the API before the API is connected to internal subsystems, with all the complications that entails.
3. Connect the façade with the internal systems to create the live API.

Figure 7-4 shows these layers.

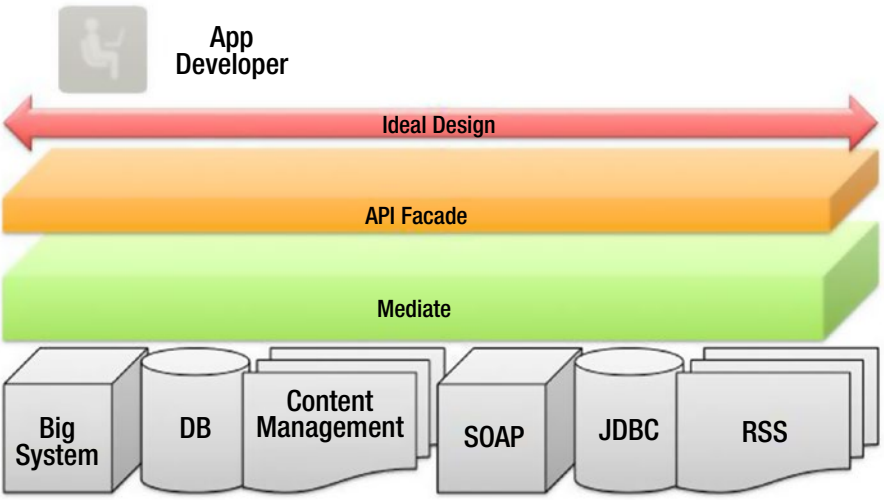


Figure 7-4. *API Façade*

API Management

An API management tool provide the means to expose your API to external developers in an easy and affordable manner.

Here are the features of an API management service:

- Documentation
- Analytics and statistics
- Deployment
- Developer engagement
- Sandbox environment
- Traffic management and caching abilities
- Security
- Availability
- Monetization
- API lifecycle management
- API management vendors implement their solution in three different ways:
    - Proxy. All traffic goes through the API management tool, which is placed as a layer between the application and users.
    - Agents. These are plug-ins for servers. They do not intercept API calls like proxies.
    - Hybrid. This approach picks features of proxies and agents, and integrates them. You can then pick which features you need.

API Life Cycle

The default API life cycle has the following stages:

- ANALYSIS: The API is analyzed and mock responses are created for a limited set of consumers to try out the API and provide feedback. It's also analyzed for monetization, as discussed in the following section.
- BEING CREATED/DEVELOPMENT: The API is being created: designed, developed, and secured. The API metadata is saved, but it is not visible yet, nor deployed.
- PUBLISHED/OPERATIONS: The API is visible and eventually published and is now in the maintenance stage, where it is scaled and monitored.

In addition, there are two more stages:

- DEPRECATED: The API is still deployed (available at runtime to existing users), but is not visible to new users. An API is automatically deprecated when a new version is published.
- RETIRED: The API is unpublished and deleted.

These are discussed in the next section.
Figure 7-5 shows an API life cycle.

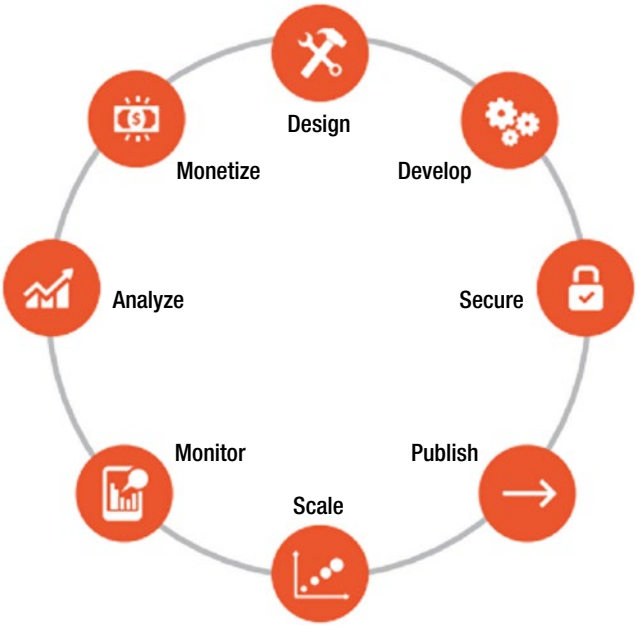


Figure 7-5. *API life cycle*

API Retirement

As old age comes we get to retire, and the same is true with APIs. With time and due to the following reasons, API is retired or deprecated.

- Lack of partner or third-party developer innovation
- Losing market share due to expose of data by API
- Changes in technology stack e.g., REST replacing SOAP
- Security concern: making public API private due to security requirement of the information or data exposed by API
- Versioning: most common reason due to functionality changes

Some of the examples of API retirement are Netflix, Google Earth, and Twitter V1.0, etc.

API Monetization

Digital assets or services provide real value to customers, partners, and end users and hence they should be a source of revenue for your company, as well as an important part of your business model.

There are three business models for monetizing APIs:

- The revenue share model, where the API consumer gets paid for the incremental business they trigger for the API provider.
- The fee-based model, where the API consumer pays the provider for API usage.
- The third and final business model is freemium. Freemium models can be based on a variety of factors such as volume, time, or some combination; they can be implemented as standalone or hybrid models (in conjunction with the revenue share or fee-based).

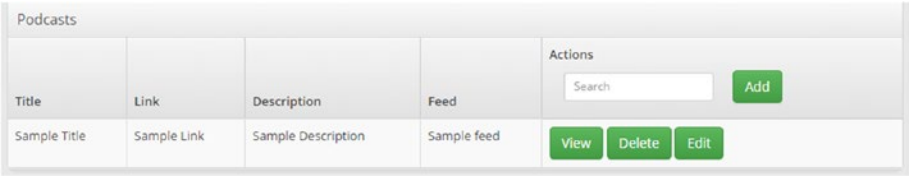
EXERCISE - API CLIENT AND CORS

API CLIENT

When you publish a REST API, you also provide a demo client application implementing API operation to show the use of API.

The client could be any of JavaScript, AJAX, JQuery or AngularJS, or even a native mobile app in Objective-C for iOS.

In this excercise we will use the style sheets of Bootstrap (Base Admin and AngularJS request) to get data from the server and bind that in the UI.



This UI can implement CRUD operations as follows:

Podcasts collection: Will load podcasts using “GET podcasts”

Podcasts collection: Add button will invoke a form to capture details which can be posted by “POST podcasts” and added to collection.

Podcast: View will invoke details of podcasts using “GET podcasts/{id}”.

Podcast: Delete will delete podcast from collection “DELETE podcasts/{id}”.

Podcast: Edit will invoke a form with current values and updates can be posted using “PUT podcasts/{id}”.

Get the bootstrap css and podcast.html from source code folder of appress site and integrate these AngularJS calls in podcast.html to test demo app.

GET ALL PODCASTS

```
var app = angular.module('app', []);

angular.module('app').controller("PodcastController",['$scope',
'$http', '$window', function($scope, $http, $window){
   $scope.podcast = {};

   $scope.getPodcasts = function() {
           url = "http://localhost:8080/lab7/rest/podcasts";
           $http({method: 'GET', url: url}).
                 success(function(data, status, headers,
                 config) {
                      $scope.podcastsList = data;
                 }).
                 error(function(data, status, headers,
                 config) {
                       $scope.apps = data || "Request
                       failed";
                       $scope.status = status;
                 });
       };
```

VIEW PODCAST

```
   $scope.viewPodcast = function(id) {

                 url = "http://localhost:8080/lab7/rest/
                 podcasts/" + id ;
                 console.log(url);
                 $http({method: 'GET', url: url}).
                 success(function(data, status, headers,
                 config) {
                         $scope.podcast = data;
                 }).
                 error(function(data, status, headers,
                 config) {
                         $scope.apps = data || "Request
                         failed";
                         $scope.status = status;
                 });

    }
```

DELETE PODCAST

```
$scope.deletePodcast = function(id) {

        url = "http://localhost:8080/lab7/rest/
        podcasts/" + id ;
        console.log(url);
        $http({method: 'DELETE', url: url}).
        success(function(data, status, headers, config) {
               $scope.podcast = data;
        }).
        error(function(data, status, headers, config) {
               $scope.apps = data || "Request failed";
               $scope.status = status;
        });

        $window.location.reload();
        }

$scope.submitForm = function(){
        $http({
          method  : 'POST',
          url     : 'http://localhost:8080/lab7/rest/
                    podcasts',
          data    : $scope.podcast, //forms podcast
                    object
          headers : {'Content-Type': 'application/json'}
         }).
         success(function(data, status, headers, config) {
                       $scope.podcast = data;
               }).
               error(function(data, status, headers,
               config) {
                       $scope.apps = data || "Request
                       failed";
                       $scope.status = status;
               });
        $window.location.reload();
}
```

UPDATE PODCAST

```
$scope.updatePodcast = function(id){
        $http({
          method  : 'PUT',
          url     : 'http://localhost:8080/lab7/rest/
                    podcasts/' + id,
```

```
                data    : $scope.podcast, //forms podcast
                          object
                headers : {'Content-Type': 'application/json'}
               }).
               success(function(data, status, headers, config) {
                           $scope.podcast = data;
                     }).
                     error(function(data, status, headers,
                     config) {
                           $scope.apps = data || "Request
                           failed";
                           $scope.status = status;
                     });
               $window.location.reload();
    }
```

SEARCH PODCAST

```
    $scope.searchPodcast = function(){
            url = "http://localhost:8080/lab7/rest/podcasts/
                  search?title=" + $scope.searchVal;
            console.log(url);
            $http({method: 'GET', url: url}).
            success(function(data, status, headers, config) {
                   $scope.podcastsList = data;
            }).
            error(function(data, status, headers, config) {
                   $scope.apps = data || "Request failed";
                   $scope.status = status;
            });

    }

}]);
```

Cross-Origin Resource Sharing (CORS)

"Cross-Origin Resource Sharing" (CORS) is a mechanism that allows JavaScript on a web page to make XMLHttpRequests to another domain, not the domain the JavaScript originated from. Such "cross-domain" requests would otherwise be forbidden by web browsers, per the same origin security policy. CORS defines a way in which the browser and the server can interact to determine whether or not to allow the cross-origin request. It is more useful than only allowing same-origin requests, but it is more secure than simply allowing all such cross-origin requests.The Cross-Origin Resource Sharing standard works by adding new HTTP headers that allow servers to describe the set of origins that are permitted to read that information using a web browser.

How to implement CORS?

The first way is by using the header method of the javax.ws.rs.core.Response.

```
return Response.ok() //200
.entity(podcasts)
.header("Access-Control-Allow-Origin", "*")
.header("Access-Control-Allow-Methods", "GET, POST, DELETE, PUT")
.allow("OPTIONS").build();
```

Another way to add the headers to the response is by using Jersey filters, which can modify inbound and outbound requests and responses, including modification of headers, entity, and other request/response parameters.

I think this is the better way to do it, especially if you want to expose the same HTTP headers in the response for all the resources of the API—this is a sort of a cross-cutting concern capability powered by Jersey filters.

```
package com.rest.filter;

import java.io.IOException;

import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerResponseContext;
import javax.ws.rs.container.ContainerResponseFilter;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.ext.Provider;

@Provider
// Marks an implementation of an extension interface that should be
discoverable by
// JAX-RS runtime during a provider scanning phase.

// Filter intercepts incoming requests and add value
public class CORSResponseFilter implements ContainerResponseFilter {

 public void filter(ContainerRequestContext requestContext,
ContainerResponseContext responseContext) throws IOException {

 MultivaluedMap<String, Object> headers = responseContext.getHeaders();

 headers.add("Access-Control-Allow-Origin", "*");
//headers.add("Access-Control-Allow-Origin", "http://ucsc.com");
//allows CORS requests only coming from appress.org
// alternatively you can maintain list of origin allowed and get orgin
from request and check in list
 headers.add("Access-Control-Allow-Methods", "GET, POST, DELETE, PUT");
 }
}
```

CHAPTER 8

API Security and Caching

In this chapter we will start with the review of the OAuth 2 standard for securing RESTful APIs and do an exercise on implementing basic Spring security. We will then review caching concepts.

API Security - OAuth 2

OAuth 2 is a standard for delegating authorization for accessing resources by HTTP

With OAuth, we can give access rights to the mobile apps without giving a password. Instead, a token is handed over to the application. A token repesents access rights for the subset of data for a short time frame. Please refer to https://oauth.net/2/ for general information about OAuth 2.

To obtain a token, the user first logs onto the web site of the OAuth server. The generated token can be an authorization code, access token, or refresh code. OAuth is used under the hood of a number of modern clouds.

List of OAuth Providers: https://en.wikipedia.org/wiki/List_of_OAuth_providers

OAuth is specified and standardized by IETF in RFC6749 http://tools.ietf.org/html/rfc6749. OAuth 1 is outdated.

There are two terms: authentication and authorization.

- Authentication is a concept that answers the question: Who are you?
- Authorization is a concept that answers the question: What are you allowed to do?

Roles

OAuth2 defines four roles:

- Resource Owner: generally yourself
- Resource Server: server hosting protected data (for example Google hosting your profile and personal information)

S. Patni, *Pro RESTful APIs*, DOI 10.1007/978-1-4842-2665-0_8

- Client: application requesting access to a resource server (it can be your PHP web site, a JavaScript application, or a mobile application)
- Authorization Server: server issuing an access token to the client. This token will be used for the client to request the resource server. This server can be the same as the authorization server (the same physical server and the same application), and it is often the case.

Tokens

Tokens are random strings generated by the authorization server and are issued when the client requests them.

There are two types of tokens:

- Access Token: This is the most important because it allows the user data from being accessed by a third-party application. This token is sent by the client as a parameter or as a header in the request to the resource server. It has a limited lifetime, which is defined by the authorization server. It must be kept confidential as soon as possible, but we will see that this is not always possible, especially when the client is a web browser that sends requests to the resource server via JavaScript. In general access, a token will be designed to be opaque to the client, but when it's used as a user authentication, the client will be needed to be able to derive some information from the token.
- Refresh Token: This token is issued with the access token but, unlike the latter, it is not sent in each request from the client to the resource server. It merely serves to be sent to the authorization server for renewing the access token when it has expired. For security reasons, it is not always possible to obtain this token. We will see later in what circumstances.

Figure 8-1 shows OAuth based interactions.

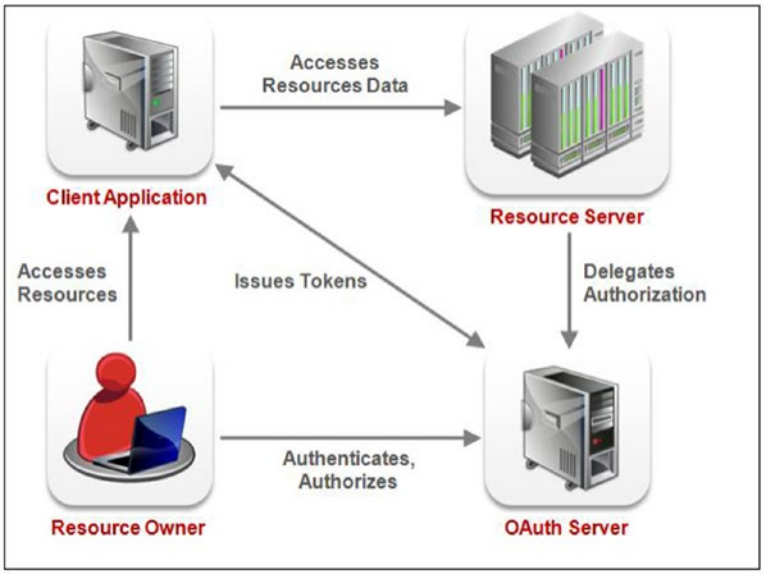


Figure 8-1. *OAuth-based interactions*

Register as a client

Since you want to retrieve data from a resource server using OAuth2, you have to register as a client of the authorization server.

Each provider is free to allow this by the method of his choice. The protocol only defines the parameters that must be specified by the client and those to be returned by the authorization server.

Here are the parameters (they may differ depending on the providers):

Client registration

Application Name: the application name

Redirect URLs: URLs of the client for receiving authorization code and access token

Grant Type(s): authorization types that will be used by the client

JavaScript Origin (optional): the hostname that will be allowed to request the resource server via `XMLHttpRequest`

Authorization server response

Client Id: unique random string

Client Secret: secret key that must be kept confidential

More information: RFC 6749 — Client Registration

Authorization grant types

OAuth2 defines four grant types depending on the location and the nature of the client involved in obtaining an access token:

Authorization Code Grant

We will review authorization code grant and its flow in this section. This type of grant is used to sign on into Google and Facebook.

When should it be used?

It should be used when the client is a web server or web site. It allows you to obtain a long-lived access token since it can be renewed with a refresh token (if the authorization server enables it).

Example:

Resource Owner: you
Resource Server: a Google server
Client: any web site
Authorization Server: a Google server

Scenario:

1. A web site wants to obtain information about your Google profile.
2. You are redirected by the client (the web site) to the authorization server (Google).
3. If you authorize access, the authorization server sends an authorization code to the client (the web site) in the callback response.
4. Then, this code is exchanged against an access token between the client and the authorization server.
5. The web site is now able to use this access token to query the resource server (Google again) and retrieve your profile data.

You never see the access token; it will be stored by the web site (in session, for example). Google also sends other information with the access token, such as the token lifetime and eventually a refresh token.

Sequence diagram:

Figure 8-2 shows sequence diagram for authorization code grant flow.

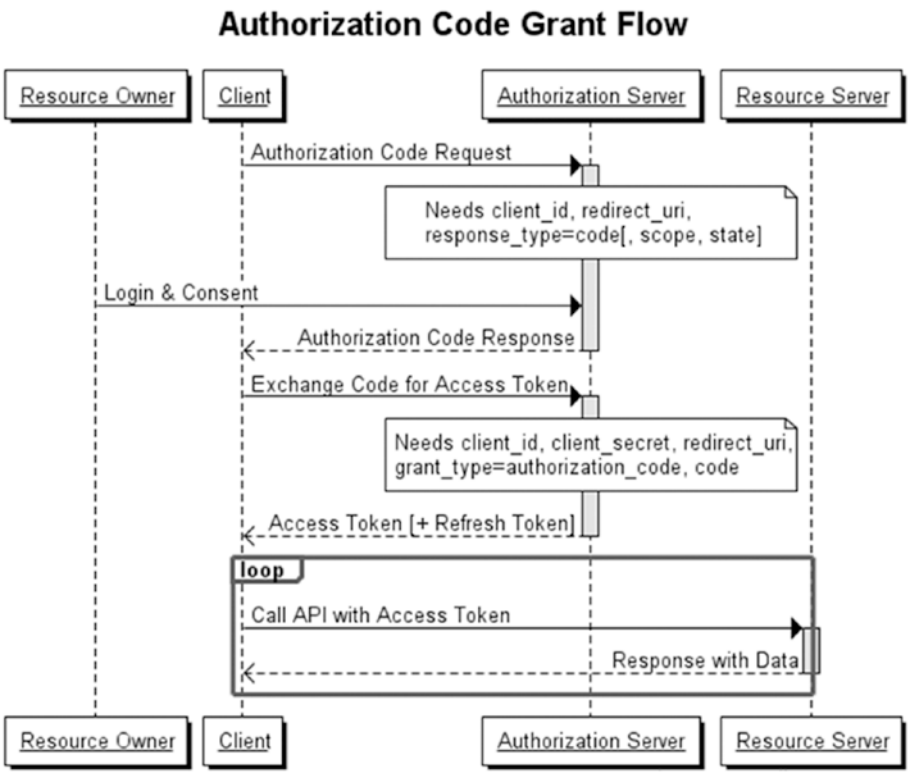


Figure 8-2. *Authorization code grant flow*

Implicit Grant Flow

We will review implicit grant and its flow in this section.

When should it be used?

It is typically used when the client is running in a browser using a scripting language such as JavaScript. This grant type does not allow the issuance of a refresh token.

Example:

Resource Owner: you
Resource Server: a Facebook server
Client: a web site using AngularJS, for example
Authorization Server: a Facebook server

Scenario:

1. The client (AngularJS) wants to obtain information about your Facebook profile.
2. You are redirected by the browser to the authorization server (Facebook).
3. If you authorize access, the authorization server redirects you to the web site with the access token in the URI fragment (not sent to the web server). Example of callback: `http://example.com/oauthcallback#access_token=MzJmNDc3M2VjMmQzN`.
4. This access token can now be retrieved and used by the client (AngularJS) to query the resource server (Facebook). Example of query: `https://graph.facebook.com/me?access_token=MzJmNDc3M2VjMmQz`

Maybe you wonder how the client can make a call to the Facebook API with JavaScript without being blocked because of the Same Origin Policy? Well, this cross-domain request is possible because Facebook authorizes it thanks to a header called Access-Control-Allow-Origin present in the response.

■ **Note** This type of authorization should only be used if no other type of authorization is available. Indeed, it is the least secure because the access token is exposed (and therefore vulnerable) on the client side.

Sequence diagram:

Figure 8-3 shows the sequence diagram for implicit grant flow.

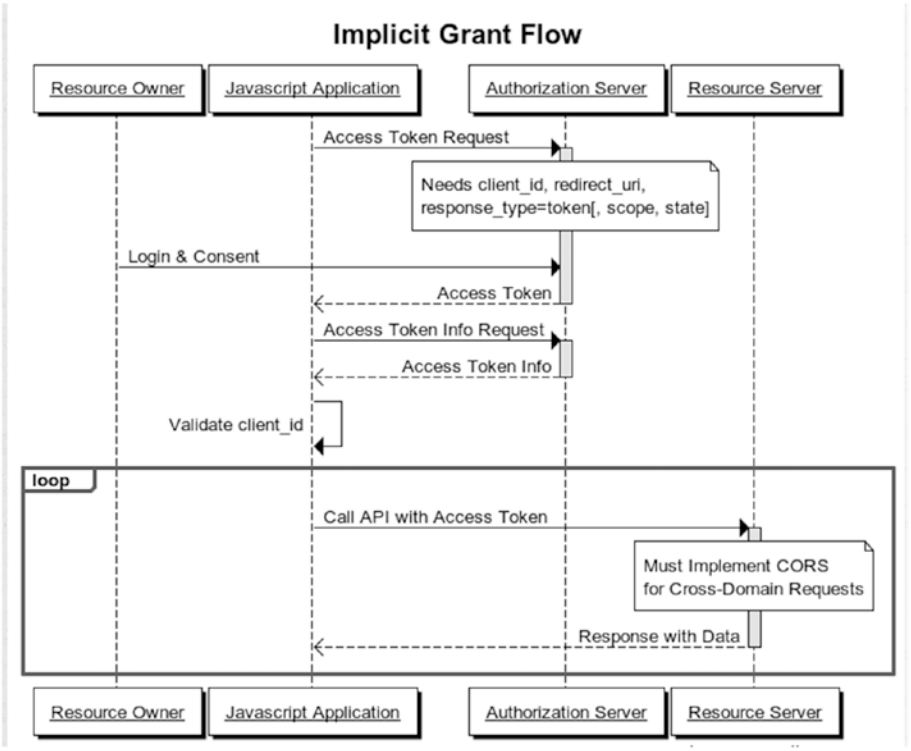


Figure 8-3

Resource Owner Password Credentials Grant

We will review resource owner password credentials grant and its flow in this section.

When should it be used?

With this type of authorization, the credentials (and thus the password) are sent to the client and then to the authorization server. It is therefore imperative that there is absolute trust between these two entities. It is mainly used when the client has been developed by the same authority as the authorization server. For example, we could imagine a web site named "example.com" seeking access to protected resources of its own subdomain "api.example.com". The user would not be surprised to type his login/password on the site "example.com" since his account was created on it.

Example:

Resource Owner: you have an account on the acme.com web site of the Acme company
Resource Server: Acme company exposes its API at api.acme.com
Client: acme.com web site from Acme company
Authorization Server: an Acme server

Scenario:

1. The Acme company, doing things well, thought to make available a RESTful API to third-party applications.
2. This company thinks it would be convenient to use its own API to avoid reinventing the wheel.
3. The company needs an access token to call the methods of its own API.
4. For this, the company asks you to enter your login credentials via a standard HTML form as you normally would.
5. The server-side application (web site "acme.com") will exchange your credentials against an access token from the authorization server (if your credentials are valid, of course).
6. This application can now use the access token to query its own resource server (api.acme.com).

Sequence diagram:

Figure 8-4 shows the sequence diagram for this flow.

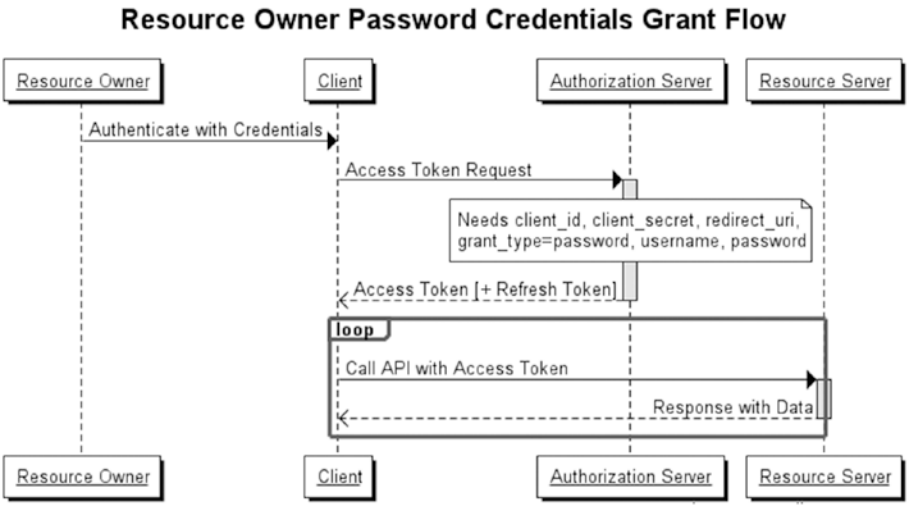


Figure 8-4. *Resource owner password credentials grant flow*

Client Credentials Grant

This type of authorization is used when the client is himself the resource owner. There is no authorization to obtain from the end user.

Example:

Resource Owner: any web site
Resource Server: Google Cloud Storage
Client: the resource owner
Authorization Server: a Google server

Scenario:

1. A web site stores its files of any kind on Google Cloud Storage.
2. The web site must go through the Google API to retrieve or modify files and must authenticate with the authorization server.
3. Once authenticated, the web site obtains an access token that can now be used for querying the resource server (Google Cloud Storage).

Here, the end user does not have to give his authorization for accessing the resource server.

Sequence diagram:

Figure 8-5 shows the sequence diagram for this flow.

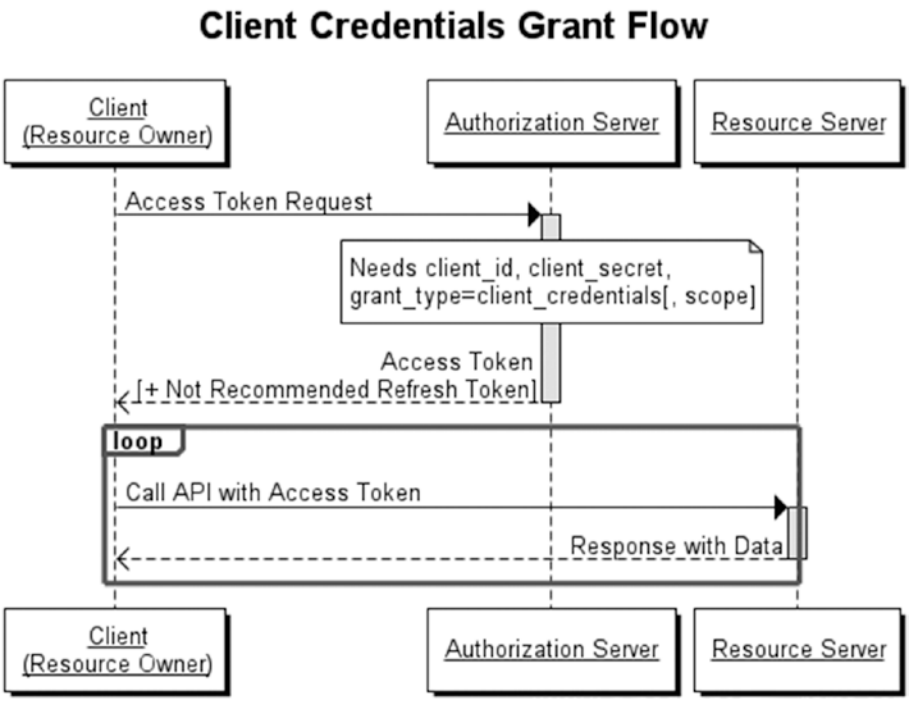


Figure 8-5. *Client credentials grant flow*

Caching

In this section we will review caching framework. Starting with caching solutions available at the services layer in the framework, we will review HTTP caching implemented in JAX-RS resources and the cover caching at the client. Using caching technology at strategic points across the multi-tier model can help reduce the number of back-and-forth communications. Furthermore, although cache repositories require memory and CPU resources, using caches can nonetheless lead to overall performance gains by reducing the number of expensive operations (such as database accesses and web page executions). However, ensuring that caches retain fresh content and invalidate stale data is a challenge, and keeping multiple caches in sync, in clustered environments, is even more so. In object caching by storing frequently accessed or expensive-to-create objects in memory, object caching eliminates the need to repeatedly create and load data. It avoids the expensive reacquisition of objects by not releasing the objects immediately after their use and, instead, the objects are stored in memory and reused for any subsequent client requests. HTTP and Web cache is used to reduce latency. It takes less time for it to get the representation and display it. It reduces load on the web site servers and reduces bandwidth needs and cost. It benefits the user, the service provider, and the web site owner. Figure 8-6 show caches used in the framework.

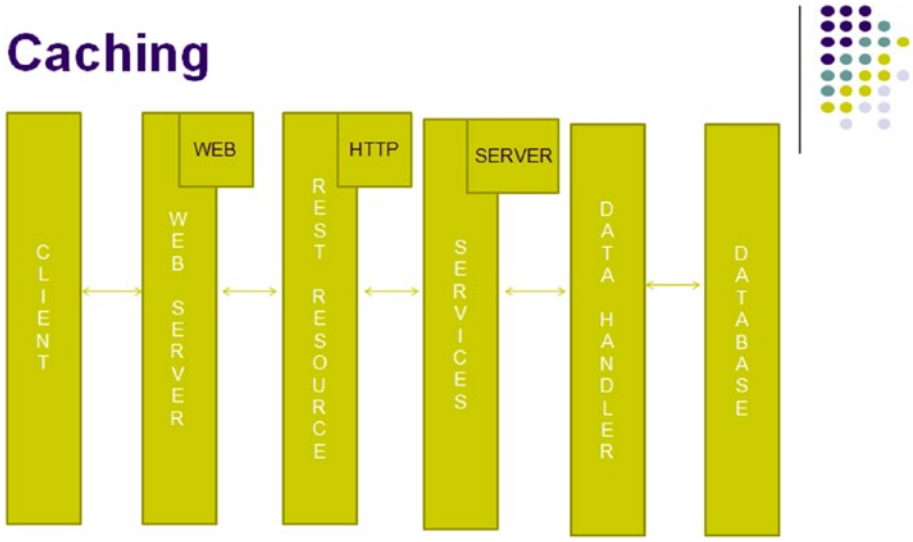


Figure 8-6. *Caching*

Server Caching

Multi-tier architectures help make complex enterprise applications manageable and scalable. Yet, as the number of servers and tiers increases, so does the communication among them, which can degrade overall application performance. In most of the web applications, data is retrieved from the database. The database operation is expensive and time-consuming. Present-day web applications are data-intensive and first response time is the basic criteria for success. If the web application is frequently accessing the database for each request, then its performance will be slow. In the Enterprise application we can cache objects by object caching. It allows applications to share objects across requests and users, and coordinates the objects' life cycles across processes. There are several Open Source Caching Frameworks for Java such as JBoss Cache, OSCache, Java Caching System, and EhCache. The drawbacks of using direct methods such as Java Hash map, Hash table, and JNDI are overcome by these frameworks.

HTTP Caching

HTTP caching is an easy and often ignored option to speed up web applications. It's standardized and well-implemented in all modern browsers and results in a lower latency app and improved responsiveness.

Time-based cached header

In HTTP 1.1 the `Cache-Control` header specifies the resource caching behavior as well as the max age the resource can be cached.

Here is a list of all the available `Cache-Control` tokens and their meaning:

- `private`: only clients (mostly the browser) and no one else in the chain (like a proxy) should cache this
- `public`: any entity in the chain can cache this
- `no-cache`: should not be cached anyway
- `no-store`: can be cached but should not be stored on disk (most browsers will hold the resources in memory until they are quit)
- `no-transform`: the resource should not be modified (for example, shrink image by proxy)
- `max-age`: how long the resource is valid (measured in seconds)
- `s-maxage`: same as max-age but this value is just for non-clients

In a JAX-RS method a `Response` object can be returned. The `Cache-Control` header can also be set on the `Response` object by using the `CacheControl` class.

Methods are provided for all of the available `Cache-Control` header tokens.

```
@Path("/podcasts/{id}")
@GET
public Response getPodcast(@PathParam("id") int id)  {
  Prodcast podcast = podcastDB.get(id);
  CacheControl cc = new CacheControl();
  cc.setMaxAge(86400);
  cc.setPrivate(true);
  ResponseBuilder builder = Response.ok(podcast);
  builder.cacheControl(cc);
  return builder.build();
}
```

Conditional cache headers

Conditional requests are those where the browser can ask the server if it has an updated copy of the resource. The browser will send one or both of the `ETag` and `If-Modified-Since` headers about the cached resource it holds. The server can then determine whether updated content should be returned or the browser's copy is the most recent.

```
@Path("/podcasts/{id}")
@GET
public Response getPodcast(@PathParam("id") long id, @Context Request request){
  Podcast podcast = podcastDB.get(id);
  CacheControl cc = new CacheControl();
  cc.setMaxAge(86400);
  EntityTag etag = new EntityTag(Integer.toString(podcast.hashCode()));
  ResponseBuilder builder = request.evaluatePreconditions(etag);
```

```
  // podcast did change -> serve updated content
  if(builder == null){
     builder = Response.ok(podcast);
   builder.tag(etag);
  }
  builder.cacheControl(cc);
  return builder.build();
}
```

A `ResponseBuilder` is automatically constructed by calling evaluate preconditions on the request. If the builder is null, the resource is out of date and needs to be sent back in the response. Otherwise, the preconditions indicate that the client has the latest version of the resource and the 304 Not Modified status code will be automatically assigned.

Web Caching

We can do web caching. The web cache sits between one or more web servers (also known as origin servers) and a client or many clients, and watches requests come by, saving copies of the responses —like HTML pages, images, and files (collectively known as representations)— for itself. If there is another request for the same URL, it can use the response that it has, instead of asking the origin server for it again.

EXERCISE - BASIC SECURITY

To secure the REST services with basic authentication, the following Spring security libraries are necessary in the classpath. Please include these in the `pom.xml`:

```
<dependency>
          <groupId>org.springframework.security</groupId>
          <artifactId>spring-security-core</artifactId>
          <version>4.0.4.RELEASE</version>
</dependency>
<dependency>
          <groupId>org.springframework.security</groupId>
          <artifactId>spring-security-web</artifactId>
          <version>4.0.4.RELEASE</version>
</dependency>
<dependency>
          <groupId>org.springframework.security</groupId>
           <artifactId>spring-security-config</artifactId>
           <version>4.0.4.RELEASE</version>
</dependency>
```

Spring Security Context

Create `security-applicationContext.xml` in `src/main/resources/spring`.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/security"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:security="http://www.springframework.org/schema/security"
    xsi:schemaLocation="
     http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans.xsd

        http://www.springframework.org/schema/security
        http://www.springframework.org/schema/security/
        spring-security.xsd">

 <!-- Stateless RESTful services use BASIC authentication -->
    <security:http create-session="stateless">
        <security:intercept-url pattern="/rest/**"
        access="hasRole('ROLE_REST_DEMO')"/>
        <security:http-basic/>
    </security:http>
    <security:authentication-manager>
        <security:authentication-provider>
            <security:user-service>
                <security:user name="rest_demo" password="rest_demo"
                authorities="ROLE_REST_DEMO"/>
            </security:user-service>
        </security:authentication-provider>
    </security:authentication-manager>
</beans:beans>
```

Web.xml Updates

Extend now the `contextConfigLocation` context parameter, to be aware of the the new Spring security configuration file `security-applicationContext.xml`.

Hook into Spring security.

```
<context-param>
 <param-name>contextConfigLocation</param-name>
 <param-value>
     classpath:spring/applicationContext.xml
      classpath:spring/security-applicationContext.xml
 </param-value>
</context-param>
```

```
 <!--Hook into spring security-->
 <filter>
 <filter-name>springSecurityFilterChain</filter-name>
 <filter-class>org.springframework.web.filter.DelegatingFilterProxy
 </filter-class>
</filter>
<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/rest/*</url-pattern>
</filter-mapping>
```

Invoking REST

In browser invoke `localhost:8080/lab8/rest/podcasts`

and enter username `rest_demo`, password `rest_demo`.

```
CURL : curl -H  "Content-Type: application/json"  -u rest_demo:rest_
demo -X GET http://localhost:8080/lab8/rest/podcasts
```

Wrapping Up

In this chapter we reviewed of OAuth 2 standard for securing RESTful APIs and did an exercise on implementing basic Spring security. We also reviewed concepts about HTTP, server, and client caching.

To summarize, in this book we reviewd REST APIs concepts in three tracks: Architecture, Design, and Coding.

Topics covered on the architecture front included Web Architecture Style, API Solution Architecture, API Portfolio Architecture, API Platform Architecture, API Management, and Security-OAuth.

Topics included on the design track included REST APIs Fundamentals, Data exchange formats, SOAP vs. REST, XML vs JSON, Introduction to API design: REST and JAX-RS, API design best practices, Modeling RESTful APIs, Building RESTful API-Framework, Interacting with RDBMS (MySQL) and NoSQL databases, Consuming RESTful API (i.e., JSON, XML), Security, and API Caching.

We also completed excercises on the coding track to understand how each concept can be implemented at the end of each chapter.

Index

A

B

S. Patni, *Pro RESTful APIs*, DOI 10.1007/978-1-4842-2665-0

C

D

E

F

G

H

I

J, K

L

M

N

O

P

Q

R

S

T

U, V

W

X, Y, Z