**Chapter 5**

**API Portfolio and Framework**

Sanjay Patnia\*

a UCSC Extension, Santa Clara, California, USA

**Abstract**

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.

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.

* 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**

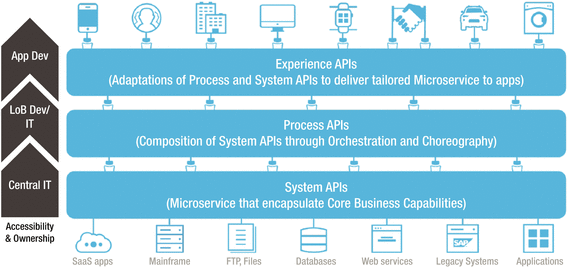
From an innovation or business perspective, there are forces to publish API as early as possible. From an IT Governance perspective, as late as possible. In a compromise solution, APIs are published early but only to pilot consumers, with the expectation that there will be changes and API will break 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](#bookmark) multilayer framework consists of:



***Figure 5-1.*** *APIs multilayered framework*

* 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 engineering design patterns.

**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](https://en.wikipedia.org/wiki/Object_(computer_science)%252525252525252523Object%2525252525252525252520(computer%2525252525252525252520science)) that provides an abstract [interface](https://en.wikipedia.org/wiki/Interface_(computer_science)%252525252525252523Interface%2525252525252525252520(computer%2525252525252525252520science)) to some type of [database](https://en.wikipedia.org/wiki/Database%252525252525252523Database) 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).

**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. Micronaut framework, which is used for the implementation in the exercises, 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 Micronaut is dependency injection. Micronaut provides a lightweight 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.

**Framework - Services**

In the previous chapter we implemented flight passenger s API for READ operations, This exercise uses message domain object to implement CRUD ( Create, Read, Update and Delete) operations. Message domain object structure is pretty simple. There is an id, which identifies a message, and several other fields that we can see in the JSON representation below.

{ "id":1,

“message”:”Welcome",

"from":"James",

"to":"John",

"creationDate":1388213547000

}

**Pom.xml**

Add following to pom.xml

<dependency>

<groupId>javax.inject</groupId>

<artifactId>javax.inject</artifactId>

<version>1</version>

</dependency>

**Message**

Here is a POJO defining properties of message.

package com.rest.model;

public class Message {

private int id;

private String message;

private String from;

private String to;

private String creationDate;

public String getMessage() {

return message;

}

public void setMessage(String message) {

this.message = message;

}

public String getFrom() {

return from;

}

public void setFrom(String from) {

this.from = from;

}

public String getTo() {

return to;

}

public void setTo(String to) {

this.to = to;

}

public String getCreationDate() {

return creationDate;

}

public void setCreationDate(String creationDate) {

this.creationDate = creationDate;

}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

}

**MessageController**

In the message controller we have CRUD operations for message

package com.rest.controller;

import com.rest.model.Message;

import com.rest.service.MessageService;

import io.micronaut.http.annotation.Get;

import io.micronaut.http.annotation.Controller;

import io.micronaut.http.HttpHeaders;

import io.micronaut.http.HttpResponse;

import io.micronaut.http.MediaType;

import io.micronaut.http.annotation.Produces;

import io.micronaut.http.annotation.Controller;

import io.micronaut.http.annotation.Delete;

import io.micronaut.http.annotation.Get;

import io.micronaut.http.annotation.Post;

import io.micronaut.http.annotation.Put;

import io.micronaut.http.annotation.Body;

import java.util.List;

@Controller("/message") // <2>

public class MessageController {

MessageService messageService;

public MessageController(MessageService messageService) { // <3>

this.messageService = messageService;

}

@Produces(MediaType.TEXT\_XML)

@Get("/xml")

public HttpResponse<?> messageXml() {

Message message = new Message();

message.setMessage("Hello from Micronaut");

final String xml = encodeAsXml(message);

return HttpResponse.ok(xml).contentType(MediaType.APPLICATION\_XML\_TYPE);

}

@Produces(MediaType.TEXT\_JSON)

@Get("/json")

public Message messageJson() {

Message message = new Message();

message.setMessage("Hello from Micronaut");

return message;

}

private String encodeAsXml(final Message message) {

return String.format("<message>%s</message>", message.getMessage());

}

@Post

public Message createMessage(@Body Message message) {

messageService.createMessage(message);

return message;

}

@Get("/{id}")

public Message getMessage (int id) {

Message message = messageService.getMessage(id);

return message;

}

@Get

public List<Message> getMessages() {

List<Message> messages = messageService.getMessages();

return messages;

}

@Put("/{id}")

public void updateMessage (int id, @Body Message update) {

messageService.updateMessage(id, update);

}

@Delete("/{id}")

public void deleteMessage(int id) {

messageService.deleteMessage(id);

}

}

**MessageService**

all the methods for CRUD (Create-Read-Update-Delete) operations which have operations on in memory of messages are moved here.

package com.rest.service;

import com.rest.model.Message;

import java.util.Map;

import java.util.List;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.concurrent.atomic.AtomicInteger;

public class MessageService {

static private Map<Integer, Message> messageRepo = new HashMap<Integer, Message>();

static private AtomicInteger idCounter = new AtomicInteger();

public Message getMessage(int id) {

Message message = messageRepo.get(id);

return message;

}

// add message

public void createMessage(Message message) {

message.setId(idCounter.incrementAndGet());

messageRepo.put(message.getId(), message);

}

// update message

public void updateMessage(int id, Message update) {

Message current = messageRepo.get(id);

current.setMessage(update.getMessage());

current.setFrom(update.getFrom());

current.setTo(update.getTo());

current.setCreationDate(update.getCreationDate());

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

}

// Delete message

public void deleteMessage(int id) {

Message current = messageRepo.remove(id);

}

public List<Message> getMessages() {

return new ArrayList<Message>(messageRepo.values());

}

**API Tests(Curl)**

curl -d '{ "id":1, "message":"test", "from":"test", "to":"test", "creationDate":"12/12//2012" }' -H 'Content-Type: application/json' <http://localhost:8080/message>

{“id":1,"message":"test","from":"test","to":"test","creationDate":"12/12//2012"}

curl -d '{ "id":2, "message":"test2", "from":"test", "to":"test", "creationDate":"12/12//2012" }' -H 'Content-Type: application/json' <http://localhost:8080/message>

{“id":2,"message":"test2","from":"test","to":"test","creationDate":"12/12//2012"}

curl <http://localhost:8080/message>

[{“id”:1,"message":"test","from":"test","to":"test","creationDate":"12/12//2012"},{"id":2,"message":"test2","from":"test","to":"test","creationDate":"12/12//2012}

curl http://localhost:8080/message/1

{“id":1,"message":"test","from":"test","to":"test","creationDate":"12/12//2012"}

curl http://localhost:8080/message/2

{“id":2,"message":"test2","from":"test","to":"test","creationDate":"12/12//2012"}

curl -X "DELETE" http://localhost:8080/message/2

curl http://localhost:8080/message/2

{"message":"Not Found","\_links":{"self":{"href":"/message/2","templated":false}},"\_embedded":{"errors":[{"message":"Page Not Found”}]}}

curl http://localhost:8080/message

[{“id":1,"message":"test","from":"test","to":"test","creationDate":"12/12//2012"}]

To Review

Now we have two APIs in our portfolio one one is for the messaging and other for flights passengers. Flight API implements relationship of two objects Flight and Passenger where as Message service allows creation and deletion of messages in addition to read. It is important to follow same design for both the APIs.