

REST Web Services

Chantri Polprasert

Readings

- Mark Masse ‘REST API Design Rulebook’
- Richardson, L., and Ruby, S., RESTful Web Services, O'Reilly, 2007.
- Daigneau, R., Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services, Addison-Wesley, 2011.
- Fielding, R.T., Taylor, R.N., Erenkrantz, J.R., Gorlick, M.M., Whitehead, J., Khare, R., and Oreizy, P., Reflections on the REST architectural style and “principled design of the modern web architecture” (impact paper award). In Foundations of Software Engineering (FSE), ACM, 2017.

Some material © Richardson and Ruby (2007), Daigneau (2011), and Fielding et al. (2017).

Outline

- Introduction
- Resource-Oriented Architecture
- Resource-Oriented Analysis and Design
- REST and ROA best practices
- SOAP vs REST

Introduction: Thin servers and fat clients

- Traditional Web applications supply HTML to the users' browsers — they use a **thin client** and **fat server** architecture.
- With Ajax, single-page applications, and native mobile apps, however, nowadays, many applications have **fat clients** and **thin servers**.
- The Google applications (Mail, Calendar, Maps, etc.) are good examples.
- On the browser side, we are moving toward more complete presentation and business logic tiers.
- On the server side, the focus is moving towards providing **data source** services efficiently to a large number of concurrent clients.

Introduction: The server as a data source

- Fat client applications mainly use HTTP to **request** small blobs of data from the server and send **updates** in response to user actions back to the server.
- This means the server must support deep and flexible access to the data source layer in **machine readable** format **over HTTP**.
- One interesting consequence of the architecture: the server-side data source layer becomes a **resource** usable by multiple client applications in new, unexpected **mashups**.
- An API Mashup: a powerful application development technique that involves the **integration and orchestration of multiple APIs** from different sources, enabling them to work together seamlessly to create new and innovative functionalities in software applications
 - e-commerce applications that integrate product information APIs with payment processing APIs
 - Travel Planning Services: Flight + Hotel + Weather APIs

Introduction: The server provides Web services

- So, we want our Web server to provide **other programs** with access to information using HTTP.
- HTTP endpoints that serve data to **other programs** rather than to Web browsers are what we call **Web services**.
- **Web services** are Web/application servers offering programmatic access to data or services.
- What are some criteria for the API/protocol for communication between client application and server?
 - Open: consumed by different type of clients
 - Consistent: Easy for developers to make use of the information provided by the web services
 - Flexible: easy to updated/modify

Introduction: Web service standards

- In the 2000's, Web service standards diverged in two directions.
- First up: **SOAP and the WS-* stack (Web Service Star)**
 - Simple Object Access Protocol.
 - It's purely about format and transport, not about what the message means.
 - Use **XML** Information Set for its message format, and relies on application layer protocols, most often HTTP/HTTPS, for message negotiation and transmission.
 - Driven primarily by industry and **complex** developer's tools
 - Aims to make invocation of a remote service similar to invoking a local function (RPC).(Add, AddResult, a, b etc.)
 - Exposes **operations** (business logic functions) to clients.
 - **The API exported by your service is up to the developer/architect.**

Request

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope">
  <soap:Body>
    <Add xmlns="http://calculator.example.com/">
      <a>10</a>
      <b>5</b>
    </Add>
  </soap:Body>
</soap:Envelope>
```

Envelope: The root element that identifies the XML as a SOAP message

Response

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope">
  <soap:Body>
    <AddResponse xmlns="http://calculator.example.com/">
      <AddResult>15</AddResult>
    </AddResponse>
  </soap:Body>
</soap:Envelope>
```

WS-* Stack

- SOAP is a messaging protocol that uses XML, and the WS-* standards are a set of extensions that add features like security, reliability, and addressing to SOAP messages.
- WS-Security is a prominent WS-* standard that provides message-level security through encryption, digital signatures, and authentication, while other standards like WS-Policy and WS-Addressing define different aspects of web service communication.

WS-*

- **Definition:** WS-* refers to a family of standards that build upon SOAP to provide a more complete framework for web services.
- **Purpose:** The WS-* standards address a wide range of concerns that are not covered by the basic SOAP protocol, such as security, reliability, and more complex routing.
- **Implementation:** These standards are often implemented as extensions within the SOAP message, particularly in the <env:Header> element.

WS-* Standards Build On Top of SOAP

Once SOAP provided a reliable way to send messages, organizations wanted to solve *enterprise-level problems*:

- How to **secure** messages?
- How to **guarantee delivery**?
- How to **coordinate transactions**?
- How to **describe** services so others can find and use them?

So the **WS-*** (Web Services) standards were created as **extensions to SOAP**, using the SOAP header to carry extra information.

Concept	Purpose
SOAP	Defines message format and transport rules
WS-Security	Encrypts/signs SOAP messages
WS-Addressing	Adds routing and endpoint metadata
WS-Policy	Declares service requirements
WS-ReliableMesaaging	Guarantees delivery
WS-Transaction	Coordinates multi-step processes

SOAP + WS-Security (Authentication & Integrity)

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"  
    xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-s  
    <soap:Header>  
        <wsse:Security soap:mustUnderstand="1">  
            <wsse:UsernameToken>  
                <wsse:Username>api_user</wsse:Username>  
                <wsse:Password Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-t  
                    9b3e9b6e02a...  
                </wsse:Password>  
            </wsse:UsernameToken>  
        </wsse:Security>  
    </soap:Header>  
  
    <soap:Body>  
        <CheckAccount xmlns="http://bank.example.com/">  
            <AccountNumber>1234567890</AccountNumber>  
        </CheckAccount>  
    </soap:Body>  
</soap:Envelope>
```

SOAP & WS-*

SOAP, combined with the WS-* stack, is rather heavyweight.

Many in the Web community believe that SOAP + WS-* goes in the **wrong direction**:

- The reason for the Web's success is its **simplicity**, based on URIs, HTTP, and XML.
- With the WS-* stack, systems become so complex only commercial code generating tools can properly connect Web service clients and servers.
- Heavyweight services **ignore** the Web, using it as a transport layer for SOAP envelopes encapsulated in HTTP POST requests.
- The alternative, coined as **Representational State Transfer** (REST), tried to put the “Web” back in Web services

Introduction: Web service standards

- WS-* is oriented towards **enterprise application integration** scenarios, especially **business-to-business** integration.
- **RESTful web services** are driven from a more **do-it-yourself** perspective, trying to maintain the simplicity of pure HTTP without extra layers of abstraction.
- The API action is fixed by HTTP verb (GET, HEAD, PUT, POST, PATCH, DELETE ect..) -> a simple way to consume API
- RESTful services are now the defacto standard for data centric rich client applications and services that mainly allow **third party mashups**.

Introduction: REST philosophy

- Representational state transfer (REST): a software architectural style that was created to guide the design and development of the architecture for the World Wide Web.
- REST defines a set of constraints for how the architecture of a distributed, Internet-scale hypermedia system, such as the Web, should behave
- The REST philosophy was advocated by Roy Fielding (one of the HTTP RFC authors):
 - Application state is divided into **resources**.
 - Resources are **addressable** using a universal syntax.
 - There is a **uniform interface** for state transfer with **a constrained set of operations**.
 - The transfer protocol is **client-server, stateless, cacheable**, and **layered**. (Fielding, 2000, as quoted in Wikipedia.)
- For **Web services**, prime candidates are the URI for addressability and HTTP for operations and state transfer.

Outline

- Introduction
- Resource-Oriented Architecture
- Resource-Oriented Analysis and Design
- REST and ROA best practices

Resource-Oriented Architecture: Introduction

- Richardson and Ruby (2007) were the first to propose a concrete design pattern for RESTful Web services.
- **Resource-Oriented Architecture** is a software architecture for distributed systems that follows the REST principles.
- Resource-oriented architectures are based on four concepts:
 - **Resources**: pointers to chunks of data that can be stored on computers as bit strings.
 - **Resource names**: URLs.
 - **Resource representations**: a resource may be represented in many format, e.g., HTML, JSON, XML, English, Vietnamese, etc.
 - **Links between resources**: resources should be linked to each other to improve usability and discoverability.

Resource-Oriented Architecture: Properties of resource-oriented services

Systems constructed using the resource-oriented approach have four RESTful properties:

- **Addressability**: All resources can be retrieved and manipulated via their URI.
- **Statelessness**: Application state is maintained by the client. Persistence of resources is provided by the server. Application state information needed by the server should be placed **in the request**, not maintained by the server.
- **Connectedness**: Resources link to each other through URIs in their representations.
- **Hierarchical Structure**: For example, a collection of users might contain individual user resources
- **A uniform interface**: the **only** operations are HTTP GET, PUT, DELETE, POST, HEAD, and OPTIONS.

Resource-Oriented Architecture: Properties of resource-oriented services

A **resource** is something that can be stored on a computer as a sequence of bits.

Examples (Richardson and Ruby, 2007):

- User: A user resource might be identified by a URI like /users/123, where 123 is the unique identifier for that user.
- Photo Collection: A collection of photos belonging to a user could be represented by the URI /users/123/photos, where each photo within this collection is also a resource.
- Blog Posts: (Resource URI: /posts/{postId})
- Notifications (Resource URI: /users/{userId}/notifications)
- Devices: (Resource URI: /devices/{deviceId}) for IoT devices
- Transactions (Resource URI: /accounts/{accountId}/transactions)

A **single** piece of data can be pointed to by **multiple** resources.

Resource-Oriented Architecture: URI in resource-oriented services

- Each resource must have **at least one** URI providing a **name** and **address** of the resource.
- URIs should be **descriptive** and have an **intuitive structure**: e.g.
 - Search with Multiple Criteria:
 - URI: /tickets/search?state=open&priority=high&assigned_to=john_doe
 - Searching for open tickets that are high priority and assigned to the user "john_doe".
 - Filter Orders by Date Range:
 - URI: /orders/search?start_date=2024-01-01&end_date=2024-12-31
 - Filtering orders placed within the year 2024.
- One resource might have multiple URIs, but it might be good to identify one as the **canonical** URI and redirect to that URI.

Resource-Oriented Architecture: Addressability in resource-oriented services

- Addressability means that every important resource can be uniquely and directly identified by a URI.
- If something exists in your system — user, order, dataset, document → it must be reachable through a stable, unique URI.
- Each URI identifies what the resource is, not what to do.
- Follow the syntax and semantics of URIs based on RFC 3986

```
GET /customers/123
```

```
GET /customers/123/orders
```

Resource-Oriented Architecture: Addressability in resource-oriented services

 Poor (RPC-style)	 Good (ROA / REST)
POST /getCustomerInfo	GET /customers/123
Hidden resource inside body	Resource visible in URI
Action-based (<code>get</code> , <code>create</code>)	Noun-based (<code>customers</code>)
Not cacheable	Cacheable (GET)
Hard to share or link	Linkable/bookmarkable

```
POST /getCustomerInfo  
{ "customerId": 123 }
```

- URI = an action, not a resource.
- “Customer 123” is hidden inside the payload.
- No stable URI for caching or linking.

RPC (originated in the 1970s–1980s)

- **Concept:** Call a function on another computer as if it were local.”
- To abstract away the network layer so developers could call remote methods like: `result = get_student(12)` even though `get_student()` executes on a remote server.

RFC 3986

- Define the generic syntax of a Uniform Resource Identifier (URI)

```
URI = scheme ":" hier-part [ "?" query ] [ "#" fragment ]
```

- Scheme:** tell how to access or interpret the resource. It's followed by a colon (:) e.g. http:, https:, mailto: (Protocol or method to reach this resource)
- Hier-part:** hierarchical part identifies the main structure of the resource — usually includes the host, path, and optional port.

<https://api.example.com/customers/123>

Component	Example	Description
//	(double slash)	Indicates start of authority
authority	api.example.com	Host or domain name
path	/customers/123	Resource location on the host

RFC 3986

- Define the generic syntax of a Uniform Resource Identifier (URI)

```
URI = scheme ":" hier-part [ "?" query ] [ "#" fragment ]
```

- ["?" query] (optional) The query component comes after a question mark (?). It's used for parameters or filters, not for identifying the resource itself.

<https://api.example.com/customers?country=TH&status=active>

- ["#" fragment] (optional): identify a secondary resource within the main one — often a subsection of a document.

<https://en.wikipedia.org/wiki/URI#Syntax>

A section inside the page

Guidelines for URI Design

RFC 3986: `URI = scheme ":" hier-part ["?" query] ["#" fragment]`

`https://api.example.com/customers/123?active=true#section1`

Component	Symbol	Example
scheme	<code>https</code>	How to access
authority (part of hier-part)	<code>api.example.com</code>	Where it lives
path (part of hier-part)	<code>/customers/123</code>	What resource
query	<code>?active=true</code>	Optional filters
fragment	<code>#section1</code>	Optional local reference

Guidelines for URI Design

RFC 3986: `URI = scheme ":" hier-part ["?" query] ["#" fragment]`

- Forward slash separator (/) must be used to indicate a hierarchical relationship between resources
 - <http://api.canvas.restapi.org/shapes/polygons/quadrilaterals/squares>
- A trailing forward slash (/) should not be included in URIs
- Hyphens (-) should be used to improve the readability of URIs
 - <http://api.example.restapi.org/blogs/mark-masse/entries/this-is-my-first-post>
- Underscores (_) should not be used in URIs
- Lowercase letters should be preferred in URI paths

`http://api.example.restapi.org/my-folder/my-doc`
`HTTP://API.EXAMPLE.RESTAPI.ORG/my-folder/my-doc`
`http://api.example.restapi.org/My-Folder/my-doc`

1
2
3

Guidelines for URI Design

- File extensions should not be included in URIs. Instead, they should rely on the media type, as communicated through the HTTP's Content-Type header

`http://api.college.restapi.org/students/3248234/transcripts/2005/fall.json` ①
`http://api.college.restapi.org/students/3248234/transcripts/2005/fall` ②

- Resource Modeling: The URI path conveys a REST API's resource model

`http://api.soccer.restapi.org/leagues/seattle/teams/trebuchet`

indicates that each of these URIs should also identify an addressable resource:

`http://api.soccer.restapi.org/leagues/seattle/teams`
`http://api.soccer.restapi.org/leagues/seattle`
`http://api.soccer.restapi.org/leagues`
`http://api.soccer.restapi.org`

Guidelines for URI Design

- Variable path segments may be substituted with identity-based values
 - In a RESTful URI, parts of the path (segments) can be variables — placeholders that represent specific instances of a resource — and when you make a request, you replace those placeholders with actual identifiers.
 - Some URI path segments are static; meaning they have fixed names that may be chosen by the REST API's designer.
 - Other URI path segments are variable, which means that they are automatically filled in with some identifier that may help provide the URI with its uniqueness.
 - An example below has three variables (leagueId, teamId, and playerId):
`http://api.soccer.restapi.org/leagues/{leagueId}/teams/{teamId}/players/{playerId}`
- The substitution of a URI template variables may be done by a REST API or its clients.
- Each substitution may use a numeric or alphanumeric identifier
`http://api.soccer.restapi.org/leagues/seattle/teams/trebuchet/players/21`

Guidelines for URI Design

```
resources :students do  
  resources :courses  
end
```

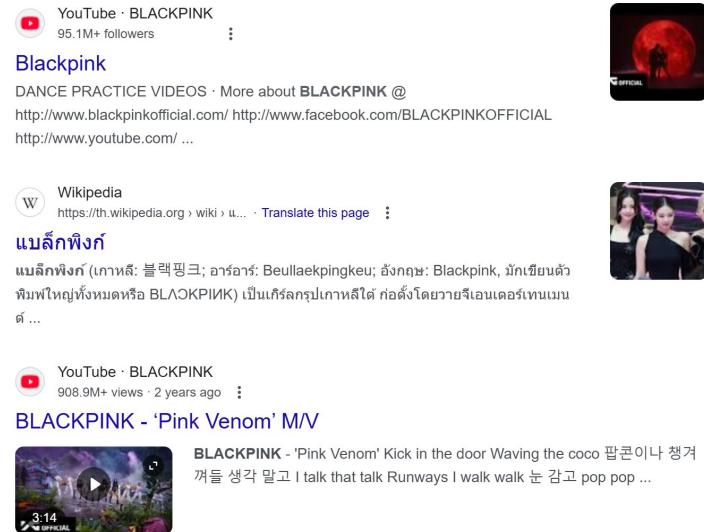


Method	URI Pattern	Controller#Action
GET	/students/:student_id/courses/:id	courses#show
POST	/students/:student_id/courses	courses#create

Given 'GET /students/42/courses/3', Rails maps student_id = 3 and id = 3.

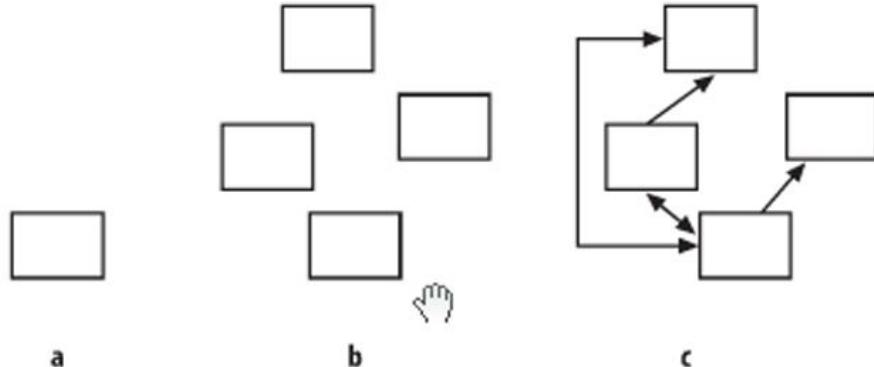
Resource-Oriented Architecture: Statelessness in resource-oriented services

- In Resource Oriented Architecture, HTTP requests happen in **isolation** and **all necessary state** is **embedded in the request**.
 - Each request from a client to a server must contain all the information needed to understand and process that request.
 - **Example:** a client can use
https://www.google.com/search?q=blackpink&sca_esv=a0b6cc145a6c8085&ei=mSEsZ52ACaiW4-EP9cPX6Ao&start=40 for **the 5th** page of results for the Google query ‘blackpink’.
 - The request includes everything the server needs: (search keyword = blackpink, start at 40)
 - Simplifies the transfer protocol and enables other features such as horizontal scaling and caching.
 - Server-maintained application state, though sometimes necessary, is not RESTful.



Resource-Oriented Architecture: Connectedness in resource-oriented services

- The property where every resource within the web service is reachable from a designated base resource through a series of successive HTTP GET requests.
- A client, starting from a known entry point (the base URI), can discover and navigate to all other resources by following links or relationships provided within the resource representations.



Richardson and Ruby (2007), Fig. 4-4

- (a) A RPC-style service, with everything hidden behind one URI.
(b) An addressable structure without connectivity.
(c) An addressable and connected structure. This is the most usable of the three services

```
{  
  "items": [  
    { "id": 1, "name": "Item A" },  
    { "id": 2, "name": "Item B" }  
  ],  
  "_links": {  
    "self": { "href": "/api/items?page=1" },  
    "next": { "href": "/api/items?page=2" },  
    "last": { "href": "/api/items?page=10" }  
  }  
}
```

When retrieving a list of resources, the API can include links for pagination (next, previous, first, last pages), to help clients navigate through large datasets.

Resource-Oriented Architecture: Uniform interface realization

- The goal of the uniform interface is to **standardize communication** between clients and servers so that:
 - Any client can interact with any resource in the same way.
 - Clients don't need to know server-side implementation details.
 - Intermediaries (proxies, caches, gateways) can understand and optimize messages without special knowledge.
- In essence: REST turns “every interaction” into a predictable pattern.

Resource-Oriented Architecture: Uniform interface realization: Why HTTP?

Resource-Oriented Architecture can theoretically use any protocol, but HTTP became its natural and most successful realization because it already embodied the **uniform interface principles**.

HTTP already provided:

- **A small, fixed set of standard operations** — GET, POST, PUT, DELETE, etc. → Maps directly to CRUD operations (Create, Read, Update, Delete).
- **Resource identification via URIs** → Each resource has a global unique address, like /students/12.
- **Stateless request-response model** → Each request carries full context (no session memory).
- **Self-descriptive messages** → Headers describe data format, encoding, caching, authentication, etc.
- **Hypermedia (links) built into the web** → Foundation for Hypermedia as the Engine of Application State (HATEOAS) — clients navigate through hyperlinks.

Resource-Oriented Architecture: Uniform interface realization: from RPC to REST

- Before REST, most distributed systems used RPC (Remote Procedure Call) or SOAP-based approaches — tightly coupled protocols where:
 - Clients called specific methods (e.g., getUserData(), updateProfile()).
 - Each API defined its own verbs, payloads, and formats.
- This created integration chaos — every system spoke a different “language”.
- Fielding proposed REST as an alternative:
 - “Let’s stop creating new verbs.
 - Let’s use one simple set of verbs (HTTP methods) and focus on resources instead of actions.”
- That’s the core of Uniform Interface realization — shifting from RPC-style verbs to resource-oriented nouns.

Resource-Oriented Architecture: Uniform interface realization

A resource-oriented service is limited to the HTTP methods:

- **GET** is used to retrieve a representation of a resource.
- **PUT** is used to **update a resource or create a new resource if it does not exist**. When a PUT request is sent, the entire resource is replaced with the new data. If the resource does not exist, it will be created.
- **POST** is used for **creating a new resource or adding a new entity to an existing resource**. When a POST request is sent, the server creates a new resource and assigns a new URL to it.
- **PATCH** is used to (partially) update one or more attributes of an existing resource.
- **DELETE** is used to delete an existing resource.

```
// PUT example  
PUT /users/1  
{  
  "id": 1,  
  "name": "Ichiro",  
  "age": 22  
}
```

Sends a request to replace user 1's record.

```
// POST example  
POST /users  
{  
  "name": "Saburo",  
  "age": 18  
}
```

Sends a request to create a new user

Resource-Oriented Architecture: Uniform interface realization

HTTP Method	Meaning (Uniform Action)	Example	Typical Server Response
GET	Retrieve a representation of a resource	GET /students/12	200 OK + JSON body
POST	Create a new resource (usually under a collection)	POST /students + JSON body	201 Created + Location header
PUT	Replace an existing resource (full update)	PUT /students/12 + JSON body	200 OK or 204 No Content
PATCH	Modify part of a resource (partial update)	PATCH /students/12 + JSON body	200 OK
DELETE	Remove a resource	DELETE /students/12	204 No Content
HEAD	Retrieve metadata only (no body)	HEAD /students/12	Headers only
OPTIONS	Discover what methods are supported on the resource	OPTIONS /students/12	Allow: GET, PUT, DELETE, OPTIONS

Resource-Oriented Architecture: Uniform interface realization

	PUT	POST
Request Body	Contains the full updated data for the resource	Contain data for the new resource.
URI Meaning	Use the URI to directly identify the resource to update (e.g. user 1)	Use the URI to specify the collection where a new resource will be created
Idempotency	Yes. The same request gives the same result	No. Can produce different results each time.
Existing Resources	Replace the entire resource with the request body	Create a new resource under a collection
Request Body	Require a body	Optional

Resource-Oriented Architecture: Uniform interface realization: PUT & POST

```
POST /students
Content-Type: application/json

{
  "name": "Ariana",
  "major": "AI"
}
```

201 Created
Location: /students/12

```
{
  "id": 12,
  "name": "Ariana",
  "major": "AI"
}
```

- Non Idempotent
- Every repeated POST creates a new record (e.g., /students/13, /students/14).

```
PUT /students/12
Content-Type: application/json

{
  "id": 12,
  "name": "Ariana Grande",
  "major": "Data Science"
}
```

200 OK

```
{
  "id": 12,
  "name": "Ariana Grande",
  "major": "Data Science"
}
```

- Idempotent
- Repeating this same PUT request does not create duplicates — it just ensures student #12 always has that state.

Resource-Oriented Architecture: Uniform interface realization: Example in Rails

config/routes.rb

```
Rails.application.routes.draw do  
  resources :students  
end
```

This automatically creates RESTful routes:

HTTP Verb	Path	Controller#Action	Purpose
GET	/students	students#index	List all students
POST	/students	students#create	Create a new student
GET	/students/:id	students#show	Show one student
PUT / PATCH	/students/:id	students#update	Update (replace/modify) existing student
DELETE	/students/:id	students#destroy	Delete a student

Resource-Oriented Architecture

What's the drawback?

Exercise

You are designing a RESTful API for a **university management system** that manages the following entities:

- **Students** — each has an **id**, **name**, and **major**.
- **Courses** — each has an **id**, **title**, and **credits**.
- **Enrollments** — a student can enroll in multiple courses.
- **Instructors** — each teaches multiple courses.

Tasks:

1. **Design appropriate URIs** that follow REST principles for the following operations:
 - (a) Retrieve the list of all students
 - (b) Retrieve details of a specific student
 - (c) Add a new student
 - (d) Retrieve all courses that a specific student is enrolled in
 - (e) Enroll a student into a course
 - (f) Retrieve all courses taught by a specific instructor
2. For each URI, **specify the HTTP method** you would use and **briefly explain why**.
3. (Bonus) Suggest one **non-RESTful URI design** for one of the above operations and explain **why it violates** REST's Uniform Interface.