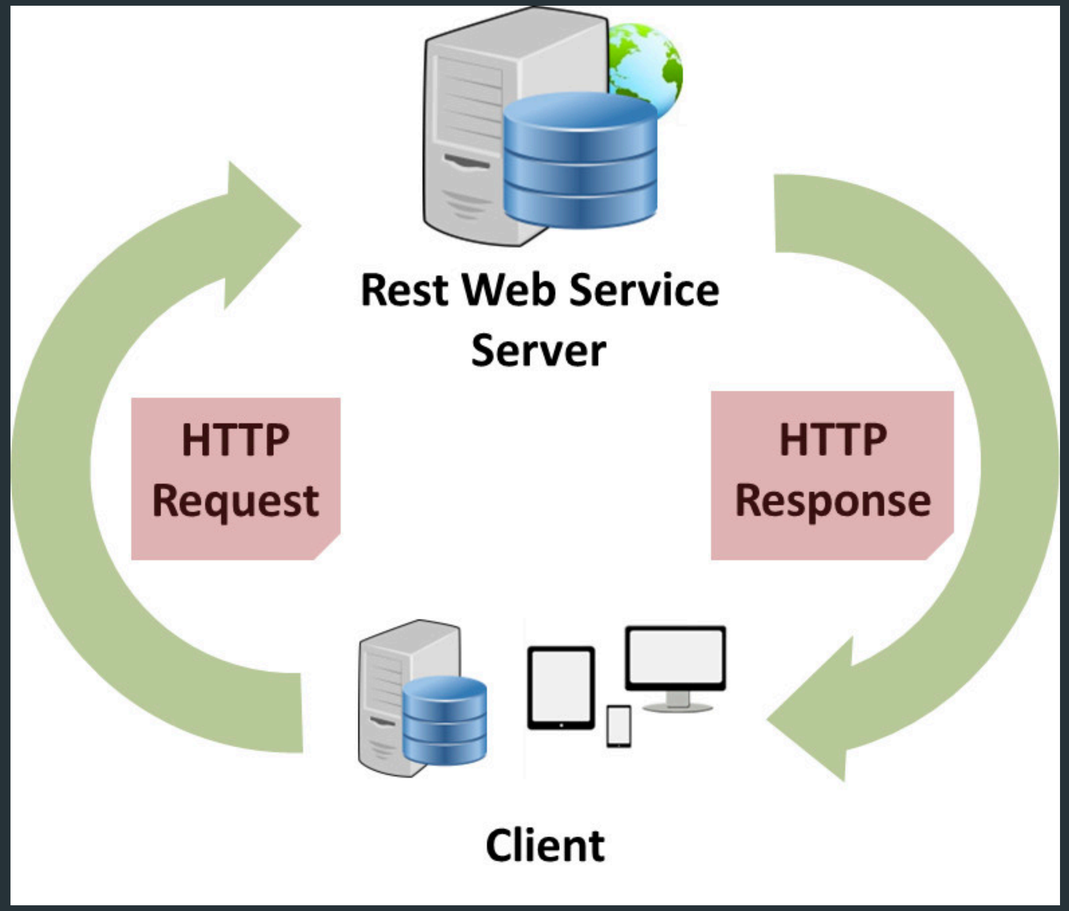
Lecture #4
RESTful Web Services

WSMT2023

Introduction

- What are RESTful Web Services?
- Why do we need RESTful Web Services?
- Advantages of RESTful Web Services



HTTP Methods

- HTTP Methods used in RESTful Web Services
  - GET
  - POST
  - PUT
  - DELETE

```python
import requests
# GET request
response = requests.get('https://jsonplaceholder.typicode.com/posts/1')
print(response.json())
# POST request
data = {
    'title': 'foo',
    'body': 'bar',
    'userId': 1
}
response = requests.post('https://jsonplaceholder.typicode.com/posts', json=data)
print(response.json())
# PUT request
data = {
    'id': 1,
    'title': 'foo',
    'body': 'bar',
    'userId': 1
}
response = requests.put('https://jsonplaceholder.typicode.com/posts/1', json=data)
print(response.json())
# DELETE request
response = requests.delete('https://jsonplaceholder.typicode.com/posts/1')
print(response.status_code)
```

RESTful API Design

- Guidelines for designing RESTful APIs
- Resource identification through URIs
- Use of HTTP Methods for CRUD operations
- Use of HTTP status codes

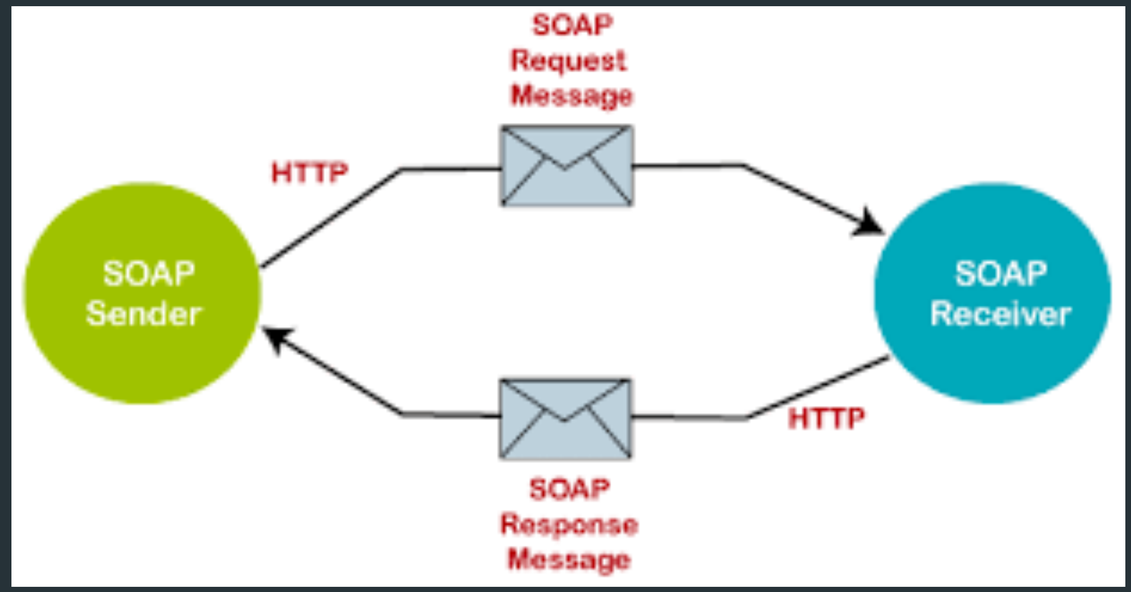
```python
from flask import Flask, jsonify, request

app = Flask(__name__)
# GET request
@app.route('/posts/<int:id>', methods=['GET'])
def get_post(id):
    post = get_post_from_database(id)
    if post:
        return jsonify(post)
    else:
        return jsonify({'error': 'Post not found'}), 404
# POST request
@app.route('/posts', methods=['POST'])
def create_post():
    data = request.get_json()
    post = create_post_in_database(data)
    return jsonify(post), 201
# PUT request
@app.route('/posts/<int:id>', methods=['PUT'])
def update_post(id):
    data = request.get_json()
    post = update_post_in_database(id, data)
```

```python
from flask import Flask, jsonify, request

app = Flask(__name__)
# GET request
@app.route('/posts/<int:id>', methods=['GET'])
def get_post(id):
    post = get_post_from_database(id)
    if post:
        return jsonify(post)
    else:
        return jsonify({'error': 'Post not found'}), 404
# POST request
@app.route('/posts', methods=['POST'])
def create_post():
    data = request.get_json()
    post = create_post_in_database(data)
    return jsonify(post), 201
# PUT request
@app.route('/posts/<int:id>', methods=['PUT'])
def update_post(id):
    data = request.get_json()
    post = update_post_in_database(id, data)
    if post:
        return jsonify(post)
    else:
```

```
def get_post(id):
    post = get_post_from_database(id)
    if post:
        return jsonify(post)
    else:
        return jsonify({'error': 'Post not found'}), 404
# POST request
@app.route('/posts', methods=['POST'])
def create_post():
    data = request.get_json()
    post = create_post_in_database(data)
    return jsonify(post), 201
# PUT request
@app.route('/posts/<int:id>', methods=['PUT'])
def update_post(id):
    data = request.get_json()
    post = update_post_in_database(id, data)
    if post:
        return jsonify(post)
    else:
        return jsonify({'error': 'Post not found'}), 404
# DELETE request
@app.route('/posts/<int:id>', methods=['DELETE'])
def delete_post(id):
    delete_post_from_database(id)
    return '', 204
```

RESTful Web Services vs SOAP Web Services

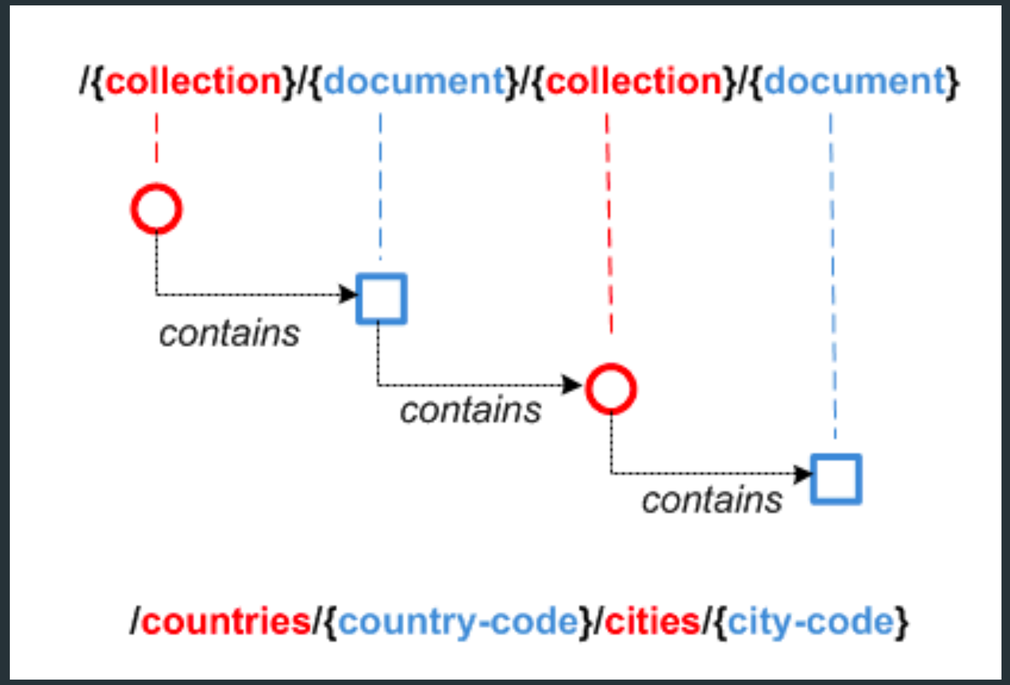
- Differences between RESTful Web Services and SOAP Web Services
- Pros and cons of RESTful Web Services
- Pros and cons of SOAP Web Services



Identifying Resources in RESTful Web Services

Identifying Resources

- Define resources as domain nouns, for example, "users," "posts," "comments," etc.
- Use a resource hierarchy for complex entities, for example, "/users/{user_id}/posts" or "/posts/{post_id}/comments".
- Avoid using verbs or actions in resource identification, as it violates the principle of using HTTP methods for operations.



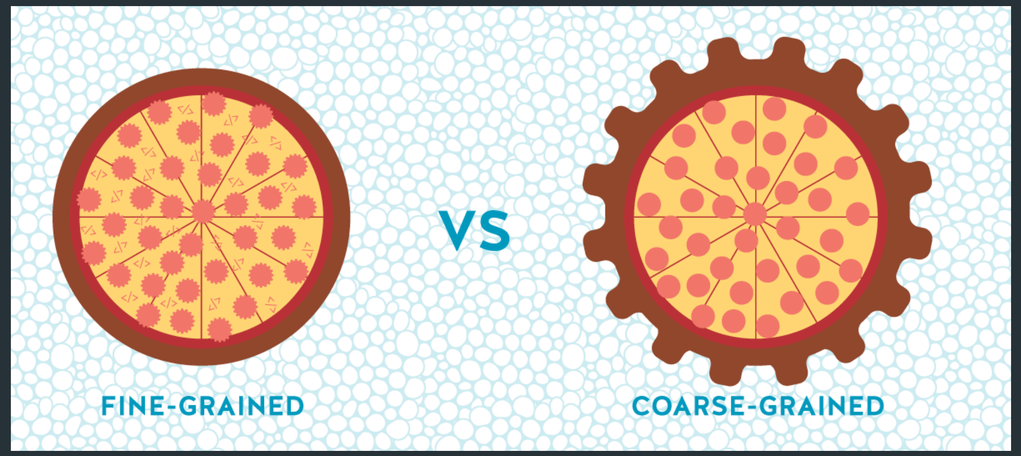
Identifying Resources

```
// Define a user resource
GET /users/{user_id}
POST /users
PUT /users/{user_id}
DELETE /users/{user_id}

// Define a post resource
GET /posts/{post_id}
POST /users/{user_id}/posts
PUT /posts/{post_id}
DELETE /posts/{post_id}

// Define a comment resource
GET /comments/{comment_id}
POST /posts/{post_id}/comments
PUT /comments/{comment_id}
DELETE /comments/{comment_id}
```

Choosing Resource Granularity in RESTful Web Services

Choosing Resource Granularity

- Define resources based on their significance and level of detail.
- Consider the audience and use cases when choosing the granularity of a resource.
- Avoid creating overly granular resources that result in excessive API calls.



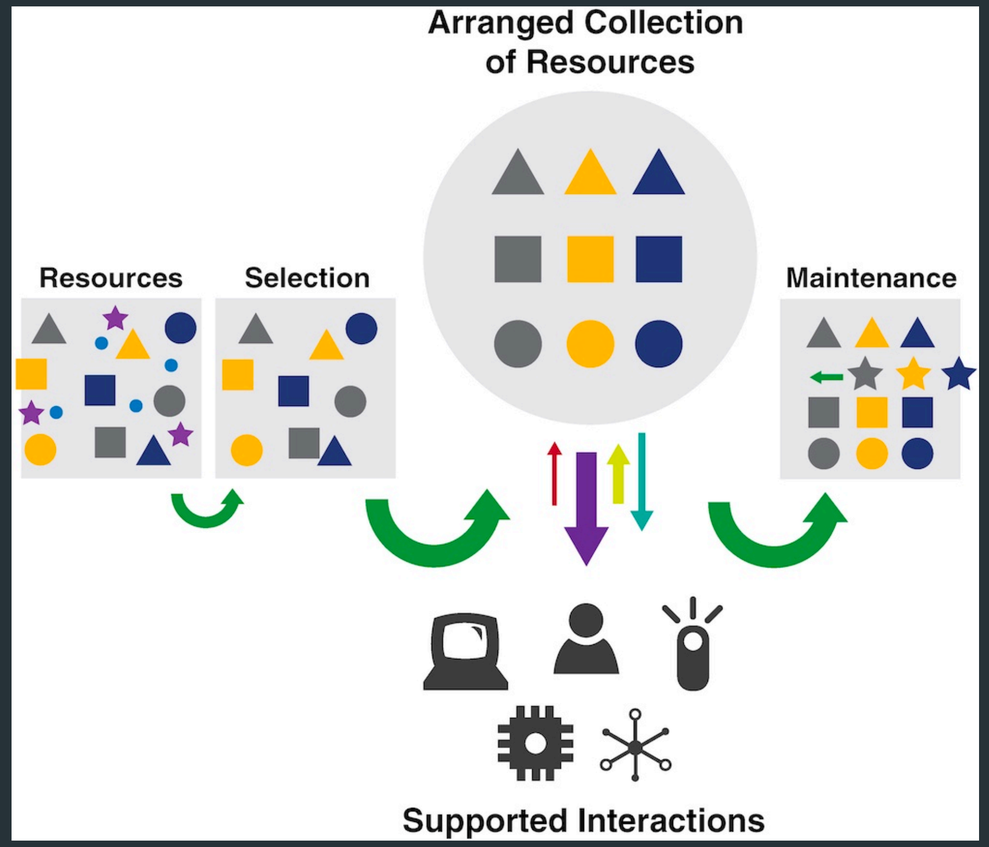
Choosing Resource Granularity

```
// Example of overly granular resources
GET /users/{user_id}/posts/{post_id}/comments/{comment_id}
POST /users/{user_id}/posts/{post_id}/comments/{comment_id}
PUT /users/{user_id}/posts/{post_id}/comments/{comment_id}
DELETE /users/{user_id}/posts/{post_id}/comments/{comment_id}

// Example of too general resource
GET /posts
POST /posts
PUT /posts/{post_id}
DELETE /posts/{post_id}

// Example of balanced resource granularity
GET /posts/{post_id}/comments
POST /posts/{post_id}/comments
PUT /comments/{comment_id}
DELETE /comments/{comment_id}
```

Organizing Resources into Collections in RESTful Web Services

Organizing Resources into Collections

- Define collections and explain how they relate to resources.
- How collections can help to organize related resources.
- Provide examples of collections, such as users, products, and orders.

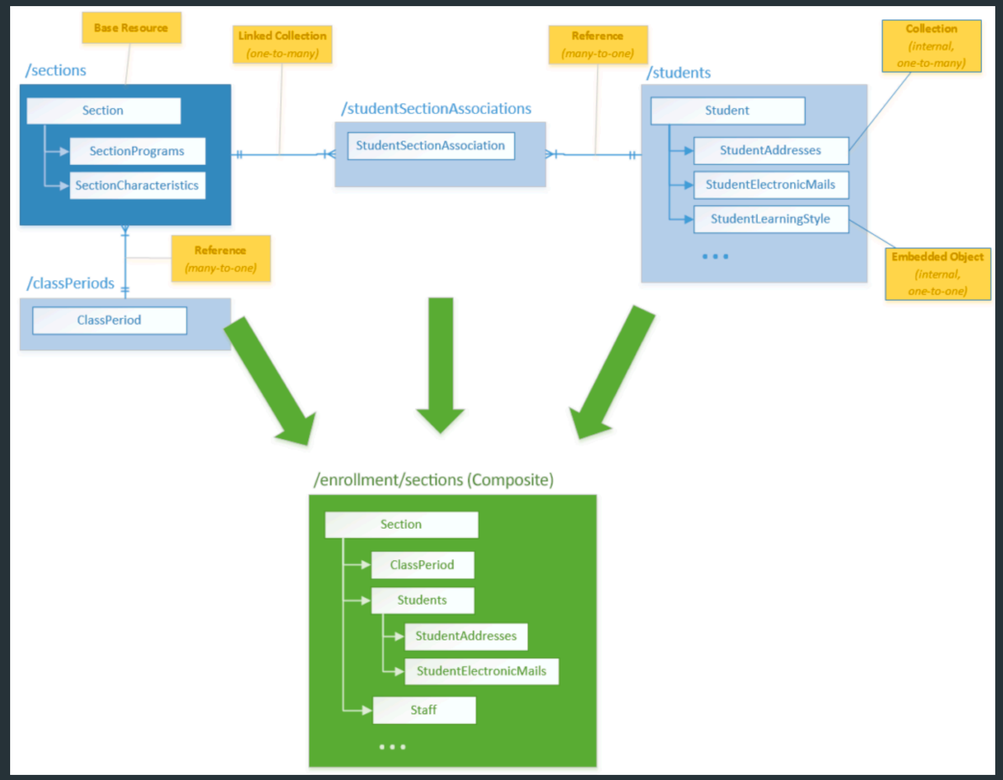


Choosing Resource Granularity

```
// Example of resource organization without collections
GET /users/{user_id}/profile
POST /users/{user_id}/profile
PUT /users/{user_id}/profile
DELETE /users/{user_id}/profile
GET /users/{user_id}/orders
POST /users/{user_id}/orders
PUT /users/{user_id}/orders/{order_id}
DELETE /users/{user_id}/orders/{order_id}

// Example of resource organization with collections
GET /users/{user_id}
POST /users
PUT /users/{user_id}
DELETE /users/{user_id}
GET /users/{user_id}/orders
POST /users/{user_id}/orders
PUT /orders/{order_id}
DELETE /orders/{order_id}
```

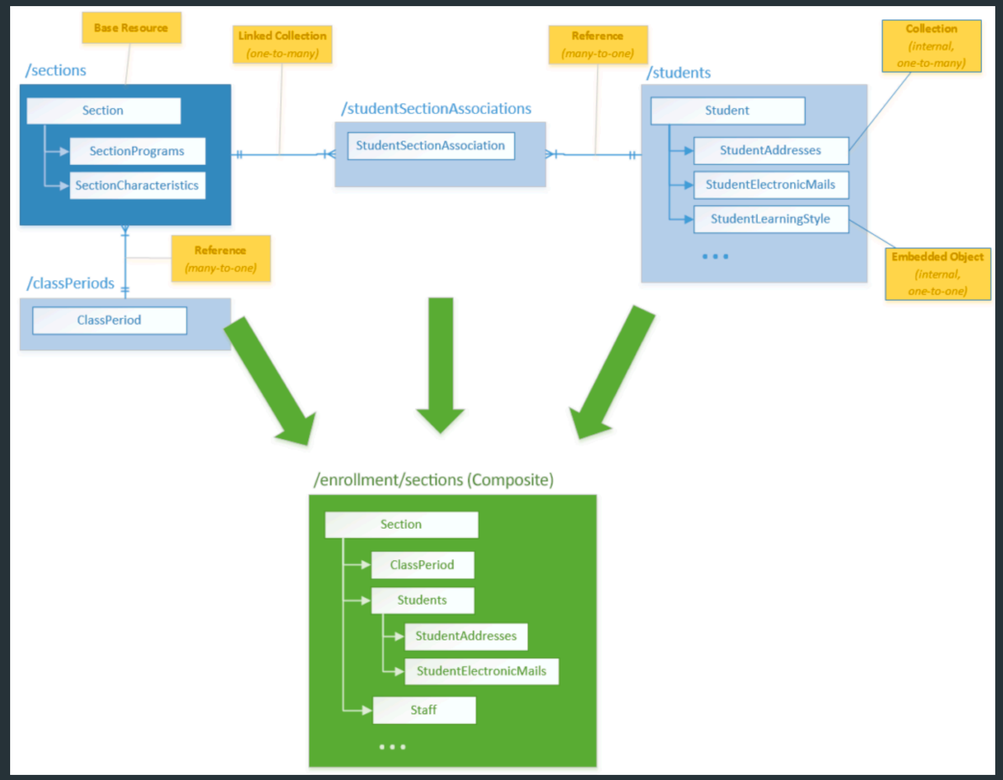
When to Combine Resources into Composites

- Understanding the decision to combine resources is important for RESTful API designers
- Combining resources can simplify client usage
- Combining resources can improve API performance



Combining resources can simplify client usage

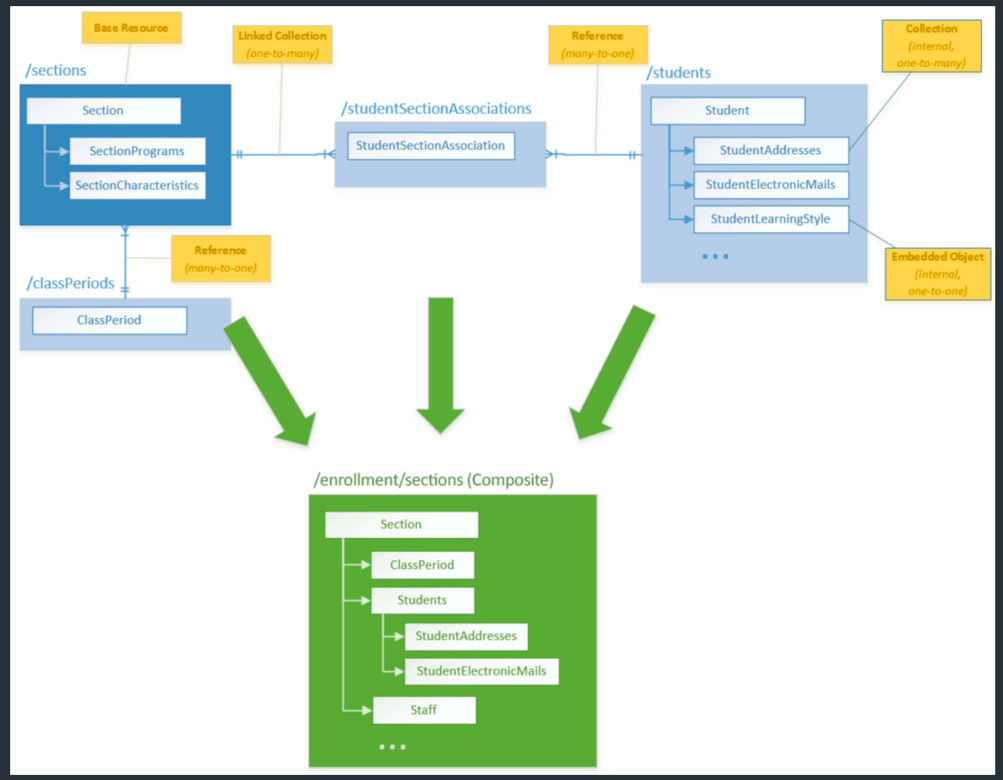
- Example scenario: retrieving all books published by a certain publisher
- Notes: This scenario highlights the benefits of combining resources into a composite resource. Without a composite resource, the client would need to make multiple API calls to retrieve all the relevant information, whereas a composite resource would provide all the information in a single API call.



Combining resources can improve API performance

- Example scenario: retrieving related resources such as publisher, author, and book data in a single API call

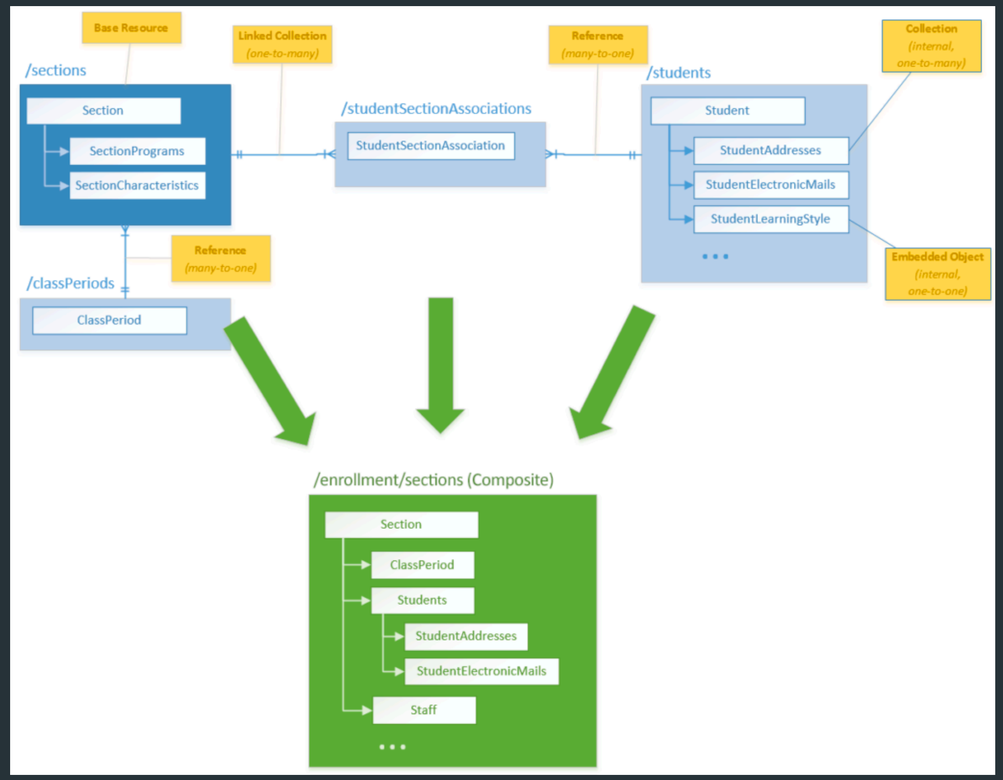
- Notes: This scenario illustrates how combining related resources into a composite resource can improve API performance. Instead of making multiple API calls to retrieve the publisher, author, and book data, a single API call to the composite resource would provide all the information needed.



Combining resources can simplify resource modeling

- Example scenario: combining resources that are always retrieved together, such as a user's profile information and their list of posts

- Notes: This scenario demonstrates how combining related resources into a composite resource can simplify resource modeling. By combining a user's profile information and their list of posts into a single composite resource, the API can be simplified and the number of resources that need to be modeled and maintained separately is reduced. This can save time and reduce complexity, especially as the API evolves and new resources are added.



When to consider not combining resources into composites

When not to combine resources into composites

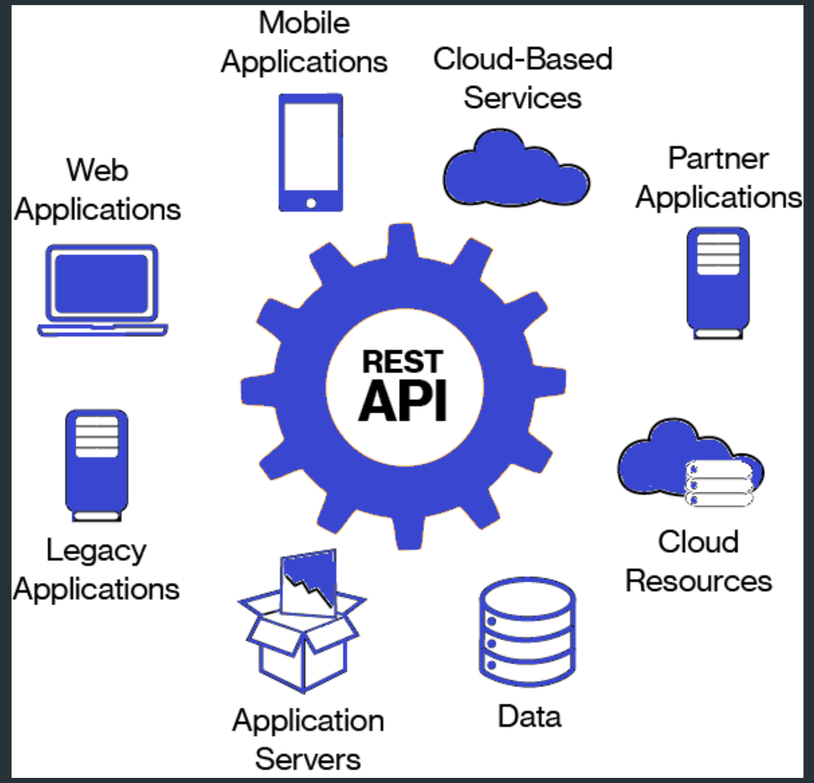
Designing an effective RESTful API requires balancing multiple factors

```
const express = require('express');
const app = express();

// Create a composite resource for books published by a certain publisher
app.get('/publishers/:publisher/books', (req, res) => {
  const publisherId = req.params.publisher;
  // Retrieve the publisher information
  const publisher = getPublisher(publisherId);
  // Retrieve the books published by the publisher
  const books = getBooksByPublisher(publisherId);
  // Combine the publisher and book information into a single response
  const response = {
    publisher: publisher,
    books: books
  };
 // Send the response to the client
  res.json(response);
});

// Start the server
app.listen(3000, () => {
  console.log('Server listening on port 3000');
});
```

How to Support Computing/Processing Functions in RESTful Web Services

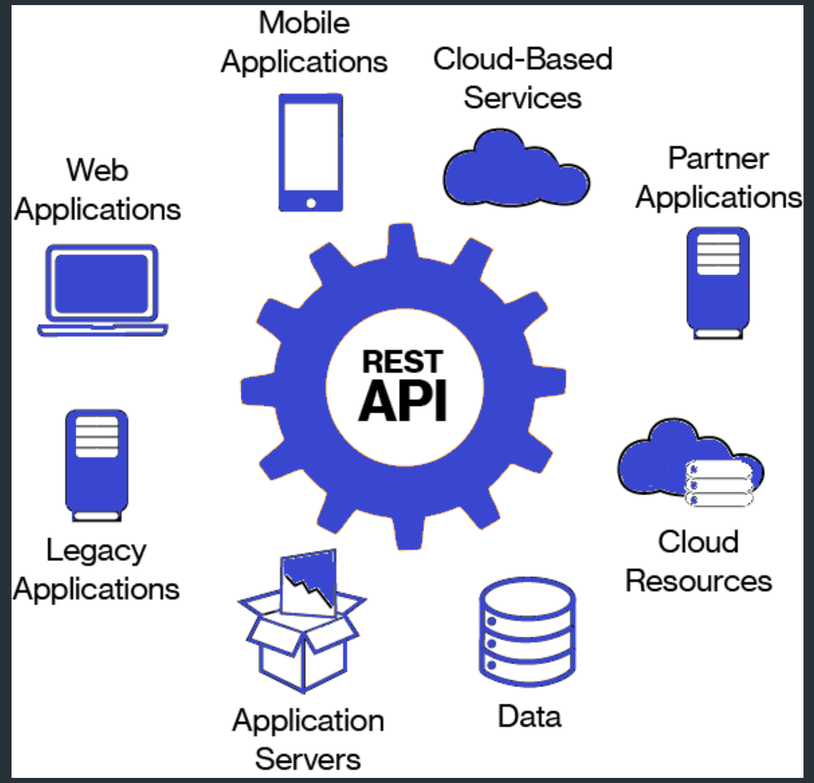
How to Support Computing/Processing Functions in RESTful Web Services

- Definition of computing/processing functions
- Importance of supporting computing/ processing functions in RESTful web services
- Challenges in supporting computing/ processing functions in RESTful web services



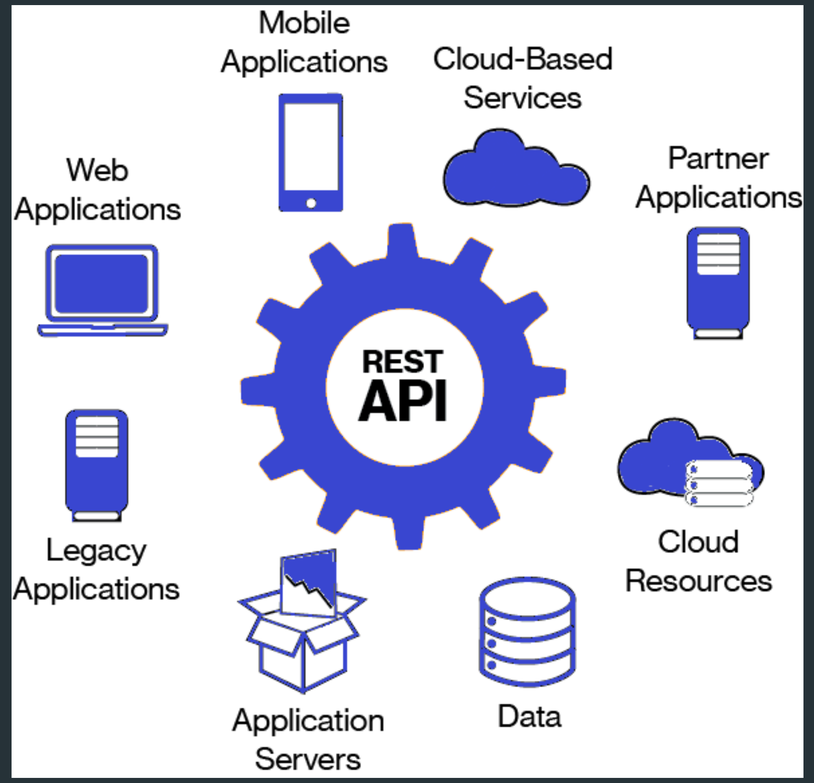
Approaches to supporting computing/processing functions

- Query parameters



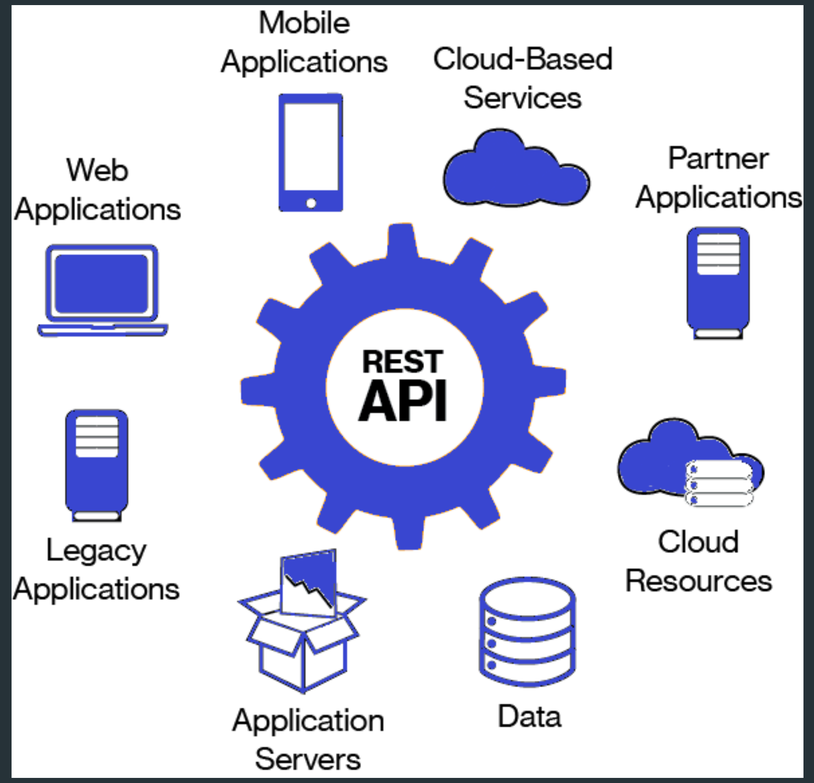
Approaches to supporting computing/processing functions

- Query parameters
- Resource-oriented APIs



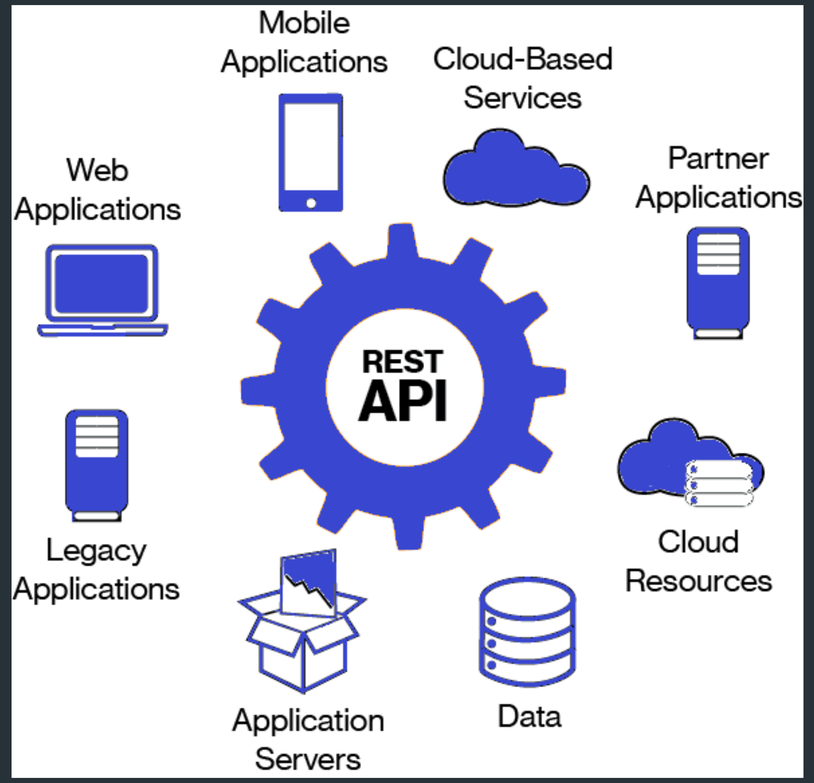
Approaches to supporting computing/processing functions

- Query parameters
- Resource-oriented APIs
- Custom API endpoints



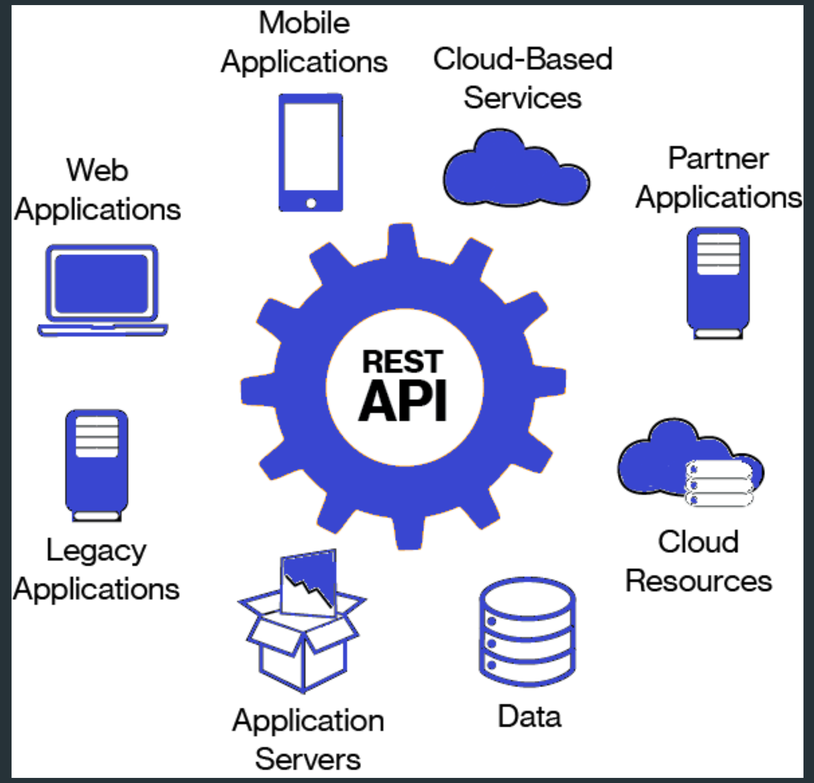
Best practices for supporting computing/processing functions

- Use HTTP caching



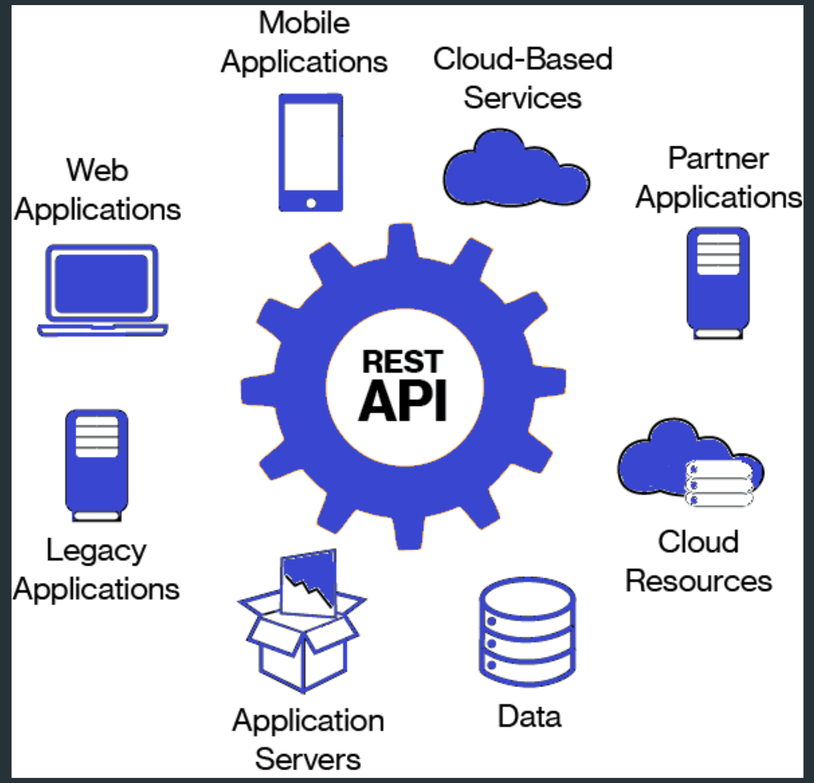
Best practices for supporting computing/processing functions

- Use HTTP caching
- Use pagination



Best practices for supporting computing/processing functions

- Use HTTP caching
- Use pagination
- Use parameter validation and error handling

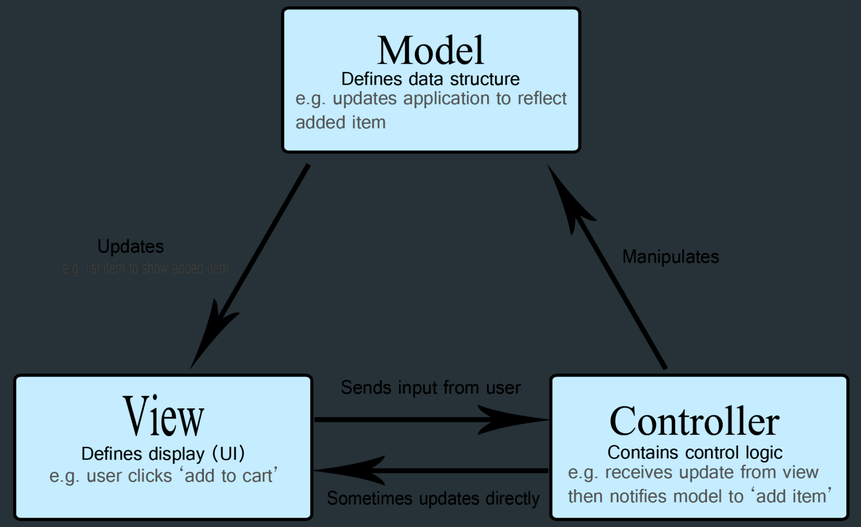


```
const express = require('express');
const app = express();

// Define an API endpoint for calculating the sum of two numbers
app.get('/calculate', (req, res) => {
  const num1 = parseInt(req.query.num1);
  const num2 = parseInt(req.query.num2);

  // Check that the query parameters are valid
  if (isNaN(num1) || isNaN(num2)) {
    res.status(400).json({ error: 'Invalid query parameters' });
    return;
  }

  // Calculate the sum of the two numbers
  const sum = num1 + num2;

  // Send the result to the client
  res.json({ result: sum });
});

// Start the server
app.listen(3000, () => {
  console.log('Server listening on port 3000');
});
```

When and How to Use Controllers to Operate on Resources

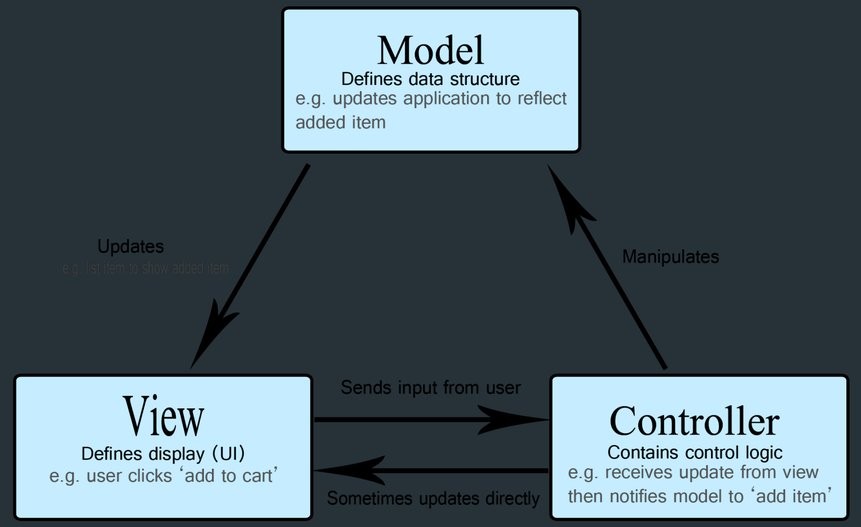
When and How to Use Controllers to Operate on Resources

- Restful Web Services allow us to create APIs that can be easily accessed by different devices and platforms.
- Controllers play an essential role in handling HTTP requests and responses to operate on resources.
- In this presentation, we'll discuss when and how to use controllers to operate on resources in RESTful Web Services.



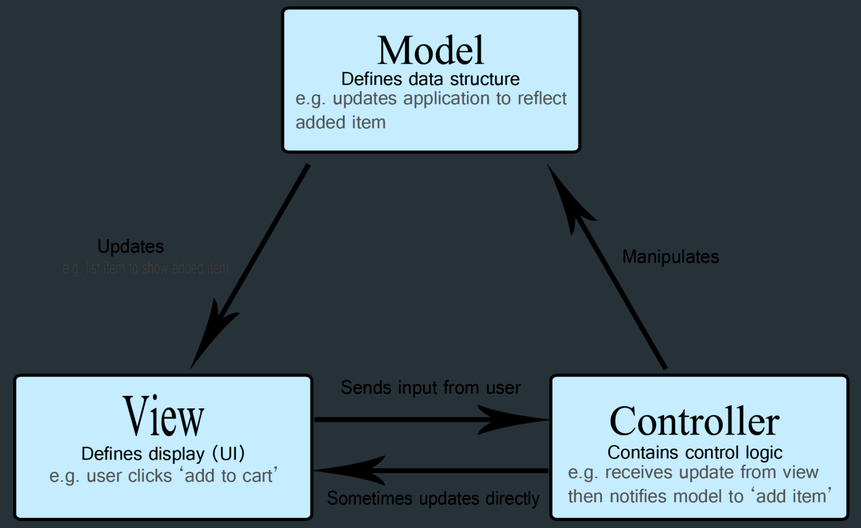
What are Controllers?

- Controllers are a key part of the Model-View-Controller (MVC) architectural pattern.
- They receive incoming requests from clients, retrieve the necessary data from models, and return the appropriate response.
- In RESTful Web Services, controllers are responsible for handling HTTP requests to perform CRUD (Create, Read, Update, Delete) operations on resources.



How to Use Controllers to Operate on Resources

- Use the HTTP GET method to retrieve a resource from the server.
- Use the HTTP POST method to create a new resource on the server.
- Use the HTTP PUT method to update an existing resource on the server.
- Use the HTTP DELETE method to remove a resource from the server.

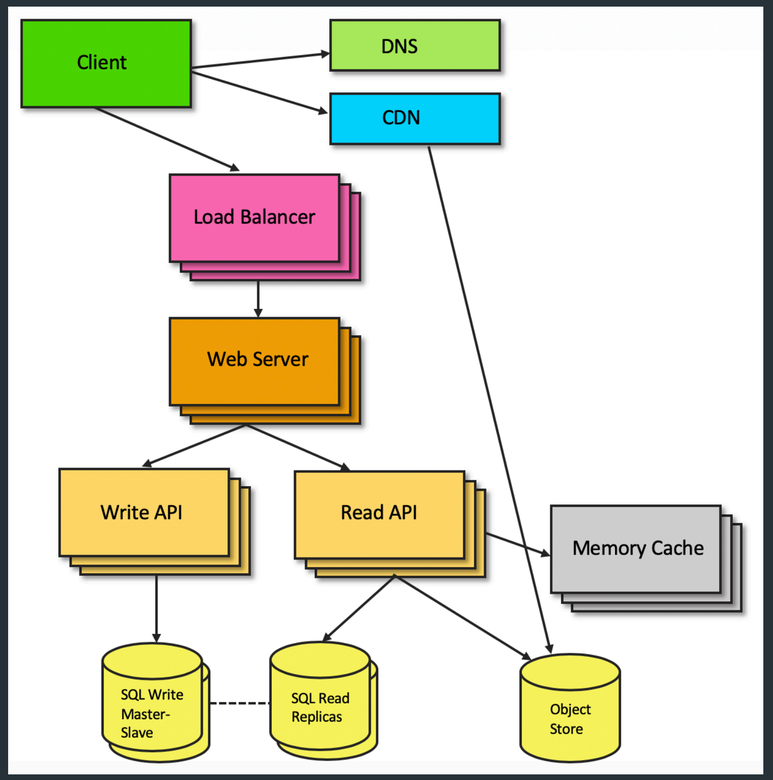


```java
// GET request to retrieve a resource by ID
@RequestMapping(value = "/resource/{id}", method = RequestMethod.GET)
public ResponseEntity<Resource> getResourceById(@PathVariable("id") String id) {
    Resource resource = resourceService.getResourceById(id);
    return new ResponseEntity<Resource>(resource, HttpStatus.OK);
}

// POST request to create a new resource
@RequestMapping(value = "/resource", method = RequestMethod.POST)
public ResponseEntity<Resource> createResource(@RequestBody Resource resource) {
    resourceService.createResource(resource);
    return new ResponseEntity<Resource>(resource, HttpStatus.CREATED);
}

// PUT request to update an existing resource
@RequestMapping(value = "/resource/{id}", method = RequestMethod.PUT)
public ResponseEntity<Resource> updateResource(@PathVariable("id") String id, @RequestBody Resource resource) {
    Resource updatedResource = resourceService.updateResource(id, resource);
    return new ResponseEntity<Resource>(updatedResource, HttpStatus.OK);
}

// DELETE request to remove a resource
@RequestMapping(value = "/resource/{id}", method = RequestMethod.DELETE)
public ResponseEntity<?> deleteResource(@PathVariable("id") String id) {
    resourceService.deleteResource(id);
    return new ResponseEntity<>(HttpStatus.NO_CONTENT);
}
```

Designing Representations

How to Use Entity Headers to Annotate Representations

- RESTful Web Services allow us to create APIs that can be easily accessed by different devices and platforms.
- Designing representations is an essential part of RESTful Web Services.
- How to use entity headers to annotate representations.



What are Entity Headers?

- Entity headers are used to provide metadata about the representation of a resource.
- They include headers such as Content-Type, Content-Length, and ETag.
- Entity headers help clients and servers to understand the format and content of the representation.

Status Line	**HTTP/1.1 200 OK**
General Header	Date : Wed, 11 Aug 2021 13:00:13 GMT
	Connection : Close
Response Header	Server : Apache / 1.3.27
	Accept-Ranges : bytes
Entity Header	Content-Type : text/html
	Content-Length : 200
	Last-Modified : 1 Aug 2021 13:00:13 GMT
Blank Line	
Message Body	<html>
	<head>
	<title> Welcome to the India <title>
	</head>
	<body>

How to Use Entity Headers to Annotate Representations

- Use the Content-Type header to specify the format of the representation.
- Use the Content-Length header to specify the length of the representation.
- Use the ETag header to specify a unique identifier for the representation.

Status Line	**HTTP/1.1 200 OK**
General Header	Date : Wed, 11 Aug 2021 13:00:13 GMT
	Connection : Close
Response Header	Server : Apache / 1.3.27
	Accept-Ranges : bytes
Entity Header	Content-Type : text/html
	Content-Length : 200
	Last-Modified : 1 Aug 2021 13:00:13 GMT
Blank Line	
Message Body	<html>
	<head>
	<title> Welcome to the India <title>
	</head>
	<body>

Example of Entity Headers in Practice

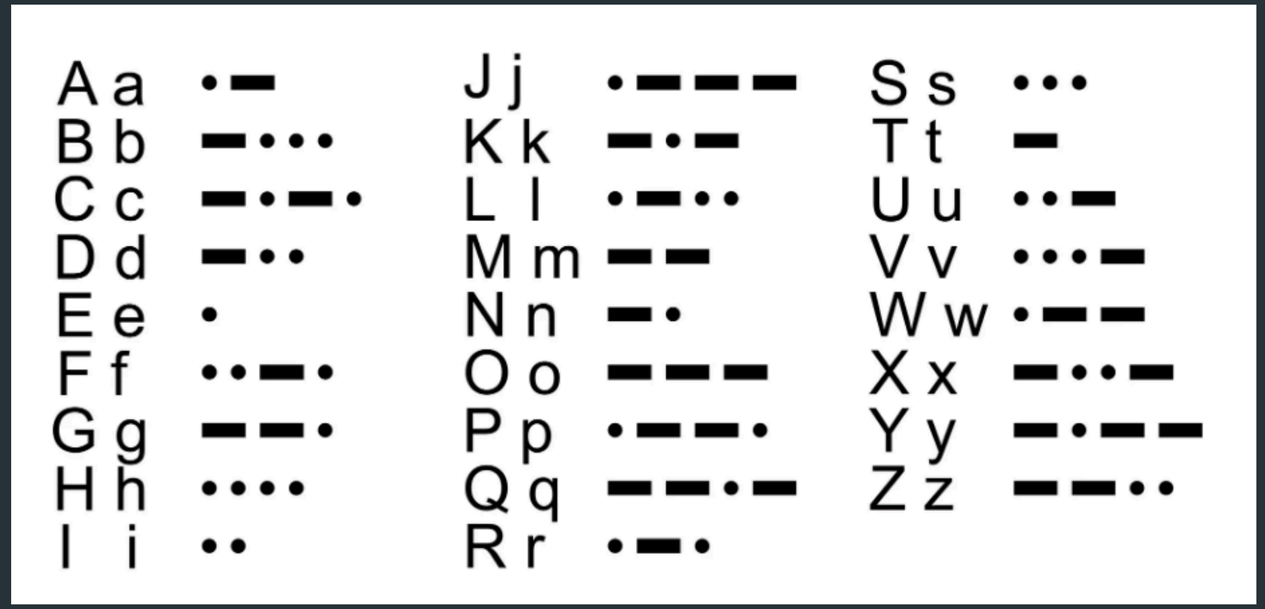
- Consider a RESTful API that returns information about a book.
- The representation of the book might include the title, author, publication date, and ISBN.
- We could use entity headers to provide additional metadata about the representation.

```
import datetime
from flask import Flask, jsonify, make_response

app = Flask(__name__)

# Dummy data representing a book
book = {
    'id': 1,
    'title': 'RESTful Web Services',
    'author': 'Leonard Richardson and Sam Ruby',
    'publication_date': datetime.date(2007, 5, 8)
}

# Define a function to return the book representation
def get_book_representation():
    # Get the current representation of the book
    representation = {
        'id': book['id'],
        'title': book['title'],
        'author': book['author'],
        'publication_date': book['publication_date'].strftime('%Y-%m-%d')
    }
```

```
        'title': book['title'],
        'author': book['author'],
        'publication_date': book['publication_date'].strftime('%Y-%m-%d')
    }

    # Set the entity headers to annotate the representation
    entity_headers = {
        'Last-Modified': book['publication_date'].strftime('%a, %d %b %Y %H:%M:%S GMT'),
        'ETag': f'"{book["id"]}-{book["publication_date"].timestamp()}"'
    }

    # Return the annotated representation and the entity headers
    return (representation, entity_headers)

# Define a route to get the book representation
@app.route('/books')
def get_book():
    representation, entity_headers = get_book_representation()

    # Return the annotated representation and the entity headers in the response
    response = make_response(jsonify(representation))
    for key, value in entity_headers.items():
        response.headers[key] = value
    return response

if __name__ == '__main__':
    app.run(debug=True)
```

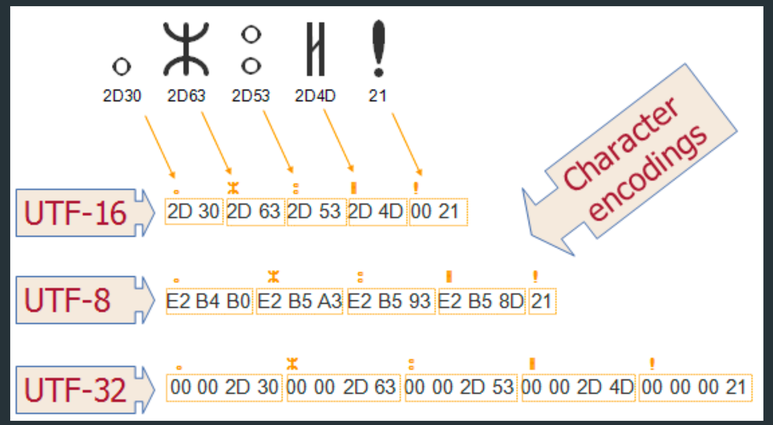
Introduction to Character Encoding

- Definition of character encoding and its role in web services
- Overview of the most commonly used character encoding schemes
- Explanation of how character encoding can impact data representation and transmission



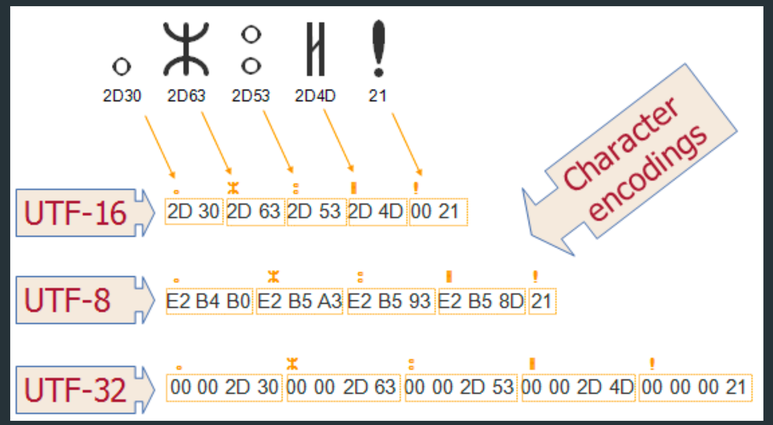
Common Character Encoding Schemes

- Detailed overview of ASCII, UTF-8, and ISO-8859-1 encoding schemes
- Explanation of the differences between these encoding schemes
- Examples of when to use each encoding scheme



Strategies for Avoiding Character Encoding Mismatch

- Establish a clear encoding scheme for all data exchanged between servers and clients.
- Validating input data.
- Use standard libraries and frameworks.



Understanding JSON Representations

- What is JSON
- How JSON is used in web services
- Benefits of using JSON

Understanding JSON Representations


```
{
  "firstName": "John",
  "lastName": "Doe",
  "age": 30,
  "email": "john.doe@example.com",
  "address": {
    "street": "123 Main St",
    "city": "Anytown",
    "state": "CA",
    "zip": "12345"
  }
}
```

Best Practices for Designing JSON Representations

- Naming conventions for JSON keys
- Keeping JSON representations simple and clear
- Using data types consistently

```
{
  "first_name": "John",
  "last_name": "Doe",
  "age": 30,
  "email_address": "john.doe@example.com"
}
```

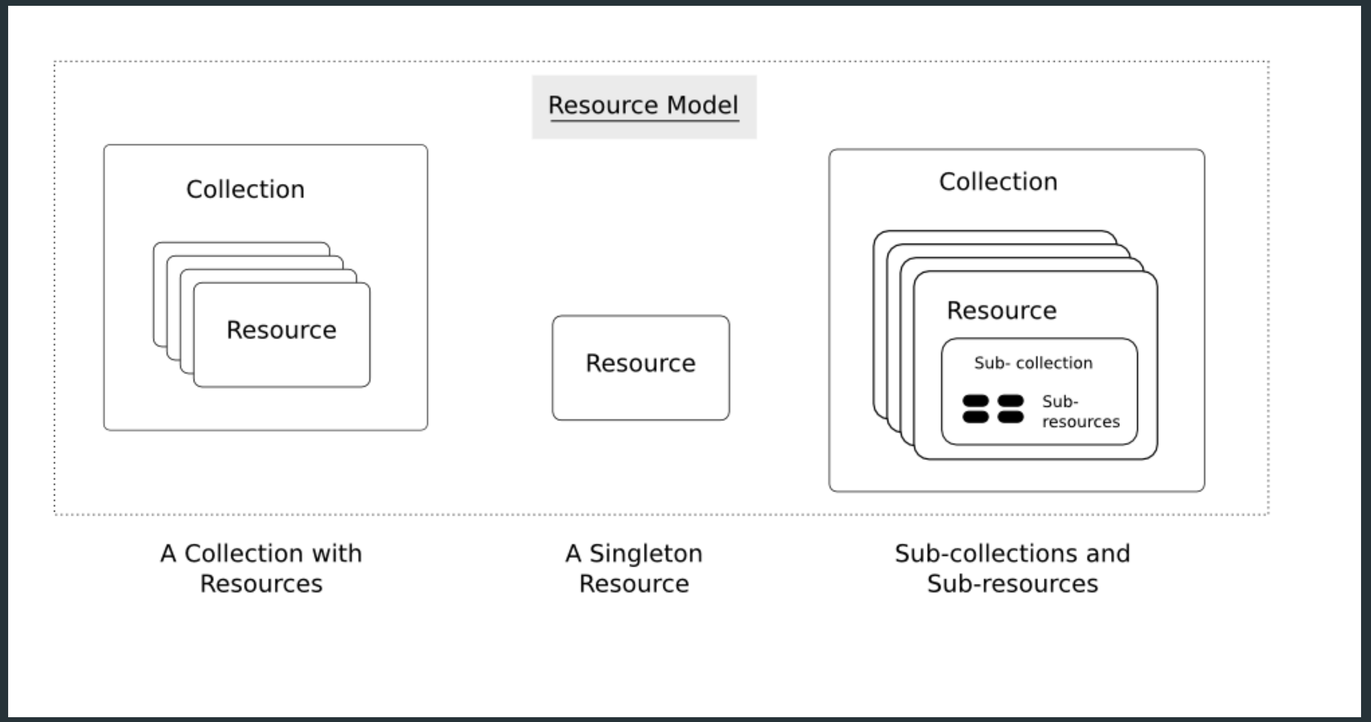
Designing Nested JSON Representations

- Using nested objects in JSON representations
- Avoiding deep nesting
- Providing documentation for nested representations

```
{
  "id": 1,
  "title": "Example post",
  "author": {
    "id": 2,
    "name": "John Doe",
    "email": "john.doe@example.com"
  },
  "comments": [
    {
      "id": 1,
      "body": "Example comment",
      "author": {
        "id": 3,
        "name": "Jane Doe",
        "email": "jane.doe@example.com"
      }
    }
  ]
}
```

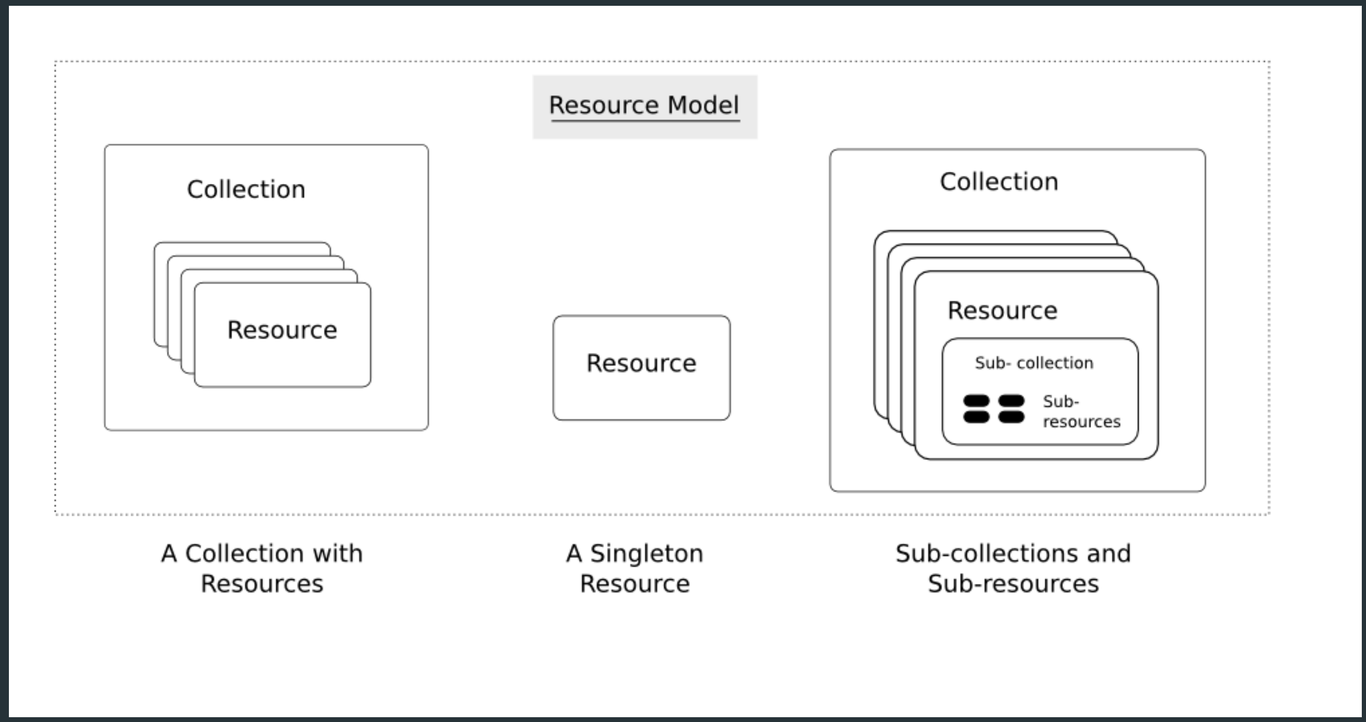
What are collections?

- Definition of collections
- Common examples of collections
- Why collections are important in web services



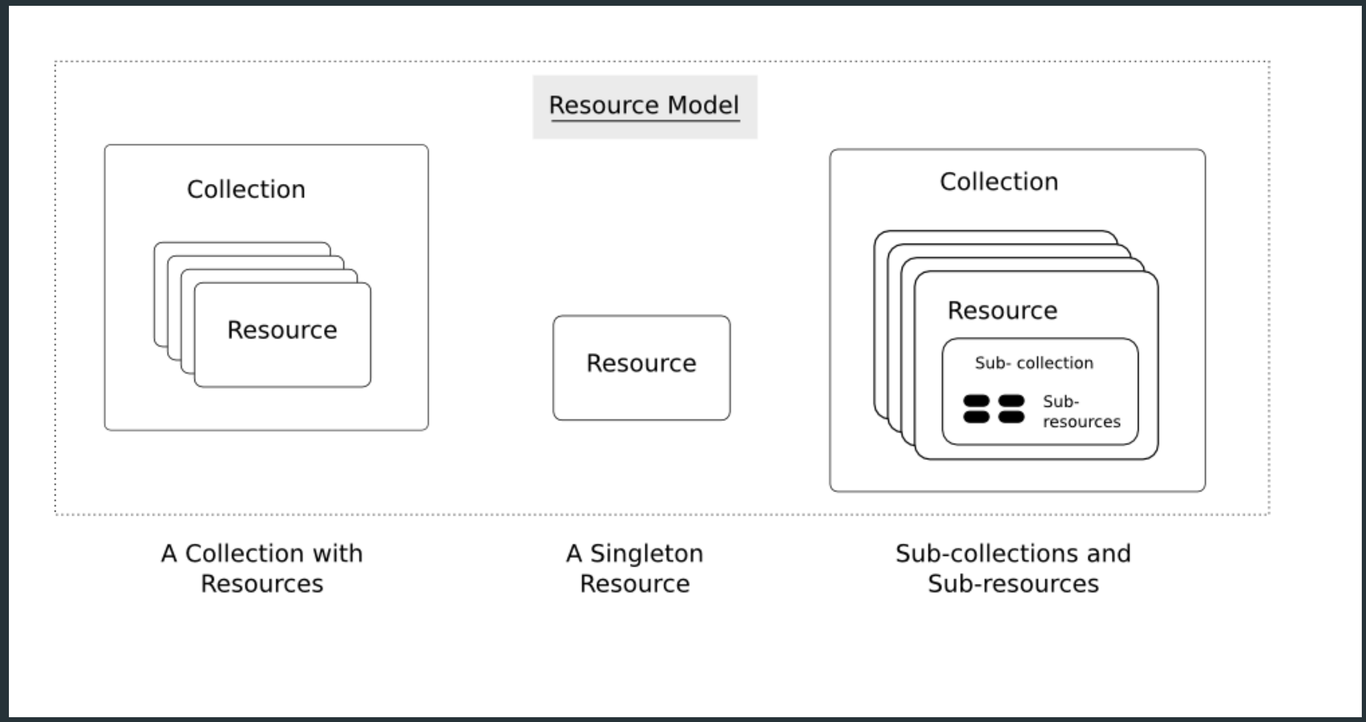
How to design representations of collections

- Choosing a format for representing collections
- Designing URLs for collection resources
- Providing metadata for collections



Example of designing a collection representation in JSON

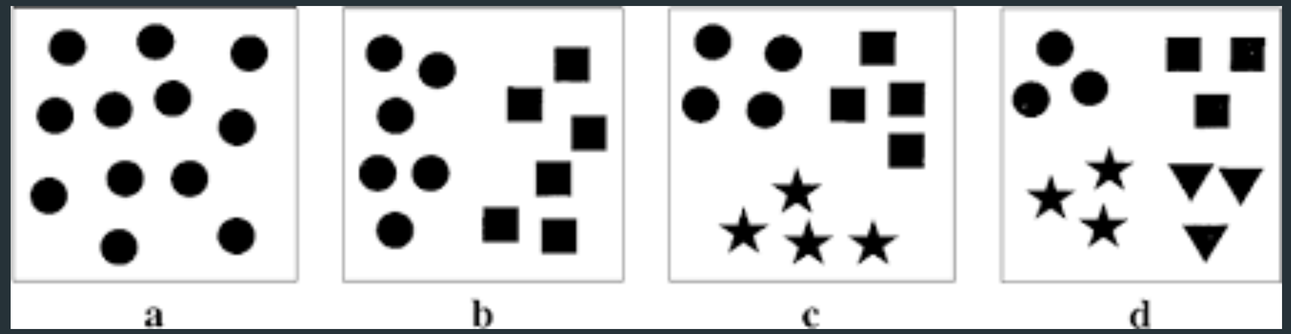
- Example of a collection of products
- Designing URLs for accessing and manipulating the collection resources
- Providing metadata for the collection



```
{
  "products": [
    {
      "id": 1,
      "name": "Product 1",
      "description": "This is the description for Product 1.",
      "price": 9.99
    },
    {
      "id": 2,
      "name": "Product 2",
      "description": "This is the description for Product 2.",
      "price": 19.99
    },
    {
      "id": 3,
      "name": "Product 3",
      "description": "This is the description for Product 3.",
      "price": 29.99
    }
  ]
}
```

What Does it Mean to Keep Collections Homogeneous?

- Definition of homogeneous collections
- Benefits of homogeneous collections
- Consequences of non-homogeneous collections



Strategies for Keeping Collections Homogeneous

- Standardize data fields and data types
- Use common formatting and naming conventions
- Validate data inputs


```
[
  {
    "id":1,
    "name":"Product 1",
    "description":"This is product 1",
    "price":10.99,
    "quantity":5
  },
  {
    "id":2,
    "name":"Product 2",
    "description":"This is product 2",
    "price":7.99,
    "quantity":10
  },
  {
    "id":3,
    "name":"Product 3",
    "description":"This is product 3",
    "price":4.99,
    "quantity":2
  }
]
```

Binary Data in JSON

- Explanation of what binary data is
- Challenges of encoding binary data in representations
- Examples of encoding binary data in JSON and XML

```
{
  "message": "aGVsbG8gd29ybGQ="
}
```

Why HTML Representations?

- Advantages of using HTML representations for web content
- Comparison with other types of representations
- Examples of web services that use HTML representations effectively

```
<!DOCTYPE html>
<html>
 <head>
  <title>My Blog Post</title>
 </head>
 <body>
  <h1>My Blog Post</h1>
  <p>This is my first blog post. I hope you enjoy it!</p>
  <ul>
   <li><strong>Author:</strong> John Doe</li>
   <li><strong>Published:</strong> January 1, 2022</li>
  </ul>
 </body>
</html>
```

When to Serve HTML Representations

- Factors to consider when deciding to serve HTML representations
- Examples of scenarios where HTML representations are appropriate
- Best practices for serving HTML representations

```
<!DOCTYPE html>
<html>
 <head>
  <title>My Blog Post</title>
 </head>
 <body>
  <h1>My Blog Post</h1>
  <p>This is my first blog post. I hope you enjoy it!</p>
  <ul>
   <li><strong>Author:</strong> John Doe</li>
   <li><strong>Published:</strong> January 1, 2022</li>
  </ul>
 </body>
</html>
```

How to Serve HTML Representations

- Explanation of different approaches to serving HTML representations (e.g. server-side rendering, client-side rendering, hybrid rendering)
- Pros and cons of each approach
- Example code for each approach

```
<!DOCTYPE html>
<html>
 <head>
  <title>My Blog Post</title>
 </head>
 <body>
  <h1>My Blog Post</h1>
  <p>This is my first blog post. I hope you enjoy it!</p>
  <ul>
   <li><strong>Author:</strong> John Doe</li>
   <li><strong>Published:</strong> January 1, 2022</li>
  </ul>
 </body>
</html>
```

Server-Side Rendering

```
const http = require('http');
const fs = require('fs');

const server = http.createServer((req, res) => {
  fs.readFile('index.html', (err, data) => {
    if (err) {
      res.writeHead(404, {'Content-Type': 'text/html'});
      res.write('404 Not Found');
    } else {
      res.writeHead(200, {'Content-Type': 'text/html'});
      res.write(data);
    }
    res.end();
  });
});

server.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

Client-Side Rendering

```
import React from 'react';
import ReactDOM from 'react-dom';

const App = () => {
  return <h1>Hello, world!</h1>;
};

ReactDOM.render(<App />, document.getElementById('root'));
```

Hybrid Rendering

```
import React from 'react';
import ReactDOM from 'react-dom';

const App = () => {
  return <h1>Hello, world!</h1>;
};

ReactDOM.hydrate(<App />, document.getElementById('root'));
```

Common error response formats


```
{
  "errors": [
    {
      "status": "404",
      "title": "Not Found",
      "detail": "The requested resource could not be found"
    }
  ]
}
```

Best practices for returning errors

- Use standard HTTP error codes

```
{
  "errors": [
    {
      "status": "404",
      "title": "Not Found",
      "detail": "The requested resource could not be found"
    }
  ]
}
```

Best practices for returning errors

- Use standard HTTP error codes
- Return a detailed error message in the response body

```
{
  "errors": [
    {
      "status": "404",
      "title": "Not Found",
      "detail": "The requested resource could not be found"
    }
  ]
}
```

Best practices for returning errors

- Use standard HTTP error codes
- Return a detailed error message in the response body
- Use a consistent error format

```
{
  "errors": [
    {
      "status": "404",
      "title": "Not Found",
      "detail": "The requested resource could not be found"
    }
  ]
}
```

Best practices for returning errors

- Use standard HTTP error codes
- Return a detailed error message in the response body
- Use a consistent error format
- Avoid returning sensitive information in error messages

```
{
  "errors": [
    {
      "status": "404",
      "title": "Not Found",
      "detail": "The requested resource could not be found"
    }
  ]
}
```

Best practices for returning errors

- Use standard HTTP error codes
- Return a detailed error message in the response body
- Use a consistent error format
- Avoid returning sensitive information in error messages
- Provide documentation for error handling

```
{
  "errors": [
    {
      "status": "404",
      "title": "Not Found",
      "detail": "The requested resource could not be found"
    }
  ]
}
```

Error Handling Strategies

- Fail fast, fail hard
- Return detailed error messages
- Use HTTP status codes

```
{
  "errors": [
    {
      "status": "404",
      "title": "Not Found",
      "detail": "The requested resource could not be found"
    }
  ]
}
```

Fail Fast, Fail Hard

- Advantages of failing fast
- How to fail fast

```python
from flask import Flask, abort, request

app = Flask(__name__)

@app.route('/some-endpoint')
def some_endpoint():
    if not request.args.get('query'):
        abort(400, 'Query parameter is required')
    # continue processing request
```

Detailed Error Messages

- Advantages of detailed error messages
- How to return detailed error messages

```
HTTP/1.1 400 Bad Request
Content-Type: application/json

{
  "error": {
    "code": "missing_query_param",
    "description": "The 'query' parameter is required"
  }
}
```

Lecture outcomes

- Restful
  - Identifying Resources
  - Designing Representations