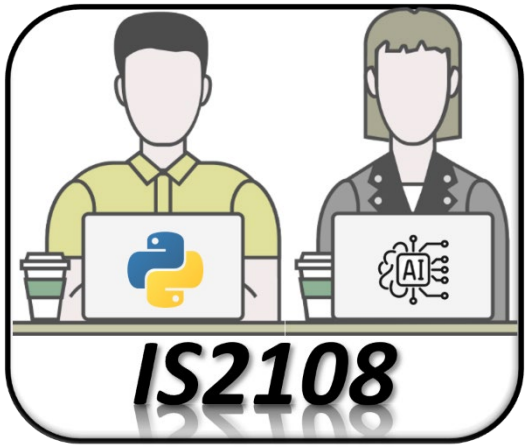


Lecture 8

API Endpoint Development with Django Rest Framework

IS2108 – Full-stack Software Engineering for AI Solutions I
AY 2025/26 Semester 1

Lecturer: A/P TAN Wee Kek

Email: distwk@nus.edu.sg :: **Tel**: 6516 6731 :: **Office**: COM3-02-35

Consultation: Tuesday, 12 pm to 2 pm. Additional consultations by appointment are welcome.

Learning Objectives

- At the end of this lecture, you should understand:
  - What are REST web services.
  - REST principles.
  - JSON format.
  - Creating API endpoint with Django Rest Framework.
  - Best Practices in REST API Design.

Readings

- Required readings:
  - None.
- Suggested readings:
  - None

Introduction to REST Web Services

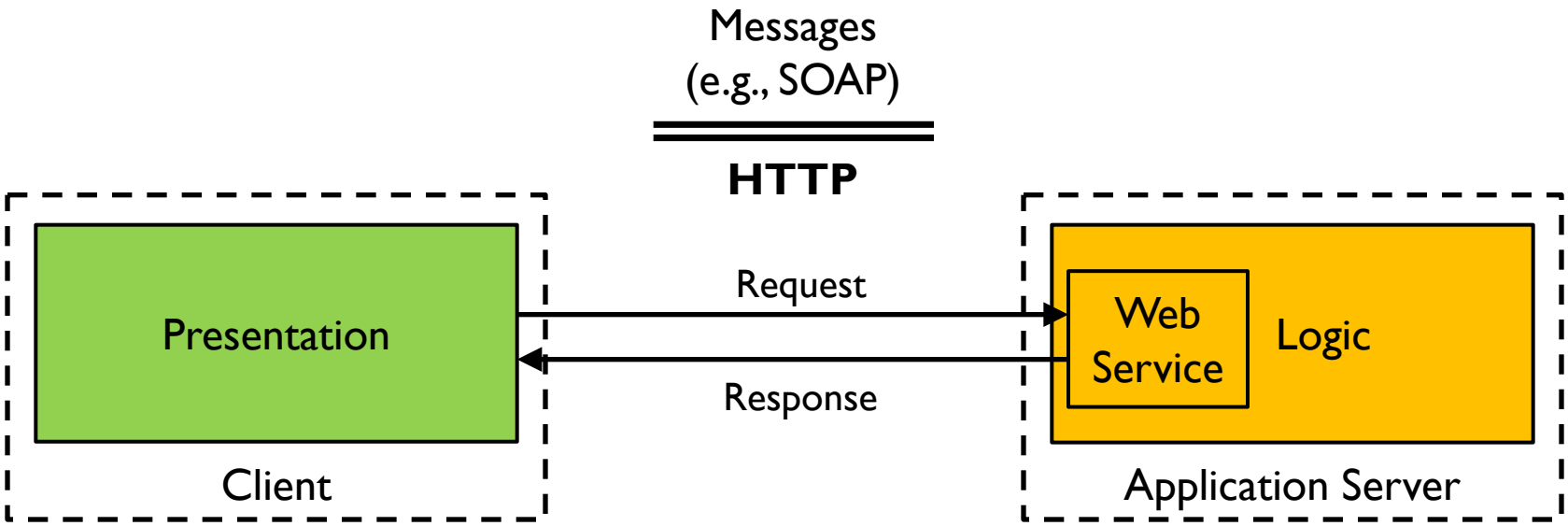
Recap on Service Oriented Architecture (SOA)

- **SOA** is a software architecture emphasising:
  - Software components providing services to other components by exchanging messages via a standard network communications protocol (e.g., Simple Object Access Protocol or SOAP).
  - In practice, a service is an interface definition that may list several discrete services/operations that are semantically related.
  - A service following a standard protocol implemented with any technology can be consumed by another software element implemented with any other technologies.

Recap on Service Oriented Architecture (SOA) (cont.)

- SOA is typically implemented using the **web services** approach:
  - A web service is a service that is offered by one software element to another via communicating over the World Wide Web (WWW).
  - Machine-to-machine communication is enabled by protocols such as HyperText Transfer Protocol (HTTP).
  - More specifically, messages are sent and received over HTTP.

Recap on Service Oriented Architecture (SOA) (cont.)



Web Services at a Technical Level

- On a technical level, web services can be implemented in two ways.
- **"Big" or SOAP web services**:
  - Use XML messages that follow the **Simple Object Access Protocol (SOAP)** standard.
  - SOAP is an XML language defining a message architecture and message formats.
  - Uses a machine-readable description of the operations offered by the service, written in the **Web Services Description Language (WSDL)**.
  - WSDL is an XML language for defining interfaces.

Web Services at a Technical Level (cont.)

- **REST web services**:
  - More suitable for basic, ad hoc integration scenarios.
  - Better integrated with **HTTP** than SOAP-based services.
  - Do **NOT** require SOAP-formatted XML messages or WSDL service-API definitions.
- Comparison of the technical characteristics between software components and software services:

Dimension	Software Components	SOAP Web Services	REST Web Services
Communication Mechanism	Method Invocation	Message Sending	
Method Definitions	Java Interface	WSDL	HTTP Methods
Message Format	N/A	SOAP	JSON/XML String
Transport Protocol	N/A	HTTP	

More about REST Web Services

- REST web services exhibit several desirable characteristics:
  - Use standard HTTP methods to manipulate resources:
    - Does not need to explicitly define methods for manipulating resources.
  - Loose coupling:
    - Client does not need to follow WSDL.
    - Client attempt a HTTP method and will receive an error status (HTTP 405) if method is not supported.
  - Architectural simplicity.
  - Ease of consumption on client side.

REST Principles

- Resource identification through URI:
  - In a web application, an URI identifies folder/subfolder and file that returns a HTML web page.
  - In a REST web service, an URI identifies an endpoint that returns data.
- Uniform interface for manipulating resources using the standard HTTP verbs/methods:
  - Create – PUT
  - Read – GET
  - Update – POST
  - Delete – DELETE

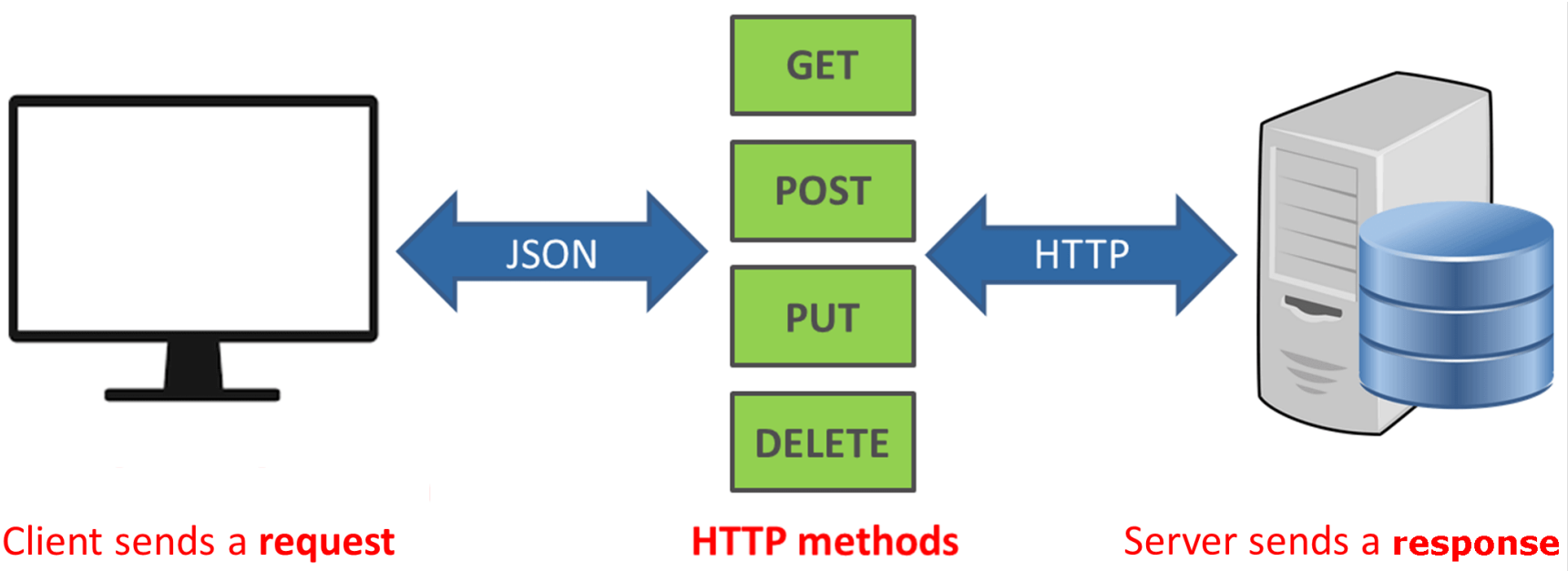
REST Principles (cont.)

- Self-descriptive messages via decoupled resources that can be represented in any string format:
  - Preferred format is JavaScript Object Notation (JSON), which is a plain text format.
  - Alternative is plain text XML (text/xml).
  - In contrast, SOAP formatted messages rely on XML Schema Definition (XSD) (application/xml).
- REST request messages are stateless or self-contained:
  - Every request is considered unique.
  - The REST web service is unable to determine whether multiple requests are from the same client.

REST Principles (cont.)

- Stateful interactions may be achieved by exchanging state, e.g., URI rewriting.

REST Principles (cont.)



Overview of JSON

- JSON or JavaScript Object Notation is one of two commonly used plain text formats for REST web services to exchange messages or data.
- The other format being XML or Extensible Markup Language.
- Between the two formats, JSON is the more popular one for good reasons.
- This example depicts a list of data records typically exchanged with a REST web service:
  - The one on the left is formatted in JSON
  - The one on the right is formatted in XML
  - In this example, we have a list of book records.

Overview of JSON (cont.)

books.json

D: > Temporary > books.json > ...

```json
{
    "books": [
        {
            "bookId": 1,
            "bookName": "Book One"
        },
        {
            "bookId": 2,
            "bookName": "Book Two"
        },
        {
            "bookId": 3,
            "bookName": "Book Three"
        }
    ]
}
```

books.xml

D: > Temporary > books.xml

```xml
<books>
  <book>
    <bookId>1</bookId>
    <bookName>Book One</bookName>
  </book>
  <book>
    <bookId>2</bookId>
    <bookName>Book One</bookName>
  </book>
  <book>
    <bookId>3</bookId>
    <bookName>Book Three</bookName>
  </book>
</books>
```

Similarities Between JSON and XML

- Observe that JSON is similar to XML to the extent that both are self-describing or human readable.
- The two formats also utilise a hierarchical structure or values within value:
  - For example, bookId and bookName are contained within a book.
- Both JSON and XML can be parsed and used by many programming languages.
- JSON and XML formatted data can both be fetched using the `XMLHttpRequest` object, which is supported in most modern web browsers.

Similarities Between JSON and XML (cont.)

- Client-side rendering web applications running within web browsers rely on the `XMLHttpRequest` object to communicate with REST web services running on the server-side.

Differences Between JSON and XML

- However, between JSON and XML, they also have several differences.
- JSON is more concise as it does not use end tag and there is no angle brackets surrounding the attribute names:
  - Consequently, JSON is faster to read and write.
- JSON also supports the array notation.
- JSON can be parsed by a standard JavaScript function whereas XML requires an XML parser.

JSON is the Preferred Format

- JSON is generally preferrable over XML as JSON is easier to parse and readily convertible to JavaScript object:
  - Note that JavaScript is one of the most popular programming languages.
- Other than structurally similar to JavaScript object, it also helps that JSON is similar to the dictionary or `dict` data structure in Python.
- Overall, it is extremely easy to work with JSON in both JavaScript and Python.

JSON is the Preferred Format (cont.)

- In this example:
  - The sample code demonstrates the parsing of the JSON book records into both JavaScript object and Python dict.
  - It also demonstrates the respective conversion back to JSON string.
  - Observe the ease of manipulation JSON across both JavaScript and Python.

JSON is the Preferred Format (cont.)

jsondemo.js

D: > Temporary > jsondemo.js > ...

```
1  var jsonStr = '{ "books": [{"bookId": 1,"bookName": "Book One"}
2  var obj = JSON.parse(jsonStr);
3
4  console.log(typeof(obj));
5  console.log(obj);
6
7  jsonStr = JSON.stringify(obj)
8
9  console.log(typeof(jsonStr));
10 console.log(jsonStr);
11
```

jsondemo.py

D: > Temporary > jsondemo.py > ...

```
1  import json
2
3  jsonStr = '{ "books": [{"bookId": 1,"bookName": "Book One"},{"bool
4  obj = json.loads(jsonStr)
5
6  print(type(obj))
7  print(obj)
8
9  jsonStr = json.dumps(obj)
10
11 print(type(jsonStr))
12 print(jsonStr)
13
```

PROBLEMS OUTPUT TERMINAL PORTS SQL CONSOLE POSTMAN CONSOLE DEBUG CONSOLE

```
D:\Temporary>node jsondemo.js
object
{
  books: [
    { bookId: 1, bookName: 'Book One' },
    { bookId: 2, bookName: 'Book Two' },
    { bookId: 3, bookName: 'Book Three' }
  ]
}
string
{"books":[{"bookId":1,"bookName":"Book One"},{"bookId":2,"bookName":"Book Two"},{"bookId":3,"boo
kName":"Book Three"}]}
```

cmd

```
d:\Temporary>python jsondemo.py
<class 'dict'>
{'books': [{'bookId': 1, 'bookName': 'Book One'}, {'bookId': 2, 'bookName': 'Book T
wo'}, {'bookId': 3, 'bookName': 'Book Three'}]}
<class 'str'>
{"books": [{"bookId": 1, "bookName": "Book One"}, {"bookId": 2, "bookName": "Book T
wo"}, {"bookId": 3, "bookName": "Book Three"}]}

d:\Temporary>
```

cmd
cmd

JSON is the Preferred Format (cont.)

- It is easy to use JSON to represent both input and output data.
- For a GET or retrieve operation, the output data can be returned as JSON string.
- For PUT and POST, or create and update, operations, the input data can also be passed in as JSON string too.

Django Rest Framework

lecture08

Overview of the Django Rest Framework

- **Django Rest Framework (DRF)**:
  - A powerful and flexible toolkit built on top of the Django web framework.
  - Simplifies the process of building web APIs (Application Programming Interfaces).
- Key features of DRF:
  - Expose data and functionality through REST web services instead of HTML web pages.
  - Handle serialization – Convert complex data like Django models into JSON or XML for transmission.
    - Also support non-ORM data.
  - Manage authentication, permissions, and view logic with minimal code.

Overview of the Django Rest Framework (cont.)

- Core components of DRF:

Component	Description
Serializers	• Convert Django model instances to and from JSON, XML, or other formats. • Similar to Django Forms, but for APIs.
Views and ViewSets	• Define API endpoints. • Can use generic views (`APIView`) or `ModelViewSet` to automatically handle CRUD operations.
Routers	• Automatically generate URL routes for viewsets, reducing boilerplate.
Authentication and Permissions	• Built-in support for token-based, session-based, and custom authentication schemes. • Can control access via permission classes.
Browsable API	• DRF includes a web-based interface that allows developers to test API endpoints interactively in the browser.

Overview of the Django Rest Framework (cont.)

- Typical development workflow:
  - Define a model (same as in Django).
  - Create a serializer class to describe how model data is converted to JSON.
  - Create a view or viewset to expose the API endpoints.
  - Map URLs using routers.
  - Test the endpoints via browser, `curl`, or API tools like Postman.

Installing DRF

- DRF requires the followings:
  - Django (>= 4.2 and <= 5.2, we are using 5.2 LTS)
  - Python (>= 3.10)
  - The following packages are optional:
    - Markdown (3.3.0+) – Markdown support for the browsable API.
    - django-filter (1.0.1+) – Filtering support.
- DRF can be installed using `pip`:

```
pip install djangorestframework
pip install markdown
pip install django-filter
```

Setting Up DRF in a Django Project

- Add `'rest_framework'` to `INSTALLED_APPS` of project-level `settings.py`:

```
INSTALLED_APPS = [
    …
    'rest_framework',
]
```

- If using browsableAPI, the login and logout views of DRF should be added to the project-level `urls.py`:

```
urlpatterns = [
    …
    path('api-auth/', include('rest_framework.urls'))
]
```

Setting Up DRF in a Django Project (cont.)

- Any global settings for DRF APIs are kept in a single configuration dictionary named `REST_FRAMEWORK`:
  - This dictionary placed in the project-level `settings.py`.
  - Usually need to add Django's standard `'django.contrib.auth'` permissions or allow read-only access for unauthenticated users:

```
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.
            DjangoModelPermissionsOrAnonReadOnly'
    ]
}
```

Creating a Serializer Class

- A **Serializer** is used to serialize and deserialize data instances into string representation such as JSON:
  - Serializer is similar to Django's forms, which serialize/deserialize data into/from HTML input elements.
  - Serializers are typically placed in an app-level `serializers.py` file.
- Similar to a ModelForm, we can create a Serializer based on an existing Django model with **ModelSerializer**:
  - Simplify the process of serializing Django models

Creating a Serializer Class (cont.)

- Basic Serializer class using Django model database access API:
  - See `StudentSerializer` for a demonstration with the `Student` model.
  - Observe that with the database access API, the code is generally straightforward but certainly can be reduced further.
- See `StudentModelSerialzier` for a demonstration with `ModelSerializer`:
  - The `model` attribute in the `Meta` class is used to specify the mapping to the required Django model.
  - Observe that the code is more concise.

Creating a Regular Django View with ModelSerializer

- DRF provides various view features, but we will examine the most basic approach of using a Django view:
  - See `student_list` and `student_details` in the app-level `views.py` for the demonstration.
  - `student_list` is used to retrieve a list of all existing students or create a new student.
  - `student_details` is used to retrieve, update and delete a single student.
  - Observe that the PUT method is used for create and POST method for update.
  - The use of the decorator `@csrf_exempt` is required as the data will not be posted from a form that has a CSRF token.

Creating a Regular Django View with ModelSerializer (cont.)

- The views need to be wired to the respective URL in the app-level `urls.py`:
  - Observe that both views are mapped to the URL 'students' but the one for student details contains the path parameter `pk`.
- The API endpoints can be tested using Postman after running the web application:

Creating a Class-based View with ModelSerializer

- It is also possible to create API views using class-based views rather than function-based views similar to Django forms:
  - Allow common functionalities to be reused.
- See `StudentListAPIViewClass` and `StudentDetailsAPIViewClass` in the app-level `views.py` for the demonstration:
  - Observe that the code are similar to those in the function-based views.
  - The class-based views are wired to different URLs for demonstration purpose.

Creating a Class-based View with ModelSerializer (cont.)

- DRF authentication is required:
  - The following command can be used to create a super user:

```
python manage.py createsuperuser
```

  - In Postman, the credential can be provided via Authorization > Basic Auth.
  - We will discuss more about authentication and permissions in the future.

Creating a ModelViewSet with ModelSerializer

- DRF provides ViewSet classes to allow developers to focus on modeling the state and interaction of the API:
  - ViewSet classes are similar to View classes, but they provide automatically generated operations.
- A `ModelViewSet` class provides a complete set of default read and write operations for the underlying model:
  - See `StudentModelViewSet` in the app-level `views.py` for the demonstration.
  - The ViewSet can be bound to URLs explicitly in the app-level `urls.py`:
    - Note that the ViewSet are also wired to different URLs for demonstration purpose.

Creating a ModelViewSet with ModelSerializer (cont.)

- Alternatively, the conventions for wiring up resources into views and urls can be handled automatically using a **Router** class:
  - This can be done by registering the appropriate view sets with a router, in this case a `ModelViewSet`.
  - Recall that our conventions for create and update are PUT and POST, respectively, but the router uses the reverse conventions.
  - See `demos` for the demonstration.

More About Views in DRF

- DRF provides various view features:
  - `ListAPIView` – Use for implementing a read-only list endpoint.
  - `ListCreateAPIView` – Use for implementing a read-only list endpoint with POST (create).
  - `ModelViewSet` – Use for implementing an endpoint with full CRUD set under one class as discussed earlier.
- More about `ListAPIView`:
  - A generic class-based view that provides a read-only endpoint to list a collection of objects (HTTP GET only).
  - Implements get() – Returns a paginated list of serialized objects.

More About Views in DRF (cont.)

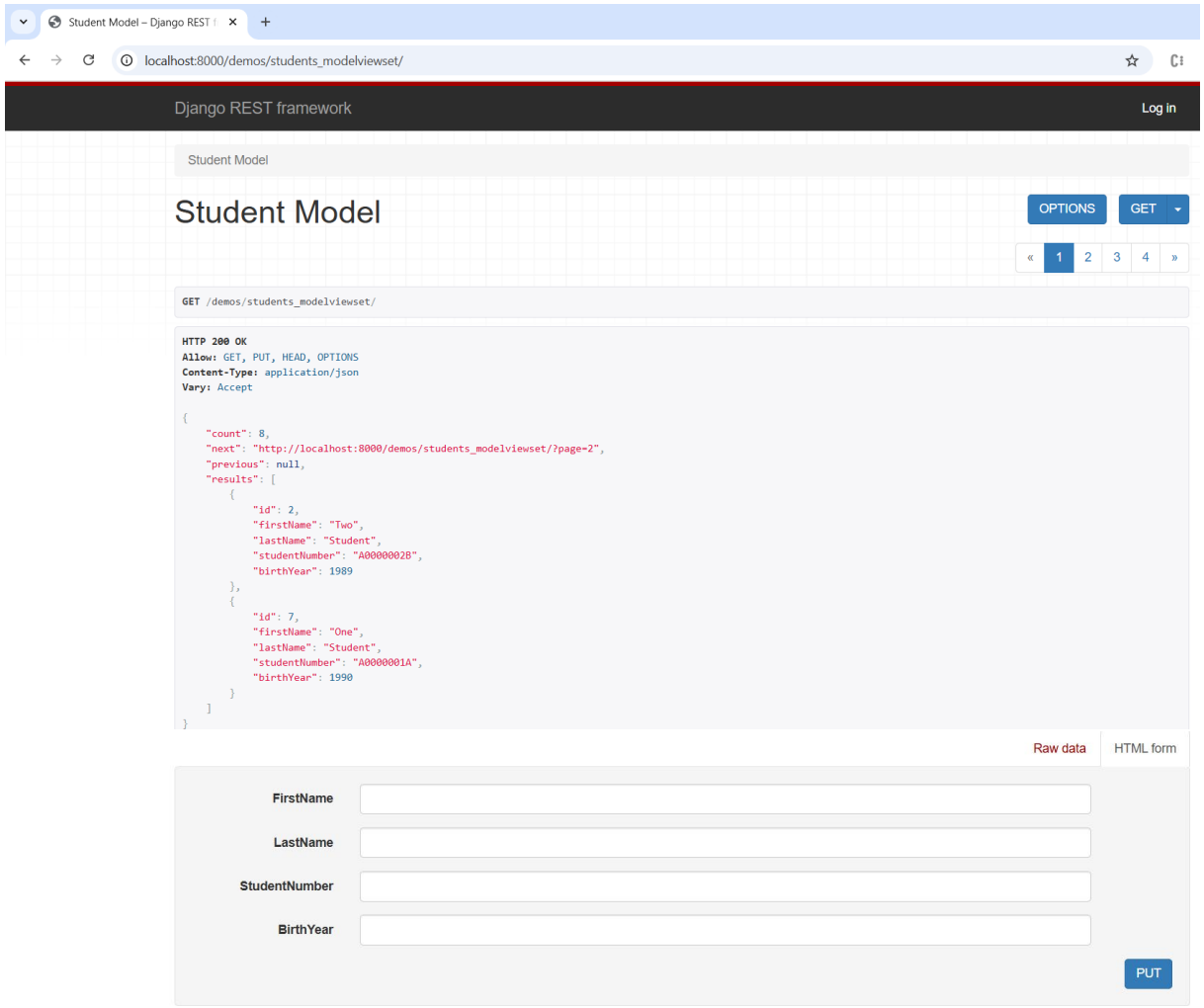
- Need to define `queryset` and `serializer_class`.
- Integrates with DRF's pagination, filtering, search, ordering, permissions, throttling.

- See `StudentListAPIView` in the app-level `views.py` for the demonstration:
  - Pagination:
    - `PageNumberPagination` – ?page=1
    - Configuration of default pagination class done in `settings.py`
  - Searching:
    - `?search=Two`
  - Ordering:
    - Ascending – `?ordering=firstName`
    - Descending – `?ordering=-firstName`

The Browsable API

- The Browsable API:
  - Provides support for generating human-friendly HTML output for each resource when the HTML format is requested.
  - These pages allow for easy browsing of resources.
  - They also provide forms for submitting data to the resources using POST, PUT, and DELETE.
- Any DRF view with a `serializer_class` will render a form when viewed from a web browser:
  - See `StudentModelViewSet` via a web browser at the URL `students_modelviewset/` for the demonstration

The Browsable API (cont.)



Best Practices in REST API Design

The Importance of Well-Designed API Endpoints

- API endpoints play a critical role in enabling communication between frontend software applications and the backend.
- Thus, REST web services must be thoughtfully designed and implemented.
- There are some good practices that you should take into consideration when creating your own REST web services.
- At a high-level, well-designed API should exhibit three characteristics.

Three Basic Characteristics of Well-Designed API Endpoints

- First, API should be easy to read and work with:
  - The resources to be manipulated and their associated operations should be intuitive to the developers.
  - Naming your API endpoints appropriately with the corresponding HTTP methods goes a long way in enhancing the usability.
- Second, API should adopt a defensive or failsafe design:
  - Make it easy for developers to integrate into their own code and avoid incorrect or erroneous usage.
  - Important to provide informative and erroneous feedback.

Three Basic Characteristics of Well-Designed API Endpoints (cont.)

- For example:
  - Input data validation and business rules checking.
  - These should be done in conjunction with appropriate HTTP status codes and meaningful messages.
  - Help developers to consume your API productively and correctly.
- Third, API should be complete and concise:
  - A complete API should enable developers to build full-fledged applications using the data exposed by your endpoints.
  - API completeness takes place over time, and it is more likely that you will incrementally build on top of existing API endpoints.

Documentation

- Maintaining a comprehensive documentation of your API is also a good practice:
  - Ease the learning curve of new and experienced developers alike.
- DRF provides the Browsable API for documenting API.
- You can also use API design tool such as Swagger to define your API:
  - This can then be used to generate the corresponding documentation automatically.
  - In Python, Connexion builds on top of Flask and uses Swagger to create API endpoints.

Error Handling in API Endpoints

- The HTTP protocol provides many status codes for good reasons:
  - Returning a more specific status code rather than a generic one helps the developer consuming your endpoints to take appropriate follow-up runtime actions.
  - Also useful for troubleshooting their own code during design time.
- This table shows some of the more commonly used HTTP status codes:
  - Observe that for successful requests, you should be using a 200 series status code and there are at least three such codes for different purposes.

Error Handling in API Endpoints (cont.)

- For erroneous conditions, you should be using a 400 or 500 series code as appropriate.

Status Code	Category	Description
200	Success	OK: The request has succeeded.
201	Success	Created: The request has been fulfilled, and a new resource is created.
204	Success	No Content: The server successfully processed the request but returns no content.
400	Client Error	Bad Request: The server could not understand the request due to invalid syntax.
401	Client Error	Unauthorized: Authentication is required and has failed or not been provided.
403	Client Error	Forbidden: The server understood the request but refuses to authorize it.
404	Client Error	Not Found: The server cannot find the requested resource.
405	Client Error	Method Not Allowed: The HTTP method is not allowed for the requested resource.
500	Server Error	Internal Server Error: The server encountered an unexpected condition.
501	Server Error	Not Implemented: The server does not support the functionality required to fulfill the request.
504	Server Error	Gateway Timeout: The server did not receive a timely response from an upstream server.

Optimising Performance of API Endpoints

- In terms of optimising the performance of API endpoints, you need to broadly differentiate between **read** and **write** operations.
- **Read operations**:
  - Make provision to retrieve one or few records based on well-defined filtering criteria.
  - Operations for retrieving records in large quantity, or worst still retrieving all records, should be avoided as the user likely only requires a small subset of records for immediate use.
  - Well-designed retrieval operations should facilitate record pagination with an appropriate page size.

Optimising Performance of API Endpoints (cont.)

- Handling relationships or associations among records or classes:
  - Avoid fetching unnecessary relationship attributes.
  - In general, fetching of to-one relationships is fine but fetching of to-many relationships should be avoided unless the user really requires those relationship data.
  - It is generally preferable to create separate API endpoints and operations for fetching relationship attributes.

Optimising Performance of API Endpoints (cont.)

- **Write operations**:
  - For create, update and delete, you should consider providing support for baulk write operations to avoid the need of making excessive number of API calls.
  - For example:
    - To delete multiple records.
    - Pass in a list of record identifiers in a single operation call instead of making multiple operation calls, each taking in only one identifier for deletion.

Input Data Validation

- Important to perform comprehensive input data validation and business rule checking:
  - Ensure that the input data passed in from the user are correct and appropriate.
- If your API is opened to external parties:
  - You should not assume that they will perform the validation or understand how to validate the data correctly.
  - Incorrect or inappropriate input data will lead to unnecessary and inefficient processing at the downstream node.
  - For example:
    - Input data is of an incorrect data type or length – Should not attempt to insert the erroneous data into the database.
    - Waste of processing resources.

Securing API Endpoints

- It is important to secure your API endpoints from unauthorised users.
- Standard security techniques:
  - JSON web token
  - Hashing of password
  - Encryption of confidential data
  - These should be implemented according to your data security requirements.
- API endpoints that are opened to external applications:
  - Guard against unauthorised applications with an API or application key mechanism.

Summary

- Web services can be implemented as "big" SOAP web services or "lightweight" REST web services.
- JSON is the preferred format for exchanging data over REST web services.
- Django Rest Framework (DRF) can be used to create API using REST web services.
- The development workflow of DRF is very similar to that of developing web application with Django.
- The use of ModelSerializer and ModelViewSet simplifies the creation of API.

Summary (cont.)

- When creating REST web services, it is useful to consider some good practices.

Q&A

Next Lecture…

- Learn about:
  - Overview of the Flask framework.
  - Creating web application with Flask.

In-class Formative Quiz for Lecture 08