Lecture 1 - Introduction to FastAPI & Development Setup

Ményssa Cherifa-Luron

2024-10-11

TL;DR:

If you're short on time, you can listen to the key takeaways from the course in this audio summary:

Listen to the Audio Overview:

Download audio

Together, we'll dive into key concepts like asynchronous programming, data validation, and API deployment—helping you sharpen your skills every step of the way :

- **APIs** (Application Programming Interfaces) are crucial for allowing different software systems to communicate, making modern apps smarter and more interconnected.
- **APIs serve as the link between front-end and back-end systems**, enabling seamless data exchange across various platforms like mobile apps, IoT devices, and desktop applications.
- **HTTP** is the protocol for web communication, using requests and responses with status codes to manage data exchange.
- **RESTful services** use HTTP methods to interact with resources, offering simplicity, flexibility, and scalability for web services.
- **URLs** are essential for locating resources on the web, comprising components like scheme, host, path, and query parameters.
- **FastAPI** stands out with asynchronous programming, automatic documentation, and type hinting, making it fast and suitable for API-heavy projects.

Welcome to **FastAPI 101**, where we're diving into one of the fastest-growing frameworks for building APIs—FastAPI!

If you've ever worked with REST APIs and thought, "There's got to be a faster, easier way," then you're in the right place.

FastAPI is a modern, fast (high-performance), web framework for building APIs with Python based on standard Python type hints. It is designed to be lightning-fast (hence the name) support for automatic documentation.

In this course, we'll get hands-on, from understanding how APIs make the modern web tick to setting up your own FastAPI environment. Whether you're new to APIs or have some experience, by the end, you'll be running your own FastAPI server and ready to build dynamic, data-driven apps.

Let's get you up to speed!

1. What the Heck is an API, and Why Should You Care?

An **API (Application Programming Interface)** is the glue that holds our digital world together. It's the bridge that lets apps and services exchange data seamlessly.

Think of it like sending a **request** to your favorite delivery app for a meal—APIs ensure your order is understood, processed, and delivered to the kitchen (and then to you). From social media apps to payment gateways, APIs power the connections that make modern software work.

More precisely, **an API is a set of protocols, tools, and definitions that allow different software applications to communicate with each other. It defines how requests are made between clients (users or systems) and servers (service providers) and ensures data is transferred in a structured way.**

Why are APIs so essential?

- They let apps **talk** to each other, allowing you to share, pull, and push data without breaking a sweat.
- They're the reason your **phone app** syncs with your **watch**—and why your boss can track your productivity tools. (Oops.)
- Modern apps are like **Legos**—modular, flexible, and built using APIs to connect their different parts.

Why APIs are crucial for data and cybersecurity experts?

APIs are the digital bridges that allow different software systems to communicate and share data.

- For **data experts**, they open up a world of possibilities by enabling the integration of diverse datasets, automating workflows, and enhancing data-driven insights. Leveraging APIs allows you to seamlessly incorporate real-time data from various sources, enriching your analyses and models with up-to-date insights.
- For **cybersecurity professionals**, understanding APIs is crucial for safeguarding digital assets. As APIs become more prevalent, they also become potential targets for cyber threats. Learning how to build and secure APIs ensures that sensitive data is protected and that your systems remain resilient against attacks.

Whether you're a data scientist looking to enhance your analytical capabilities or a cybersecurity expert aiming to fortify your defenses, this course will introduce you to the knowledge and tools you need to succeed.

2. The Importance of APIs in Client-Server Architecture

In the world of applications, it's essential to distinguish between the **Front-End** and the **Back-End**:

- **Front-End**: This is what users interact with—think of it as the stylish storefront of your application. **It includes everything the user sees and engages with, from buttons to text.** Front-end development focuses on user experience and design.
- **Back-End**: This is the engine that powers the application. **It handles data storage, business logic, and server-side functionality.** Users don't see it, but without the Back-End, nothing would work!

Now, here's where APIs come into play. **They are the vital links that connect these two worlds.** APIs allow the Front-End to request and send data to the Back-End without needing to know how the Back-End operates.

But wait! **APIs aren't just for web services**—they're the backbone of various applications. Here are a few examples:

- **Mobile Apps**: Think about how your favorite weather app gets real-time data. It communicates with a weather service API to fetch the latest forecasts and display them beautifully on your screen.
- **IoT Devices**: Smart home devices, like thermostats or lights, use APIs to send and receive data. For instance, a smart thermostat might send temperature readings to a cloud service via an API, allowing you to control it remotely through a mobile app.
- **Desktop Applications**: Applications like Microsoft Word can use APIs to integrate with online services. For example, you might use an API to fetch stock photos from a service like Unsplash directly within the app, streamlining your workflow.

- **Third-Party Integrations**: E-commerce platforms often integrate payment gateways (like Stripe or PayPal) using APIs. When you check out, the platform calls the payment API to process your transaction securely.

This separation enables multiple clients—like web apps, mobile apps, desktop applications, and even IoT devices—to communicate with the same Back-End services. **This means you can build a seamless experience across platforms, ensuring that all your applications can tap into the same data and functionality.**

3. HTTP Basics: It's Like a Postman for the Web

HTTP might sound fancy, but think of it as the mail service of the web—delivering packages (requests) from clients (you) to servers (your favorite app).

It's like sending a letter to a friend.

You write the letter **(create a request)**, put it in an envelope **(format it)**, address it **(specify the URL)**, and send it off **(make the request)**. The server gets your letter and responds with a package **(response)** containing your requested information.

3.1. Requests in the Tech World

> Requests are all about getting what you want, with a sprinkle of charm and a dash of politeness.

- **The Friendly Ask**: Picture this—you're at a dinner party, and you spot the salad across the table. You lean in and say, "Could you please pass the salad? It looks amazing!" That's a request. It's all about expressing your desires with a touch of grace and a hint of enthusiasm.
- **The Formal Plea**: Now, let's get a bit official. You're applying for that dream library card. You fill out a form, scribble your details, and voilà! You've made a formal request. It's like sending a little note to the universe saying,

  "I'm ready for all the books!"

In the tech world, requests are the stars of the digital show. When you type a website URL into your browser and hit enter, you're basically saying,

> "Hey server, can you show me this awesome webpage?"

Your browser sends a request to the server, and the server—like a gracious host—returns with the webpage you asked for, often with a little note on how things went down (that's the HTTP status code, see after).

Requests are everywhere, from the dinner table to the internet, keeping everything moving along with a polite ask every step of the way.

3.2. HTTP Status Codes

Like life, requests have outcomes. Some are good, some are bad, and some make you want to throw your laptop out the window.

HTTP status codes are like the traffic signals of the web, guiding the flow of data between clients and servers. They indicate the outcome of an HTTP request, helping developers and users understand what happened during the interaction.

Here's a deeper dive into some of the most common and important HTTP status codes:

1xx: Informational Responses

These codes indicate that the request was received and understood, and the process is continuing.

- **100 Continue**: The server has received the request headers, and the client should proceed to send the request body.

2xx: Success

These codes indicate that the request was successfully received, understood, and accepted.

- **200 OK**: The request was successful, and the server returned the requested resource.
- **201 Created**: The request was successful, and a new resource was created as a result.

3xx: Redirection

These codes indicate that further action needs to be taken by the user agent to fulfill the request.

- **301 Moved Permanently**: The requested resource has been permanently moved to a new URL.
- **302 Found**: The requested resource is temporarily located at a different URL.

4xx: Client Errors

These codes indicate that the client seems to have made an error.

- **400 Bad Request**: The server could not understand the request due to invalid syntax.
- **401 Unauthorized**: The request requires user authentication.
- **403 Forbidden**: The server understood the request but refuses to authorize it.
- **404 Not Found**: The server can't find the requested resource. This is often the result of a broken link or a mistyped URL.

5xx: Server Errors

These codes indicate that the server failed to fulfill a valid request.

- **500 Internal Server Error**: The server encountered an unexpected condition that prevented it from fulfilling the request.
- **502 Bad Gateway**: The server, while acting as a gateway or proxy, received an invalid response from the upstream server.
- **503 Service Unavailable**: The server is not ready to handle the request, often due to maintenance or overload.

Understanding these codes is crucial for debugging and improving user experience.

For instance, too many 404 errors might indicate broken links that need fixing, while frequent 500 errors could suggest server-side issues that require attention.

By monitoring these codes, developers can ensure smoother and more reliable web interactions.

For a comprehensive list of HTTP status codes, you can refer to resources like Wikipedia or detailed guides on developer platforms.

To wrap it all up, the **Request/Response Cycle** works like this:

- **Knock, Knock!**: You (the client) send a request to the server, similar to knocking on the door of your favorite restaurant, asking for your meal.
- **Server Response**: The server opens the door and either provides the data (response) or, on a not-so-great day, might say, "Not today" with an error message like 404 Not Found.

3.3. Role of HTTP Verbs in RESTful Web Services

HTTP verbs, also known as HTTP methods, are integral to **RESTful** web services. They define the actions that can be performed on resources available on a server, forming the backbone of how clients and servers communicate over the web.

Let's explore how these verbs contribute to the functionality and efficiency of RESTful services.

What is a RESTful Service?

A **RESTful service** is a web service that follows the principles of REST (Representational State Transfer), an architectural style for designing networked applications.

Let's explore the **core principles of RESTful services** and illustrate each concept.

1. **Resource-Based**

In REST, everything is treated as a resource, identified by a unique URI (Uniform Resource Identifier). Consider an online bookstore. Each book can be a resource with a URI like `https://api.bookstore.com/books/123`, where `123` is the unique identifier for a specific book.

2. **Stateless Communication**

RESTful services are stateless, meaning each request from a client to a server must contain all the information needed to understand and process the request.

When you log into a website, each request you make (like viewing your profile) includes your authentication token. The server doesn't remember your login state between requests; it relies on the token you provide each time.

3. **Use of Standard HTTP Methods**

RESTful services use standard **HTTP methods** to perform operations on resources.

- **GET**: Retrieve a list of books with `GET https://api.bookstore.com/books`.
- **POST**: Add a new book with `POST https://api.bookstore.com/books` and include the book details in the request body.
- **PUT**: Update book information with `PUT https://api.bookstore.com/books/123` and provide the updated data.
- **DELETE**: Remove a book with `DELETE https://api.bookstore.com/books/123`.

Get more details on method here

4. **Representation-Oriented**

Resources can be represented in various formats, such as **JSON, XML, or HTML**. When you request a book's details, you might receive the data in JSON format:

```
    {
       "id": 123,
       "title": "RESTful Web Services",
       "author": "John Doe",
       "price": 29.99
}
```

5. **Statelessness and Scalability**

The stateless nature of RESTful services allows them to scale easily. An e-commerce site can handle thousands of simultaneous users because each request is independent, allowing the server to distribute requests across multiple servers without maintaining session state.

6. **Cacheable Responses**

Responses from RESTful services can be cached to improve performance. A news website might cache the response for the latest headlines, so repeated requests for the same data can be served quickly without hitting the server again.

Benefits of RESTful Services

- **Simplicity and Flexibility**: RESTful services are straightforward to use and flexible, allowing developers to build APIs that can be easily consumed by different clients, such as web browsers, mobile apps, and other servers.
- **Interoperability**: By using standard HTTP methods and formats, RESTful services can be accessed by any client that understands HTTP, making them highly interoperable.
- **Scalability**: The stateless nature of RESTful services allows them to scale horizontally, handling more requests by adding more servers.

3.4. REST, SOAP, and GraphQL: The Web's Favorite Squabble

RESTful services have become a popular choice for building APIs due to their simplicity, scalability, and ability to integrate seamlessly with the web's existing infrastructure.

Building APIs can be like debating pizza toppings—REST, SOAP, and GraphQL are all different flavors:

- **REST** is the most popular, like classic **pepperoni** pizza. Simple, reliable, and everyone loves it.

- **SOAP** is for the serious types, like a fancy **deep-dish**. It's packed with structure, uses a lot of XML, and is great for enterprises that need bulletproof security.
- **GraphQL** is for the data nerds, a **custom pizza** where you choose exactly what you want on it. No more, no less. It's flexible and efficient for complex data retrieval.

If REST is the crowd-pleaser, SOAP is for the suit-and-tie folks, and GraphQL is the cool kid shaking things up with customization.

No matter which flavor of API you prefer—be it REST, SOAP, or GraphQL—URLs are the essential connectors that enable these technologies to function. They act as the precise coordinates that direct your API calls to the appropriate destinations on the internet, ensuring efficient and accurate data retrieval and interaction.

4. URL Construction: What is a URL?

A **URL** (Uniform Resource Locator) is essentially the **address** used to locate resources on the web. Just as a physical address directs you to a building or a house, a URL tells your browser or application where to find a specific resource, be it a webpage, image, or an API endpoint.

When working with APIs, URLs are crucial because they serve as the precise **coordinates** that direct API calls to the right destinations. Whether you're making a simple GET request or a complex data manipulation with POST, the URL is where it all starts.

4.1. Components of a URL

A URL is made up of several key components, each serving a unique purpose. Understanding these parts helps in correctly constructing and troubleshooting API requests.

1. Scheme (or Protocol)

This defines the protocol used to access the resource. The two most common schemes are `http` (Hypertext Transfer Protocol) and `https` (Hypertext Transfer Protocol Secure). The `https` scheme is preferred over `http` because it encrypts the data being transmitted, ensuring a secure connection.

2. Host

The **host** specifies the domain name or IP address of the server that is hosting the resource. It essentially tells your browser where to send your request.

Example: `www.example.com` In the context of APIs, the host is often the base URL of the API provider, such as `api.openweathermap.org` or `api.example.com`.

3. Path

The **path** points to the specific resource on the server. It acts like the directory structure in a file system, guiding the server to a particular location.

Example: `/products`

In RESTful APIs, the path usually represents the endpoint or resource being accessed, such as `/users`, `/orders`, or `/products/123`.

4. Query Parameters

Query parameters are key-value pairs appended to the end of the path, prefixed by a `?` symbol. They provide extra information or filtering criteria for the request.

Example: `?category=fruits&sort=price_asc`

Multiple parameters are separated by an `&`. In the above example: - `category=fruits`: Filters products to show only items in the "fruits" category. - `sort=price_asc`: Orders the results by price in ascending order.

5. Fragment Identifier:

This part of a URL points to a specific section within a webpage, often used in HTML documents.

Example: `#section2`

For APIs, this is rarely used, but it's helpful for navigating large documents.

Putting it all together, a complete URL looks like this:

```
https://www.example.com/products?category=fruits&sort=price_asc
```

In this URL:

- **Scheme**: `https://`
- **Host**: `www.example.com`
- **Path**: `/products`
- **Query Parameters**: `?category=fruits&sort=price_asc`

4.2. Constructing a URL with Parameters

When constructing URLs for API requests, query parameters are often used to **filter**, **sort**, or **modify** the returned data.

For example, consider a URL for fetching a list of products filtered by category and sorted by price:

```
GET /products?category=fruits&sort=price_asc
```

Breaking down the query parameters:

- `category=fruits`: Specifies that only products categorized as "fruits" should be returned.
- `sort=price_asc`: Instructs the server to sort the returned products by price in ascending order.

This method of passing parameters is especially useful for creating **dynamic queries** without modifying the underlying codebase.

4.3. Best Practices for URL Construction

1. **Use Descriptive Paths**:
    - Use readable and meaningful paths that describe the resource.
    - **Good**: `/users/{id}/profile`
    - **Bad**: `/u123?details=profile`
2. **Limit the Number of Query Parameters**:
    - Overusing query parameters can make URLs long and difficult to manage.
    - Use query parameters only when filtering or modifying the request.
3. **Always Use HTTPS for Secure Data Transmission**:
    - Never use `http` for sensitive data to prevent data interception.
    - Always opt for `https://` when constructing your base URLs.
4. **Use Proper Encodings for Special Characters**:
    - Special characters (`?`, `&`, `=`, etc.) must be URL-encoded to prevent misinterpretation by the server.
    - Use libraries or built-in functions to safely encode URLs, such as Python's `urllib.parse`.

4.4. Constructing API URLs: Example Scenarios

Let's explore a few common scenarios to see how URLs are constructed for different use cases:

1. Fetching User Profile Data

If you want to retrieve a user's profile data using an API:

```
GET https://api.example.com/users/123/profile
```

Here:

- `https://api.example.com` is the base URL.
- `/users/123/profile` is the path, where `123` is the user ID.

2. Searching for Products in an E-commerce API

To search for laptops priced under $1000:

```
GET https://api.shop.com/products?category=laptops&price_max=1000
```

- The **category** parameter filters by product category (`laptops`).
- The **price_max** parameter restricts results to products under $1000.

3. Paginated Results in a Large Dataset**

To navigate through a paginated list of items:

```
GET https://api.data.com/resources?page=2&limit=20
```

- The **page** parameter specifies the current page (`2`).
- The **limit** parameter defines the number of items per page (`20`).

Incorporating these elements properly ensures your API calls are both efficient and readable, reducing the likelihood of errors and making debugging easier.

URLs are the backbone of API communication. Understanding how to construct them accurately—using appropriate schemes, paths, and query parameters—will help you efficiently interact with web resources and design effective API interactions.

5. Full Architecture of an HTTP Request and Response: A Breakdown

To truly grasp how web applications function, we need to understand the full architecture of an HTTP request and response.

It's like uncovering the inner workings of a conversation between a client (like a web browser) and a server (the one holding the data).

Let's walk through a scenario where a user tries to view their profile information on a web application. This is a simplified yet typical process:

1. The Scenario: User Wants Profile Information

Imagine a user logging into their favorite web app to check their profile. The magic starts when they click on the profile page. This action sends a request to the server, asking it to fetch the profile data.

2. The Client Sends a Request

- **User Action**: The user clicks their profile page.
- **Client Side**: The front end (browser or app) constructs an HTTP request to fetch the necessary data from the server.

3. The Structure of an HTTP Request

Here's what that request might look like:

```
GET /api/users/profile HTTP/1.1
Host: example.com
Content-Type: application/json
Authorization: Bearer <token>
Accept: application/json
```

Each part serves a specific purpose:

- **GET**: This is the method, telling the server we're retrieving information.
- **/api/users/profile**: This is the endpoint—the resource (in this case, the profile) we're trying to access.
- **HTTP/1.1**: This is the version of the HTTP protocol being used.

The headers provide additional context, such as:

- **Host**: Specifies the domain (example.com).
- **Content-Type**: Declares that the data being sent is in JSON format.
- **Authorization**: A token that proves the user is allowed to access the data.
- **Accept**: Specifies that the client expects a JSON response.

4. The Server Processes the Request

Once the server receives this request, it jumps into action:

- **Authentication**: First, it checks the `Authorization` token to ensure the user has permission to access the resource.
- **Data Retrieval**: If the authentication is successful, the server dives into its database to fetch the requested user data.

5. The Server Sends Back a Response

With the requested data in hand, the server responds to the client. Here's an example:

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-cache

{
    "username": "user123",
    "email": "user123@example.com",
    "fullName": "User OneTwoThree"
}
```

This response is made up of:

- **Status Line**:
  - `HTTP/1.1 200 OK` tells the client that the request was successful.
- **Headers**: These give additional information, such as the fact that the response contains JSON data (`Content-Type: application/json`) and that the data should not be cached (`Cache-Control: no-cache`).
- **Response Body**: This holds the actual data—in this case, the user's profile details.

To transition smoothly between these sections, we need to bridge the gap from the technical process of the client receiving the server's response to introducing FastAPI's standout features. Here's a natural transition:

6. The Client Receives the Response

Once the client (the web app) gets the server's response, it processes the information:

- **Parsing**: The client reads the JSON response.
- **Rendering**: The app updates the user interface with the profile details, like the username and email.

The request-response cycle is the bread and butter of web communication, but what if we could supercharge the way our apps handle these interactions? That's where **FastAPI** comes in.

6. FastAPI's Superpowers

FastAPI's key features combine to create a powerful, efficient, and developer-friendly framework for building APIs. Its focus on performance, simplicity, and modern practices makes it an excellent choice for both new and experienced developers.

(image/ stats repos github /documentation)

6.1. FastAPI vs. Flask vs. Django

Let's explore how FastAPI compares with other popular Python web frameworks—Flask and Django—and see why it might be the right choice for your next project.

- **Flask**: Flask is a lightweight framework that offers flexibility and simplicity for building applications quickly. However, it may require additional libraries for larger projects, which can complicate development. It's great for smaller applications or prototypes but lacks the performance optimizations of FastAPI.
- **Django**: Django is a comprehensive framework that includes numerous built-in features, making it suitable for complex, feature-rich applications. However, its size and structure may be overkill for simpler API projects, leading to unnecessary complexity.
- **FastAPI**: FastAPI stands out as a modern framework optimized for performance and efficiency, especially for building APIs. It combines the best of both worlds by being lightweight yet powerful, making it ideal for high-performance applications.

6.2. Key Features

Whether you need high performance for I/O-bound apps or want to build robust APIs with minimal effort, FastAPI delivers with its superpowers of async support, auto-generated docs, and tight integration with Pydantic for data validation.

Feature	Definition	Real-Life Example
Asynchronous Programming	Enables non-blocking operations, allowing multiple tasks to run concurrently without waiting.	A travel booking website retrieves flight and hotel data from various APIs simultaneously, reducing user wait times.

Feature	Definition	Real-Life Example
Automatic Interactive Documentation	Generates API documentation automatically, making it easy to understand and test the API.	A developer building a payment processing API uses FastAPI's auto-generated docs to help external developers integrate quickly.
Request and Response Models with Pydantic	Defines the structure of incoming requests and validates them against specified models.	An online learning platform ensures all course registration requests include valid student names and emails, preventing errors.
Path and Query Parameters	Simplifies the extraction of parameters from the URL, making it easy to handle dynamic requests.	A ride-sharing app allows users to filter their ride history using path and query parameters for user IDs and ride statuses.
Dependency Injection System	Facilitates the management of external dependencies in a clean and modular way.	A blogging platform manages user authentication and database connections through dependency injection, keeping the code organized.
Performance and ASGI Integration	Leverages ASGI for high concurrency and efficient handling of web requests.	A streaming service can handle thousands of concurrent video uploads without delays, ensuring a smooth user experience.
Handling JSON, Headers, and Forms	Provides intuitive methods for extracting and validating JSON data, headers, and form submissions.	An online food delivery app manages order submissions and user feedback through well-structured handling of JSON and forms.
Data Validation with Pydantic	Validates incoming data against defined schemas, ensuring data integrity and reducing errors.	A financial services company verifies that loan applications meet specific criteria (e.g., income, credit score) before processing.

FastAPI stands out in the world of web frameworks, offering a unique blend of speed, efficiency, and user-friendliness.

By leveraging modern Python features, it simplifies API development while ensuring high performance and reliability.

Each key feature—from asynchronous programming to automatic documentation—contributes to a seamless development experience that can adapt to the growing demands of modern applications.

In Summary

APIs: The Digital Bridge

APIs are the essential link between clients and servers, enabling smooth communication. Every time you check your email or log into an app, APIs are handling the requests and responses in the background, ensuring everything works seamlessly.

HTTP: The Web's Mail Carrier

HTTP acts like the postman of the web, delivering requests from the client to the server and bringing responses back, complete with headers, status codes, and content. Mastering HTTP is key to mastering APIs.

FastAPI: The Supercharged Framework

FastAPI takes API development to the next level:

- **Speed & Performance**: Thanks to asynchronous programming with `async/await`, FastAPI can handle high loads of requests efficiently.
- **Auto-Generated Documentation**: FastAPI automatically creates clear and interactive API docs with Swagger UI and ReDoc, so you don't have to.
- **Type Safety & Validation**: By leveraging Python's type hints, FastAPI keeps your code clean, safe, and easy to maintain, while handling data validation without extra hassle.

In short, FastAPI is built for performance, simplicity, and productivity—especially for API-heavy projects.

Now that we've got the essentials covered, let's dive into the **Lab 1**