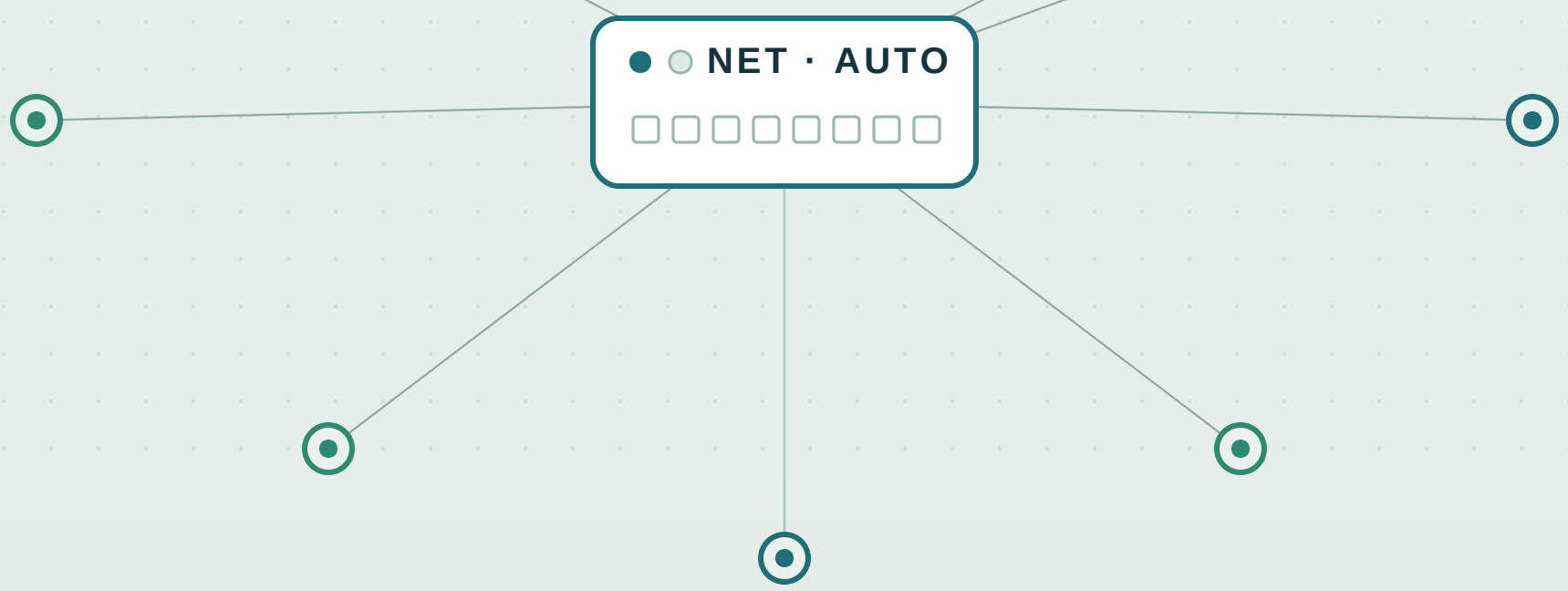
CCNP ENTERPRISE • AUTOMATION



ENAUTO
300-435

Complete Learning Guide

Automating and Programming Cisco Enterprise Solutions v2.0

DOMAINS 1.0 – 5.0 • 130+ PRACTICE QUESTIONS

Jozef Baroš

AUTHOR

CCNP ENTERPRISE · CCNP AUTOMATION

ENAUTO 300-435

Complete Learning Guide

Automating and Programming Cisco Enterprise Solutions v2.0

Device-Level · Controller-Based · Operations · AI in Automation

DOMAINS 1.0 – 5.0 • 130+ PRACTICE QUESTIONS

Jozef Baroš

AUTHOR

ENAUTO 300-435 — Complete Learning Guide

Automating and Programming Cisco Enterprise Solutions v2.0

First Edition · 2026

Trademarks. Cisco, IOS XE, Catalyst, Catalyst Center, Meraki, ThousandEyes, Identity Services Engine (ISE), and related marks are trademarks or registered trademarks of Cisco Systems, Inc. All other trademarks are the property of their respective owners. This book is an independent study guide and is **not** affiliated with, authorized, endorsed, or sponsored by Cisco Systems, Inc.

Disclaimer. This guide is provided for educational purposes. While every effort has been made to ensure accuracy, the author assumes no responsibility for errors or omissions, or for damages resulting from the use of the information herein. Always test automation in a non-production environment before deploying to a live network. Exam topics and product behaviour may change without notice; consult the official Cisco documentation and exam blueprint for the definitive reference.

About This Book

This guide is a complete, exam-focused preparation resource for **ENAUTO 300-435 — Automating and Programming Cisco Enterprise Solutions v2.0**, the 90-minute exam shared by the **CCNP Enterprise** and **CCNP Automation** certification tracks. It is the **second volume** in this network-automation series, following the *AUTOCOR 350-901 Complete Learning Guide* — but it stands entirely on its own and assumes no prior volume.

It is written for two audiences: engineers coming from the CCNP Enterprise core (ENCOR 350-401) with strong routing-and-switching skills but limited programmability experience, and automation practitioners who want a structured, Cisco-specific path through the blueprint. Every concept is built from first principles — terms such as *idempotency*, *declarative*, and *datastore* are defined where they first appear — so no reader is left behind.

What makes this guide different

Every sub-section follows a deliberate rhythm designed for retention:

- **Plain-English explanation** — the concept introduced from first principles.
- **Cisco-specific depth** — the details Cisco actually tests, not just the generic theory.
- **Runnable code** — real Python, Ansible, and configuration you can type into a lab.
- **Value boxes** — scattered throughout: 💡 *Did you know?*, ✅ *Exam Tip*, ⚠️ *Common Pitfall*, and 🌍 *From the Field*.
- **Summary** and **Key Takeaways** — a tight recap of every section.
- **Knowledge Check** — five focused questions per section with fully explained answers.

The detailed orientation for new readers — how the sections build on one another and how to get the most from the labs — follows in the **How to Use This Guide** page that opens Section 1.

> ✅ **Exam Tip — Read the verb in every objective**
>
> Cisco writes each blueprint line with a precise verb. **Describe** objectives are tested with conceptual, “which statement is true” questions. **Construct** objectives are tested with code, payloads, and “drag the lines into order” items. Knowing which verb governs an objective tells you whether to memorise facts or to practise building artifacts.

The Official Exam Blueprint

ENAUTO 300-435 is a **90-minute** exam. The five domains and their exam weightings are below — note that the largest domain, Controller-Based Automation, is worth nearly a third of your score, and that **AI in Automation** is the headline addition in the **v2.0** blueprint. This book devotes a full section to each domain, proportional to its weight.

#	Domain	Weight	In this book
1.0	Network Automation Foundation	10%	Section 1
2.0	Device-Level Network Automation	25%	Section 2
3.0	Controller-Based Network Automation	30%	Section 3
4.0	Operations	20%	Section 4
5.0	AI in Automation	15%	Section 5

The technologies woven through these domains include **Cisco IOS XE**, **Cisco Meraki**, **Cisco Catalyst Center**, **Cisco SD-WAN**, **Cisco Identity Services Engine (ISE)**, and **Cisco ThousandEyes** — driven through NETCONF, RESTCONF, Python, Ansible, and, new in v2.0, the **Model Context Protocol (MCP)** for AI agents.

> 💡 **Did you know? — The blueprint can change without notice**
>
> Cisco reserves the right to adjust exam topics at any time. This guide maps to the v2.0 blueprint published for the 300-435 exam, but you should always cross-check the current objectives on Cisco's official certification site before your exam date. Treat the blueprint as the authoritative table of contents for your study.

Table of Contents

Domain 1.0 — Network Automation Foundation

Domain 1.0 carries **10% of the ENAUTO exam** — the smallest slice of the five domains. New candidates routinely under-prepare it for exactly that reason and then lose easy marks, because the foundation concepts here resurface everywhere: the YANG models you learn in 1.1 are the same models you push with Netmiko, ncclient, and RESTCONF in Domain 2.0 and through Catalyst Center in Domain 3.0. A weak foundation does not cost you 10% — it quietly erodes your score across the whole paper.

The domain answers one question in three parts: **how is network configuration represented as data, and how does that data move?** The representation is defined by a schema language, **YANG**. The data itself is serialised as **XML** or **JSON**. And it is transported by two protocols, **NETCONF** and **RESTCONF**. The three sub-sections map exactly onto those parts:

- **1.1** — the three families of YANG models (OpenConfig, IETF, native) and how to read a module.
- **1.2** — NETCONF and RESTCONF: their transports, encodings, operations, and the datastore model that separates them.
- **1.3** — turning a YANG model into a correct JSON or XML payload, and reading the RFC 8340 tree notation, using YANG Suite and pyang.

Two mental models will carry you through every question. The first is **declarative versus imperative**. An *imperative* approach lists the steps ("enter config mode, type these commands"); a *declarative* approach states the desired end result ("VLAN 10 named SALES shall exist") and lets the system work out the steps. YANG-driven automation is fundamentally declarative. The second is **idempotency**: an operation is idempotent if running it twice leaves the device in the same state as running it once. Model-driven configuration is naturally idempotent because you declare a target state rather than a sequence of changes.

> 💡 **Did you know? — Programmability was bolted onto IOS XE, not the other way round**
>
> On a Catalyst 9000 or Catalyst 8000v, the CLI you type and the NETCONF/RESTCONF requests you send both funnel into the **same** internal data store via the **Binder** and the management infrastructure. That is why a change made over RESTCONF appears instantly in `show running-config`, and vice-versa — there is one source of truth on the box, exposed through several front doors.

1.1 YANG Data Models — OpenConfig, IETF, and Native

Why a modelling language exists at all

For three decades the network management interface was the **CLI**, designed for humans. Automation tools that screen-scrape the CLI are brittle: a software upgrade re-words a `show` command, a prompt changes, and the parser breaks. **SNMP** added a machine interface but its writable surface was limited, its MIBs were inconsistent, and almost nobody used it to configure devices. The industry needed a way to describe configuration and state as **structured, typed, validated data** with a published schema — and that is precisely what **YANG** provides.

YANG (Yet Another Next Generation) is a **data modelling language** standardised in **RFC 7950** (YANG 1.1; the original was RFC 6020). Three points are worth fixing in your mind immediately, because the exam tests all three:

1. YANG is **not** a protocol. It does not move data. NETCONF and RESTCONF do that.
2. YANG is **not** a data format. The data is carried as XML or JSON. YANG is the *schema* those documents must conform to.
3. YANG is **vendor-neutral by design**, even though vendors publish their own (native) models in it.

A useful analogy: YANG is the architectural blueprint, an XML or JSON document is a house built to that blueprint, and NETCONF/RESTCONF are the trucks that deliver the building materials. The blueprint says "a VLAN has an integer `id` between 1 and 4094 and a string `name`"; the document says "`id` 10, `name` SALES"; the protocol carries that document to the device.

The three model families

Cisco devices typically support **all three** families simultaneously. Understanding what each is for — and the trade-off between feature coverage and portability — is the single most-tested idea in 1.1.

Native models

Native models are written by the device vendor and expose the device's **entire feature set**, mapping almost one-to-one onto the CLI. On IOS XE the root module is `Cisco-IOS-XE-native`, and beneath it hang feature modules such as `Cisco-IOS-XE-bgp`, `Cisco-IOS-XE-ospf`, and `Cisco-IOS-XE-interfaces`. If a feature exists in the CLI, it almost certainly has a native YANG node; this is the only family guaranteed to reach the newest, most platform-specific knobs.

The price is **zero portability**. A payload written against `Cisco-IOS-XE-native` is meaningless to a Nexus switch or a Juniper router. You also tie your automation to Cisco's module structure, which can change between releases.

```
# Native: configure a loopback's IP via RESTCONF, IOS XE
# Notice the deeply Cisco-specific path — this works ONLY on IOS XE
PUT /restconf/data/Cisco-IOS-XE-native:native/interface/Loopback=0
Content-Type: application/yang-data+json

{
  "Cisco-IOS-XE-native:Loopback": {
```

```
    "name": 0,
    "ip": {
      "address": {
        "primary": { "address": "10.0.0.1", "mask": "255.255.255.255" }
      }
    }
  }
}
```

IETF (standard) models

IETF models are published as RFCs by the Internet Engineering Task Force and are intended to be **vendor-neutral**. The two you must recognise by name are `ietf-interfaces` (**RFC 8343**) and `ietf-routing` (**RFC 8349**). They model common functionality that every vendor implements, so the same payload works across compliant devices. Their weakness is **lag and lowest-common-denominator coverage**: standardisation is slow, so the newest or most proprietary features often have no IETF node.

```
# IETF standard: the SAME payload works on any RFC 8343 device
PUT /restconf/data/ietf-interfaces:interfaces/interface=Loopback0
Content-Type: application/yang-data+json

{
  "ietf-interfaces:interface": {
    "name": "Loopback0",
    "type": "iana-if-type:softwareLoopback",
    "enabled": true
  }
}
```

OpenConfig models

OpenConfig models are produced by a consortium of large network **operators** (Google, AT&T, Microsoft, and others) rather than by a standards body or a single vendor. They were created out of operational frustration: operators running multi-vendor fleets wanted **one** model per feature that worked everywhere. Module names carry the `openconfig-` prefix, e.g. `openconfig-interfaces` and `openconfig-network-instance`. OpenConfig moves faster than IETF and is the usual choice for vendor-neutral, streaming-telemetry-friendly automation at scale.

A defining structural habit of OpenConfig is the **`config` / `state` split**: writable intended values live under a `config` container and the corresponding read-only operational values live under a parallel `state` container with the same leaf names. This makes it trivial to compare *what you asked for* against *what the device is actually doing*.

```
# OpenConfig: note the parallel config/state containers
{
  "openconfig-interfaces:interface": [
    {
      "name": "GigabitEthernet2",
      "config": {                // <-- writable intended state
```

```
        "name": "GigabitEthernet2",
        "description": "Uplink to core",
        "enabled": true
      },
      "state": {                  // <-- read-only operational state
        "name": "GigabitEthernet2",
        "oper-status": "UP",
        "admin-status": "UP"
      }
    }
  ]
}
```

Property	Native	IETF	OpenConfig
Author	Device vendor	IETF (RFCs)	Operator consortium
Portability	None (vendor-locked)	High	High
Feature coverage	Complete	Common subset	Broad, operator-driven
Prefix example	Cisco-IOS-XE-*	ietf-*	openconfig-*
Config/state split	No	No	Yes
Use when…	You need every IOS XE knob	Standards compliance matters	One model for many vendors

> ✅ **Exam Tip — "Multi-vendor" and "every feature" are the keywords**
>
> If a question stresses **multi-vendor**, **standards-based**, or **portable**, choose **IETF** or **OpenConfig**. If it stresses **every IOS XE feature**, **maps to the CLI**, or **the newest proprietary knob**, choose **native**. If it specifically mentions a parallel **config/state** structure, that is a fingerprint of **OpenConfig**.

Anatomy of a YANG module

You will not write YANG on the exam, but you must read it fluently — questions show you a module and ask what a payload built from it must look like. A module is assembled from a small, fixed vocabulary of node types:

- **container** — a grouping node that holds other nodes but has no value of its own (think: a folder).
- **list** — a set of entries, each uniquely identified by one or more **key** leaves (think: a table with a primary key). Lists are the only nodes that take a key.
- **leaf** — a single node holding exactly one value of a given **type**.
- **leaf-list** — a node holding multiple values of the same simple type, with no key.

Around those, YANG provides **statements** that constrain and document each node: `type` (the data type), `mandatory` (must be present), `default` (assumed value if omitted), `description` (human documentation), `must` and `when` (conditional constraints), and `range` / `length` / `pattern` (type restrictions). Cisco loves to test these constraints — especially that supplying a value outside a `range` is rejected by the model before it ever reaches the device.

A trimmed YANG 1.1 module for VLANs — read it top-down and note every constraint

```
module acme-vlans {
  yang-version 1.1;
  namespace "http://acme.example/vlans";
  prefix acmevlan;

  import ietf-yang-types { prefix yang; }   // reuse standard types

  container vlans {
    description "Top-level container for all VLANs";

    list vlan {
      key "id";                        // 'id' is the primary key
      description "A single VLAN definition";

      leaf id {
        type uint16 { range "1..4094"; }   // model rejects 0 or 4095+
        description "VLAN identifier";
      }
      leaf name {
        type string { length "1..32"; }    // 1–32 characters
        mandatory true;                     // cannot be omitted
      }
      leaf state {
        type enumeration {
          enum active;
          enum suspend;
        }
        default "active";                   // assumed if not sent
      }
      leaf-list tagged-ports {
        type string;                        // many values, no key
      }
    }
  }
}
```

Reading top-down: the `vlans` **container** holds a `vlan` **list** keyed by `id`. The `id` leaf is a `uint16` restricted to **1–4094** — send `4095` and the *model* rejects it. `name` is a mandatory string of 1–32 characters. `state` is an **enumeration** limited to `active` or `suspend`, defaulting to `active`. `tagged-ports` is a **leaf-list**, so it can hold many port names with no key.

> ⚠️ **Common Pitfall — Confusing a list with a leaf-list**
>
> A **list** holds keyed, multi-field entries (e.g. each VLAN has an `id`, `name`, and `state`). A **leaf-list** holds many copies of a single simple value (e.g. a flat list of port names). On the exam, if a node has a `key` statement and child leaves, it is a **list**; if it is a single typed node that simply allows multiple values, it is a **leaf-list**. Treating one as the other produces an invalid payload.

Namespaces, prefixes, and how modules combine

Every module declares a **namespace** (a globally unique URI) and a short **prefix** used to refer to it. Namespaces are what let two different vendors each define an `interface` node without collision — the full identity of a node is *namespace + name*, not just the name. In an XML payload the namespace appears as an `xmlns` attribute; in **RFC 7951** JSON it appears as a `module:node` prefix on the topmost node contributed by each module (for example `"ietf-interfaces:interfaces"`).

Modules reuse each other through **import** (pull in types or groupings from another module, as `acme-vlans` imports `ietf-yang-types` above) and extend each other through **augment** (a module bolts additional nodes onto a tree defined elsewhere). Cisco frequently *augments* standard IETF/OpenConfig trees with proprietary IOS XE extensions — a detail worth recognising because it explains why a "standard" path on an IOS XE box sometimes exposes extra Cisco-only leaves.

> 💡 **Did you know? — identity and identityref make interface types extensible**
>
> The interface `type` leaf is not a free string — it is an **identityref** drawn from the `iana-if-type` module, which defines identities such as `ethernetCsmacd`, `softwareLoopback`, and `l3ipvlan`. Using identities (rather than an enumeration) means new interface types can be added in a separate module **without modifying** the interfaces model — a small design choice that turns up in payload-construction questions because the value must always carry its module prefix.

> 🌍 **From the Field — Why operators standardised on OpenConfig for telemetry**
>
> In a production network spanning Cisco, Juniper, and Arista, writing three separate parsers for the same interface counter is exactly the brittleness operators wanted to escape. By subscribing to `openconfig-interfaces` sensor paths everywhere, a single telemetry pipeline ingests identical data structures from every vendor. This is why OpenConfig and streaming telemetry grew up together — and why ENAUTO pairs model knowledge with the telemetry objectives in Domain 4.0.

Summary

YANG (RFC 7950) is a vendor-neutral **schema** language — not a protocol and not a data format. It defines the structure, types, and constraints of configuration and operational data, which is then serialised as **XML** or **JSON** and moved by **NETCONF/RESTCONF**. Three model families exist: **native** (vendor-authored, complete feature coverage, no portability, e.g. `Cisco-IOS-XE-native`), **IETF** (RFC-standardised and portable, e.g. `ietf-interfaces`/RFC 8343), and **OpenConfig** (operator-driven, portable, with a distinctive **config/state** split). Modules are built from **container**, **list** (keyed), **leaf**, and **leaf-list** nodes, constrained by statements such as `type`, `range`, `mandatory`, and `default`. **Namespaces + prefixes** keep node names unique; **import** and **augment** let modules reuse and extend one another; interface types use **identityref** values that always carry a module prefix.

Key Takeaways

YANG describes the **schema**; XML/JSON carry the **data**; NETCONF/RESTCONF move it. **Native** = every feature, no portability. **IETF/OpenConfig** = portable, possibly fewer features. The **config/state split** is OpenConfig's fingerprint. A **list** has a **key** and structured entries; a **leaf-list** holds many simple values. Model **constraints** (`range`, `mandatory`, `length`, `enumeration`) are validated before the device sees the data. **identityref** values (e.g. `iana-if-type:ethernetCsmacd`) are always module-prefixed.

Knowledge Check — 1.1 YANG Data Models

Q1. An operator must apply identical interface configuration across Cisco, Juniper, and Arista devices using one model. Which family fits best, and why?

- **A.** Native, because it maps directly to each vendor's CLI
- **B.** OpenConfig, because it is operator-driven and vendor-neutral
- **C.** A separate native model per vendor, validated by SNMP
- **D.** IETF only, because OpenConfig cannot model interfaces

Correct answer: B. OpenConfig was created by operators precisely to give one vendor-neutral model per feature across a multi-vendor fleet. Native models are vendor-locked, and IETF does model interfaces but the question's emphasis on operator-driven multi-vendor scale points to OpenConfig.

Q2. A YANG model defines `leaf id { type uint16 { range "1..4094"; } }`. A client sends id = 4095. What happens, and where?

- **A.** The device accepts it and truncates the value to 4094
- **B.** It is rejected by model validation before reaching the device's config logic
- **C.** It is silently ignored and the leaf is left unset
- **D.** It succeeds because range is only documentation

Correct answer: B. A range restriction is an enforceable type constraint, not documentation. The value 4095 fails validation against the schema, so the request is rejected before the configuration is applied. Truncation and silent-ignore are not how YANG type constraints behave.

Q3. You see a payload containing parallel `config` and `state` containers with the same leaf names under each interface. This structure is the signature of which model family?

- **A.** Cisco native
- **B.** IETF ietf-interfaces
- **C.** OpenConfig
- **D.** SNMP MIB conversion

Correct answer: C. The parallel config (writable intended) and state (read-only operational) containers are OpenConfig's defining structural habit. Native and IETF models do not split intended and operational state this way, and SNMP is unrelated.

Q4. Which statement about the interface `type` leaf (an identityref from iana-if-type) is correct?

- **A.** Its value must be a module-prefixed identity such as iana-if-type:ethernetCsmacd
- **B.** It accepts any free-form string the user chooses
- **C.** It is an integer index into a vendor table
- **D.** It is read-only operational state and cannot be set

Correct answer: A. type is an identityref, so its value is an identity that carries its defining module's prefix (e.g. iana-if-type:ethernetCsmacd). It is not free text, not an integer index, and it is configurable, not read-only.

Q5. What is the practical difference between a YANG list and a leaf-list?

- **A.** A list is read-only; a leaf-list is writable
- **B.** A list holds keyed, multi-field entries; a leaf-list holds many values of one simple type with no key
- **C.** A leaf-list can have a key but a list cannot

D. They are identical; the names are interchangeable

Correct answer: B. A list models keyed, structured records (each entry has a key and child leaves); a leaf-list simply allows multiple values of a single simple type and has no key. The other options invert or confuse these properties.

END OF SAMPLE

You've just read one complete section of

ENAUTO 300-435 — Complete Learning Guide

The full book contains all five exam domains:

- ❖ Domain 1.0 — Network Automation Foundation (YANG, NETCONF, RESTCONF)
- ❖ Domain 2.0 — Device-Level Automation (Netmiko, ncclient, Ansible, ZTP, EEM)
- ❖ Domain 3.0 — Controller-Based Automation (Catalyst Center, Meraki, ISE, Jinja2)
- ❖ Domain 4.0 — Operations (pyATS, CML, SWIM, Telemetry, Webhooks)
- ❖ Domain 5.0 — AI in Automation (AIOps, AI-assisted code, MCP / FastMCP)

123 pages · 130+ practice questions with explained answers

Runnable Python, Ansible & IOS XE code throughout

Exam Tips · Common Pitfalls · Did You Know · Real-World scenarios

Get the complete guide on LeanPub

Part of the network-automation series alongside AUTOCOR 350-901

Jozef Baroš

AUTHOR