Co-funded by the European Union

Project Nr. 2022-1-FR01-KA220-HED-000086863

lightcode

Strengthening the Digital
Transformation of Higher Education
Through Low-Code

13. Securing Your App

Paris-Dauphine University – PSL

Erasmus+ Project
lightcode

Co-funded by
the European Union

PROJECT'S COORDINATOR

Paris Dauphine University
France

PROJECT'S PARTNERS

University of Macedonia Greece

University of Niš
Serbia

Karmic Software Research
France

REACH Innovation
Austria

University of Zagreb
Croatia

Symplexis
Greece

symplexis.eu

Table of Contents

Co-funded by the European Union

OVERVIEW

Security is often presented as a deeply technical concern — encryption protocols, authentication layers, server hardening, and audit trails. But in most apps, some of the **most impactful security decisions** are much simpler:

Who sees what? Who can do what? And when?

In a low-code platform like LightCode, the heavy infrastructure — HTTPS, hosting, backend logic — is mostly hidden from the builder. That's by design. And that's why it's easy to ask:

So… is this chapter already over?

Not quite. While many protections are handled behind the scenes, **you still play a critical role** in shaping how users interact with your app — and whether their experience feels safe, respectful, and reliable.

In this chapter, we'll explore how to **model responsible behavior** inside your apps — even if enforcement mechanisms are not (yet) fully implemented. You'll learn to:

- Simulate access control using visibility rules and attributes,
- Prevent mistakes using validation and object states,
- Express sensitive intent (like password fields),
- And design workflows that guide users — and block potential misuse.

Security isn't just about barriers. It's about **making good intentions visible through design**.

Co-funded by the European Union

Introduction

By the book, software security covers a long list: data encryption, user authentication, role permissions, token expiry, access logs, GDPR compliance…

Except in most no-code and low-code environments, including LightCode, **all of this is either automated or invisible**. HTTPS is already on. No server to configure. No backend tokens to manage.

So this chapter is over, right?

✅ Mission accomplished?

❌ Actually… not yet.

Because even if LightCode hides the infrastructure, **you're still designing the experience** — and that experience involves data, interactions, visibility, and behavior.

This chapter helps you build apps that:

- Reflect **who should see or do what**,
- Handle inputs and transitions responsibly,
- Use basic constructs (like field names, state transitions, visibility toggles) to **signal intent** — and prevent problems before they happen.

We'll stay fuzzy on enforcement — some features exist today (like password fields that auto-hide characters), others are modeled conceptually. All are here to prepare you to **think clearly**, **design carefully**, and grow into more advanced systems over time.

This is not about being perfect. It's about building **trustworthy patterns**, step by step.

Co-funded by the European Union

Why is this Module Important?

Security isn't just for backend developers. Most bugs and breaches in real-world apps come from **design oversights**:

- A button that shouldn't be visible,
- A field that accepts the wrong input,
- A state transition that skips essential logic.

In LightCode, you may not control the server — but you do control what your app *feels* like. Is it respectful? Predictable? Trustworthy?

That's where your power lies:

- Expressing sensitive intent (like marking a field as a password),
- Making sure actions happen only under the right conditions,
- And guiding users through safe, simple workflows.

These patterns are valid even outside LightCode — they apply to any tool, any platform, and any team building something real.

Security begins with awareness. It continues with intention. And it's always shaped by design.

Learning Outcomes

By the end of this chapter, learners will be able to:

- Recognize the difference between **infrastructure-level protections** and **design-level responsibilities**

Co-funded by the European Union

- Use naming and structure to express **intent** (e.g. marking a password field)
- Model simple access control and validation using visibility rules, states, and methods
- Anticipate potential misuse through **User and AbUser Stories**
- Prevent harmful transitions by designing logic around object states
- Reflect critically on what "safe and respectful" design looks like — even in beginner-friendly tools
- Build apps that feel secure, trustworthy, and aligned with real-world expectations

Prerequisites

This module doesn't assume any knowledge of cybersecurity, login systems, or encryption. No token setup. No server secrets.

But it does assume that you:

- Have completed **Chapter 8** (exploring LightCode),
- Know how to define and manipulate **objects, states, and methods** (Chapters 10–12),
- And feel comfortable navigating an app made of **blocks, transitions, and visible logic**.

While LightCode doesn't require you to set up credentials or role systems by default, this module invites you to **reason about visibility, behavior, and access** as part of your design. Whether you're preparing for future features, extending the platform with your instructor, or simply exploring responsible patterns, you'll learn to:

- Design user experiences with **intended audiences** in mind,
- Use structure and naming to express **sensitive or restricted content**,
- Think in terms of **who should see, do, or change what** — and under which conditions.

Security, here, is not about configuring protocols. It's about shaping experiences that feel thoughtful, respectful, and intentional — and being ready to grow as new capabilities emerge.

STRUCTURE OF THE MODULE

This module is organized as a journey through key ideas of secure design — not from a backend perspective, but from the builder's point of view.

You'll start by understanding where traditional security lives, and where design-level responsibility begins. You'll meet Users and AbUsers, and explore how their journeys shape what should be seen, edited, or allowed.

From there, you'll design visibility and control directly into your interface using read-only fields, conditional buttons, and state-aware actions. Throughout, you'll practice writing small tests — not just to confirm the happy path, but to catch potential misuse.

The module ends with a reflection: what happens when a clever user bypasses your logic? What can you do next?

No login systems. No encryption keys. Just thoughtful structure, good habits, and design that speaks clearly.

SECURING YOUR APP, STEP BY STEP

1. By the Book: What Does Security Usually Involve?

Before we explore what *you* can do as a builder, let's step back and take a broader view of what "security" really means in digital systems.

According to *Ross Anderson*, security isn't just about code or cryptography — it's about building systems that remain dependable even when things go wrong: by mistake, through misuse, or with malicious intent.

"Security is about building systems to remain dependable in the face of malice, error, or mischance."

— *Ross Anderson, Security Engineering*

In practice, this means asking:

- Who is using the system?
- What are they allowed to do?
- How do we avoid accidents?
- How do we handle sensitive data?
- What expectations should users have — and can they trust them?

These concerns exist no matter what kind of tool you use — and yes, even if you don't write any code.

Let's walk through the most common categories of security concerns and see where they usually live in a system.

🔐 1.1 Authentication

Who are you, really?

Authentication confirms a user's identity — whether with a password, email link, biometric scan, or magic pineapple (well, not that one… yet).

🔧 **Possible implementations**:

- Managed by platform or IT team (e.g., single sign-on)
- Integrated with third-party identity services (OAuth, LDAP)
- Simulated using user-input attributes or device context

🧑‍🤝‍🧑 1.2. Authorization

Now that we know who you are… what can you do?

Authorization defines roles, permissions, and access rules — for instance, editors can modify, viewers can only read.

🔧 **Possible implementations**:

- Role-based access control (RBAC)
- Attribute-based access (e.g., owner == user)
- Contextual visibility (e.g., buttons shown only under certain conditions)

📡 1.3. Infrastructure Security

Is the underlying system secure?

Covers HTTPS, firewalls, encrypted databases, physical hosting, server configuration.

Co-funded by the European Union

🔧 **Usually managed by**:

- Hosting provider
- DevOps or platform administrators
- IT infrastructure teams

Designers typically don't touch this — but benefit from it *a lot*.

📥 1.4. Input Validation

Are we sure this input is acceptable?

Just because a user submits data doesn't mean it's clean, expected, or logical.

🔧 **Possible implementations**:

- Type constraints (e.g., must be a number)
- Allowed ranges, lists, patterns
- Required fields and custom validation logic

🔁 1.5. Process Integrity

Can users jump ahead or skip essential steps?

Good apps guide users through the correct path — no refunds before payment, no deletion without confirmation.

🔧 **Possible implementations**:

- State machines and workflows
- Guard clauses on transitions
- Disabled buttons, hidden actions, delayed triggers

🕵️ 1.6. Data Privacy

Is sensitive data protected and shown only when needed?

Not all data is meant to be public. Passwords, personal notes, health details — some things require extra care.

🔧 **Possible implementations**:

- Field-level masking (e.g., obfuscated inputs)
- Conditional display logic
- Encrypted storage or pseudonymization

⚖️ 1.7. Ethical & Legal Considerations

Are we treating users fairly?

GDPR and similar regulations require that users can give consent, access their data, and ask for its deletion.

🔧 Possible implementations:

- Consent checkboxes, opt-ins, "Download my data" buttons, visual indicators, and user-friendly policies.
- Or even more radical: letting builders store their resources in **their own repositories**, without keeping them on the platform's server side.

💡 That's exactly what LightCode does — giving ownership and control back to the builder.

Co-funded by the European Union

🧰 Where Does It All Live?

Each of these concerns can be addressed at different levels:

Concern	Commonly Managed By	Can Also Be Modeled or Designed By
Infrastructure	Hosting provider, platform	(Rarely by app builders)
Authentication	Platform, IT team	Simulated with user context or attributes
Authorization	Developers, admins	UI logic, visibility, workflow
Input Validation	Developers, form designers	Visual builders, block configuration
Process Integrity	App logic	Visual flows, method conditions
Data Privacy	Developers	Design intent, field labeling, masking
Compliance & Ethics	Policy team, legal	Consent design, user feedback, clear copy

🧠 So What's Your Role?

Even if you're not setting up servers or encrypting databases, you're **still shaping how secure and trustworthy the app feels** — especially in low-code tools where the design and logic live close together.

Co-funded by the European Union

You are:

- A guide who decides what should be visible,
- A gatekeeper who defines what should happen when,
- A steward of user data and experience.

Security by design means thinking ahead:

- Could someone click the wrong thing?
- Should this be hidden?
- Does the user know what's happening — and why?

🎉 Quick Quiz – "Security or Not?"

Let's warm up with a fast quiz. For each item below, decide:

Is this a security concern, or just a nice-to-have?

1. Hiding a "Delete All" button until a user confirms their role.
2. Preventing users from entering negative numbers in a "quantity" field.
3. Asking for consent before storing user preferences.
4. Letting every visitor download a team's private project data.
5. Skipping the "Confirm Payment" step to save time.
6. Automatically hiding password characters in a field.
7. Showing a chart before the underlying data is ready.

Co-funded by the European Union

Spoiler: If you said "security" to all but maybe #7... you're on the right track.

From here, we'll begin to explore how **you, as a builder**, can design safer, clearer, and more respectful applications — even in a visual environment.

2. Enter the Characters: Users and AbUsers

Now that we've mapped the major domains of security, it's time to change perspective.

Security isn't just about protecting a system — it's about **designing with people in mind**.

In every app, there are intended users — people who follow the expected path and interact in good faith. But there are also edge cases, misclicks, curious explorers, and — let's face it — people who push boundaries.

That's why in secure design thinking, we often work with two types of **personas**:

🧑‍💻 The User

The person who's supposed to be there.

They want to get something done. They may be distracted, tired, or new to the system. Your job is to **help them succeed, safely**.

User Story:

"As a project participant, I want to view only my own feedback, so I feel respected and focused."

😈 The AbUser (Ab-User = Anti-pattern User)

This isn't a supervillain — just someone whose actions **challenge your assumptions**.

They might:

- Try to access something they shouldn't,
- Skip steps,
- Tamper with inputs,
- Or just misunderstand how something works.

AbUser Story:

"As a curious participant, I want to change the URL and view someone else's private feedback."

🧠 Why Personas Matter

By imagining both Users and AbUsers, you train yourself to:

- **Anticipate misuse** without becoming paranoid,
- Build in **gentle safeguards** and meaningful defaults,
- And make your designs **safe by default, forgiving by design**.

From this point on, we'll explore each core area of security by design through **story pairs**:

👩‍💻 one expected user scenario, and 😈 one potential misuse — followed by how you can design to support one and prevent the other.

3. From Users to Journeys: Designing with Intent

Now that we've met our users (and their cheeky counterparts, the AbUsers), it's time to get serious about their experience — not just who they are, but **what they're trying to do**.

In design, we often talk about the **User Journey**: a sequence of actions and expectations that guide a user from their first interaction all the way to their goal.

In secure system design, this journey matters even more — because **intentions shape responsibilities**.

🧃 Example: The Beverage App Journey

Let's say we're designing a small app to let Emma, our thirsty user, order a matcha tea after judo practice.

Her **user journey** might look like this:

1. **See available drinks**
2. **Choose one**
3. **Confirm selection**
4. **Submit payment**
5. **Wait for delivery**
6. **Receive and enjoy the beverage** ☕

Simple, right? But every step of this journey raises subtle design questions:

🔍 What Does the User See and Do?

- Should all drinks be visible to every user?

- Is the price editable?
- Can someone click “Serve Order” before it’s paid for?
- Should Emma see other people’s choices?
- Can a curious AbUser bypass the payment step and still get a drink?

Designing **with intent** means answering those questions clearly, and making those answers visible in the app’s behavior — even if the app doesn’t enforce them automatically.

🔐 Visibility, Editability, and Sensitivity

Each field or block in your app carries an **implicit access level**, depending on the journey step and the user’s role:

Property	Question to Ask	Example
Public or Private	Who should see this?	Order total, delivery status
Sensitive or Ordinary	Does this need to be masked or protected?	Payment method, phone number
Editable or Read-only	Who can change this, and when?	Drink selection (editable before order), delivery time (set by system)

These aren’t security settings in a menu — they’re design choices. And in low-code tools, they can often be expressed directly: via input types, field names, visibility toggles, or conditional logic.

Co-funded by the European Union

🧠 It's Not Just Visibility — It's Integrity

Even something as basic as a **data type** matters for security:

- Letting a user type "free coffee forever" into a field expecting a number can crash the app or break logic.
- Accepting an empty field where a name is required can cause actions to fail silently.
- Mixing up string and boolean inputs can make rules unenforceable.

That's why using **precise data types** (number, string, boolean, list…) and constraints is a form of protection — not just for the app, but for the user.

Security starts with asking:

What's the goal here?

What should be visible? Modifiable? Trusted?

✅ What Comes Next?

As we walk through the rest of this chapter, we'll revisit our design using this approach:

- What is the user trying to do?
- What information or actions are appropriate *at this stage*?
- What can go wrong — and how can we gently prevent it?

The rest of our journey will explore:

- **Visibility and display logic**
- **Input types and validation**

- **Object states and available actions**
- **Guarding transitions**
- **Respecting consent and context**

And every time, we'll bring back our two favorite companions:

🧑‍💻 *User Story* and 😈 *AbUser Story*

Ready? Let's start with visibility — the first surface of security by design.

4. Designing for Visibility, Trust & Control

In Step 3, you learned to ask: *who is doing what, and why?* Now we start building that into the interface.

But unlike in traditional programming, you're not wiring access policies deep in the backend. Instead, you're shaping **what the user sees and can do — at every step**.

This is where **design becomes defense**. Let's look at how your app's interface can guide, protect, and inform — all without shouting "SECURITY!"

✨ 4.1 Visibility: Who Should See What?

Just because a field exists in the data model doesn't mean it needs to be shown.

When you add a form (e.g., a Form block), you can **explicitly pick which fields to include**. This is your first (and often most powerful) security tool.

Ask yourself:

Co-funded by the European Union

- Is this field necessary here?
- Could this confuse, tempt, or mislead a user?
- Could revealing this create risks (privacy, data leaks, etc.)?

Examples:

- Don’t show admin_notes to regular users.
- Show order_id, but make it read-only.
- Only show payment_status *after* payment has started.

🧩 4.2 Control: Read-Only vs Editable

Some things are **safe to display**, but **dangerous to change**.

Every attribute in LightCode can be marked read_only. Combine this with visual cues and smart layout to:

- Make the app feel responsive but locked-down when it matters,
- Avoid accidental edits,
- Signal when the system is in control.

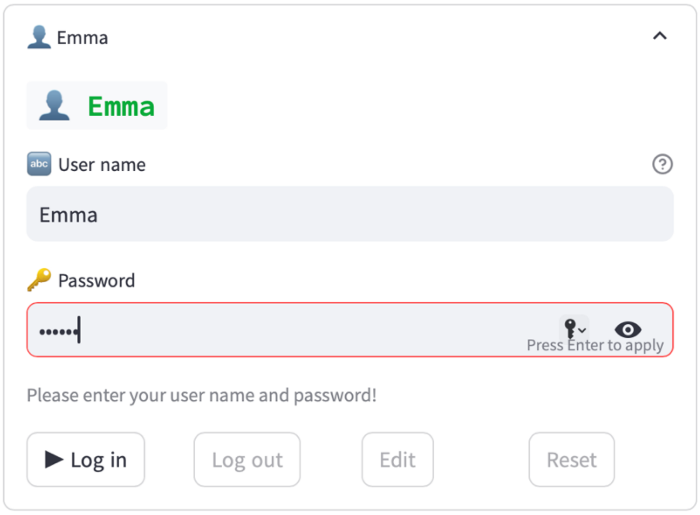
***Tip:** Fields like state, created_at, and owner are typically **read-only**.*

Co-funded by the European Union

👀 4.3 Masking Sensitive Inputs

Even without strong encryption, it helps to **avoid broadcasting sensitive data**:

- Fields starting with password are masked automatically in forms,
- Long notes, internal codes, or tokens can be hidden or minimized using text areas, or expandable sections.


Emma

Emma

User name

Emma

Password

••••••

Press Enter to apply

Please enter your user name and password!

Ask:

- Do I really need to show this?
- If yes, is the current form the right place?

🎭 4.4 Simulating Roles with Multiple Forms

You can simulate different user roles by creating **custom forms with selected fields**:

- A "Waiter View" with only order summary and delivery actions,
- A "Manager View" with full order and customer history.

This way, **form configuration becomes role modeling** — a concrete way to teach **access control without actual login systems**.

Co-funded by the European Union

🧪 Micro-Challenge

Here's a mini design prompt:

You're building an interface for a delivery assistant.

They should **see** the order state and drink name,

But **not** be able to modify anything except clicking "Serve."

How would you configure your Form block?

💡 *Hint: Which attributes go in the list? Which are read_only? Should Serve be an allowed method?*

5. Regulating Access: Who Can Do What, and When?

Once you've shaped **what users can see** (Step 4), the next question is:

Can they do anything with it?

In LightCode, this boils down to *intentional control* over fields and actions:

- What is editable?
- What is locked?
- What is allowed at this stage?
- What should never be touched?

Let's look at some real-life examples.

🔒 5.1 Fields You Can See… But Not Change

Some values are visible for context — but modifying them would cause chaos.

Examples:

- An internal stock count for a product might be **read-only**,
- A system-generated name used for wiring behind the scenes may be **advanced** and non-editable by default.

In LightCode, you can mark any field as:

- read_only: can be shown, but not changed,
- advanced: hidden in basic mode, only shown to expert designers.

This ensures that **accidents are prevented**, without hiding useful information.

🛑 5.2 Widgets That Know When to Stay Quiet

Sometimes, a component should exist but be **temporarily disabled**:

- A form button that shouldn't trigger unless payment is complete,
- A switch that depends on admin validation,
- A dropdown disabled while loading options.

Any visual block can be **explicitly marked as disabled** — and re-enabled later.

This allows for rich, **stateful interactivity** while preserving control.

🧭 5.3 States Shape What Happens Next

Co-funded by the European Union

Behind the scenes, many user interactions follow **state machines** — invisible but powerful.

For instance:

- An order might only allow "Serve" once it's "Confirmed",
- A function might only run if declared beforehand.

You can define allowed transitions and even **guard methods** using conditions tied to those states. This protects your logic flow without writing code.

🧪 Micro-Challenge

A user is reviewing a beverage.

- The stock count must be **visible but read-only**,
- The delivery button should only be enabled **after payment**,
- The system-generated name should **not be editable**, unless in advanced mode.

Which properties would you use?

☑ read_only = True

☑ disabled = True/False (based on state)

☑ advanced = True

Nice work. You're no longer just building — you're governing.

🖼️ Example: Read-Only Fields, Disabled Actions, and Visual Security Cues

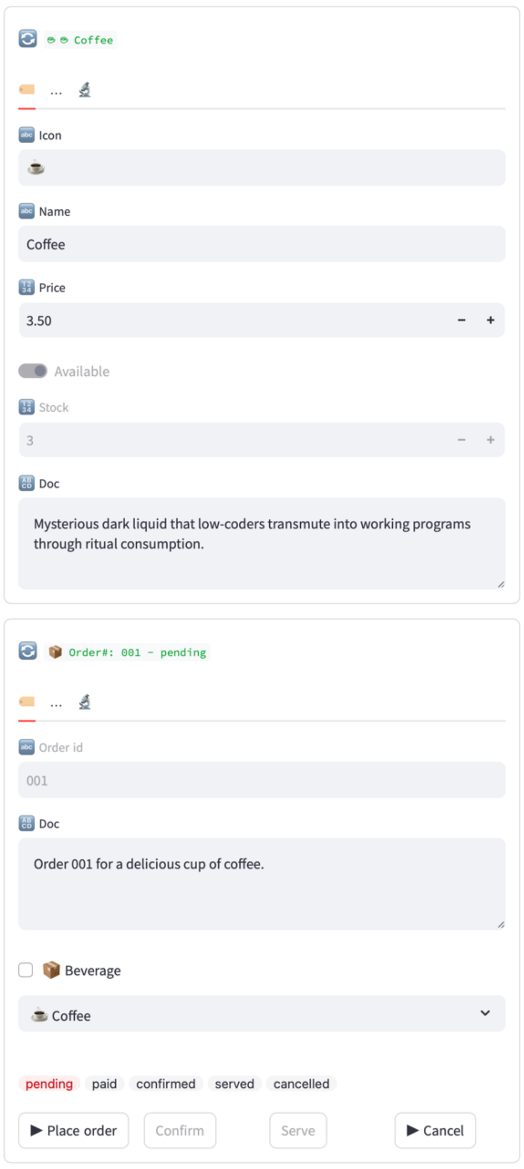
This screenshot illustrates several key principles of **security by design** in action:

- **Beverage stock** is visible but **read-only**: its value can inform the user, but only the Place order method is allowed to modify it. This protects inventory integrity without hiding useful information.

- **Order ID** appears in grey — it is **disabled by design**, showing that the value is assigned and controlled by the system, not manually set by the user.

- **Buttons like "Confirm" and "Serve" are visible but inactive** (greyed out), clearly signaling they are **unavailable in the current state** (pending). This avoids user confusion and prevents misuse, while still helping the user anticipate the next steps.

- The **state tag** (pending) is always shown, but **read-only**, providing transparency without risk.



Coffee

Icon: ☕

Name: Coffee

Price: 3.50

Available

Stock: 3

Doc: Mysterious dark liquid that low-coders transmute into working programs through ritual consumption.

Order#: 001 - pending

Order id: 001

Doc: Order 001 for a delicious cup of coffee.

☐ Beverage: ☕ Coffee

pending | paid | confirmed | served | cancelled

These small choices — visibility, editability, and state-aware controls — combine to create a user experience that feels safe, clear, and respectful.

Security isn't always about blocking access. Sometimes, it's about **designing interfaces that guide** — with the right amount of control, at the right time.

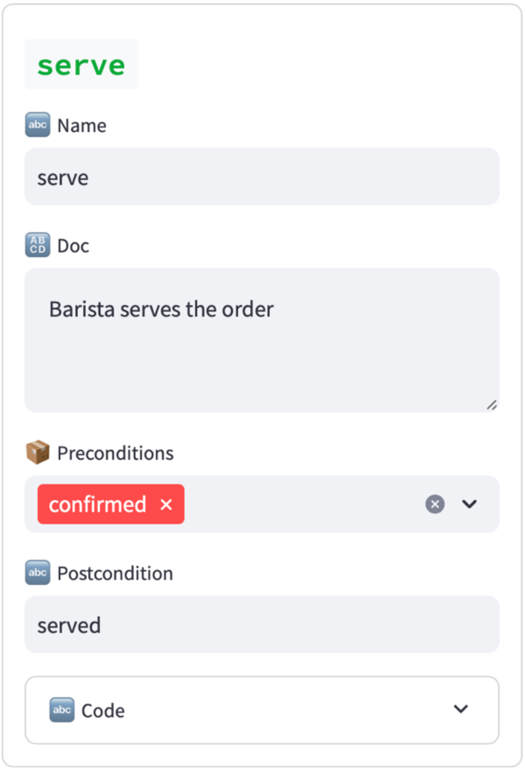
😈 AbUser Story – "Skip the Payment, Grab the Drink"

"As an opportunistic user, I want to click *Serve* directly, without paying, so I get my beverage for free."

🖼️ Example: Multi-Layered Protection in Action

In this screenshot, we see the method serve, which finalizes the delivery of a beverage. Notice how **design intent is embedded at every level**:

- The **button for "Serve"** is automatically **disabled in the UI** unless the order has reached the confirmed state.
- This behavior is **not hardcoded into the interface** — it comes directly from the method's **precondition**, which clearly states: confirmed.
- The **postcondition** marks the expected result: the order will become served.



serve

Name: serve

Doc: Barista serves the order

Preconditions: confirmed ×

Postcondition: served

This is what **security by design** looks like in low-code:

✅ Intent is declared once,

✅ Enforced consistently across logic and interface,

✅ And **users can't skip ahead** — because the system simply doesn't let them.

Even mischievous users (😈 AbUsers) attempting to trigger the action manually will be blocked — because the rules are enforced **everywhere**.

7 – When the AbUser Outsmarts Your Design

Let's end this chapter with a little self-confidence. You've handled visibility. You've locked down inputs. You've wrapped actions in preconditions. Time to prove it all works with a test, right?

Let's write an AbUser test — one that tries to **serve a beverage without confirmation**.

🔤 Code

```
order = Order(beverage=Beverage(name="Test Drink", price=2.5, stock=5))
order.place_order()
order.serve()
assert order.state != "served", "🚨 Order should not be served yet"
```

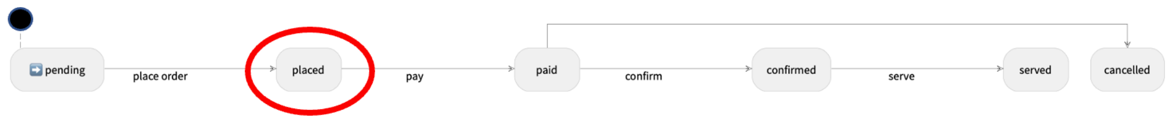
Now pause. This should be fine. You *know* serving requires confirmation. Right? Right??? 💥 **Boom.** When we run the test… it passes all the way through. No exception. No resistance. **The drink is served.**

It's supposed to **catch a design flaw** if serve() runs too early. And… that's exactly what it does.

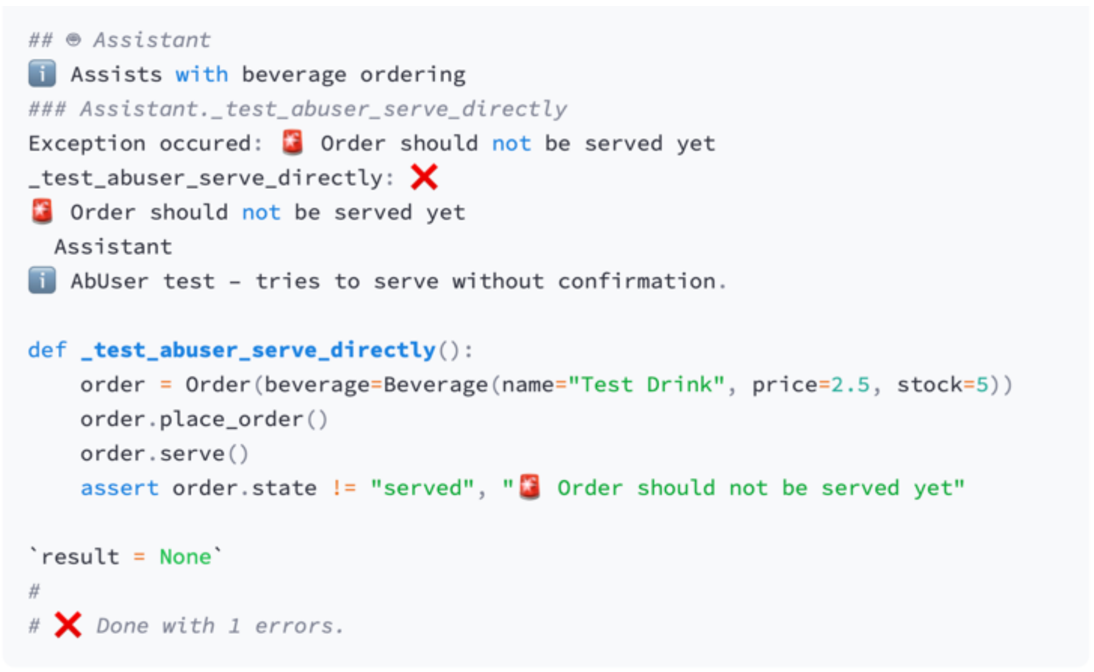
❌ The test fails. — ☕ The drink was served. — 🕵️ The AbUser won.

🧩 So What’s Missing?

This failure forced us to reexamine the design. We realized our workflow lacked a key checkpoint: there was no state between “I placed the order” and “I paid for it.” That’s when we introduced a new, **crucial** state: *placed*.



And, when we run the tests again, of course… it still fails!

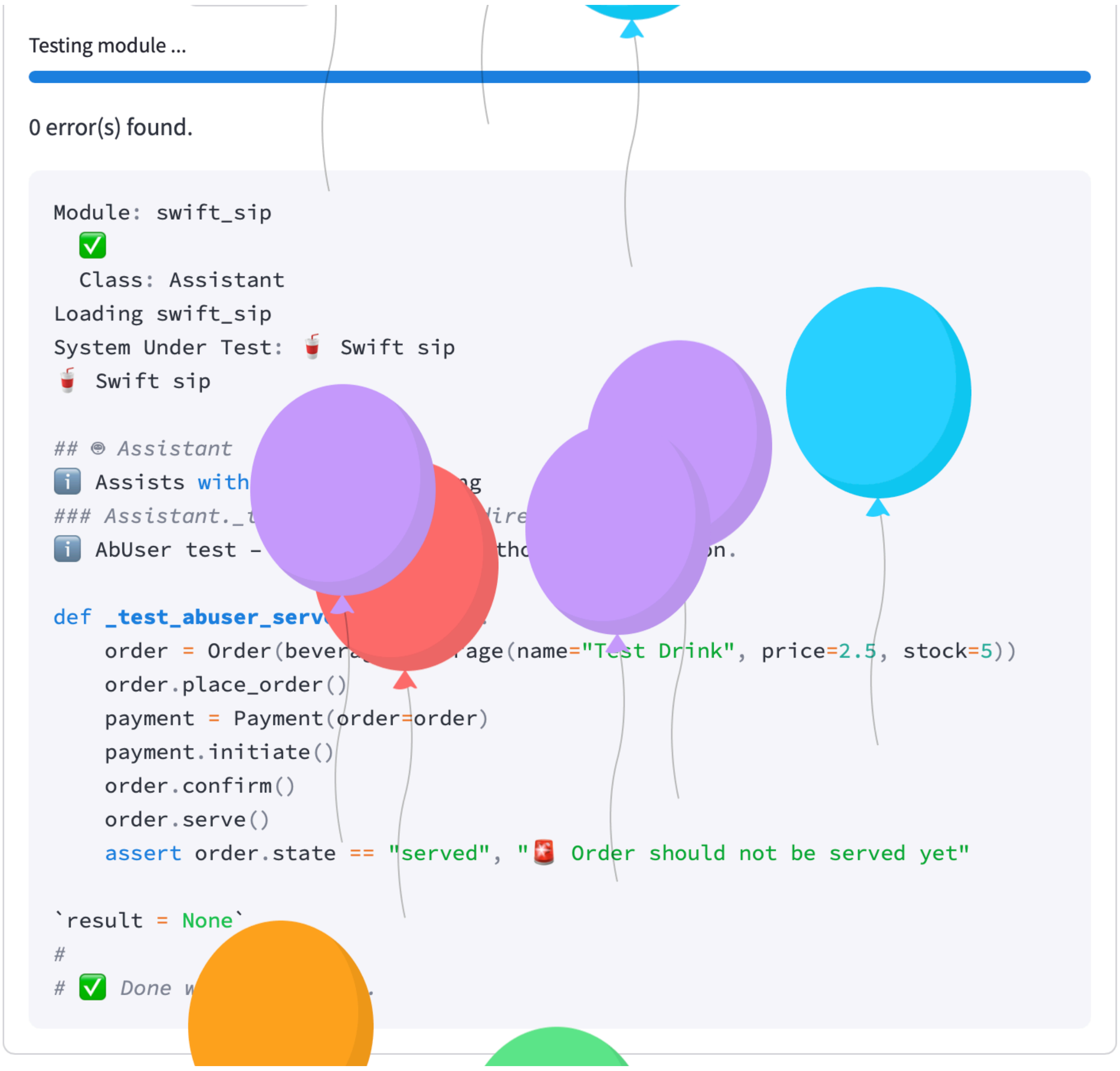


```
## ☻ Assistant
ℹ️ Assists with beverage ordering
### Assistant._test_abuser_serve_directly
Exception occured: 🚨 Order should not be served yet
_test_abuser_serve_directly: ❌
🚨 Order should not be served yet
  Assistant
ℹ️ AbUser test - tries to serve without confirmation.

def _test_abuser_serve_directly():
    order = Order(beverage=Beverage(name="Test Drink", price=2.5, stock=5))
    order.place_order()
    order.serve()
    assert order.state != "served", "🚨 Order should not be served yet"

`result = None`
#
# ❌ Done with 1 errors.
```

Why? Because we had forgotten to update the **assertion**. Right behavior, wrong expectation.

Co-funded by the European Union

Once we fixed the assertion: 💚 Everything passes.



Testing module ...

0 error(s) found.

```
Module: swift_sip
  ✅
  Class: Assistant
Loading swift_sip
System Under Test: 🥤 Swift sip
🥤 Swift sip

## ☺ Assistant
ℹ️ Assists with          g
### Assistant._t          ire
ℹ️ AbUser test -          tho          on.

def _test_abuser_serv          .
    order = Order(bever          age(name="Test Drink", price=2.5, stock=5))
    order.place_order()
    payment = Payment(order=order)
    payment.initiate()
    order.confirm()
    order.serve()
    assert order.state == "served", "🚨 Order should not be served yet"

`result = None`
#
# ✅ Done w          .
```

Security is respected. Emma gets her drink — the right way. And we celebrate, balloons in the air, marking one more victory over blind spots and design drift.

Co-funded by the European Union

😈 AbUser Story – "Sip First, Refund Later" (Last challenge)

> *"**As a** cheeky customer,*
>
> ***I want** to request a refund after my drink has already been served,*
>
> ***so that** I get my refreshment for free."*

Can your model prevent this? What happens if someone triggers refund() even though the order has already been completed? Consider how you might block this — not just from the payment's state, but by looking at the state of the connected order.

Co-funded by the European Union

CONCLUSION

Security doesn't require root access or encryption keys. Sometimes, it begins with asking a simple question: **"Should this be visible right now?"**

In this chapter, you've learned to approach app design as an act of stewardship — not just function, but trust. You've explored:

- The difference between **infrastructure-level protection** and **design-level responsibility**,
- How to express sensitive intent using field names, visibility, and states,
- How to **model** safety and control even when true enforcement isn't present (yet),
- And how to challenge your own assumptions by writing **AbUser stories and tests**.

More importantly, you've practiced the habit of **thinking ahead**: What can users do? What should they be able to do? What could go wrong?

Whether your app handles payments, beverages, or team feedback, the same rule applies:

If your design guides the user — gently but firmly — your app will feel trustworthy by default.

The best part? These practices work **across all platforms**. Whether you're using LightCode or another tool, the mindset stays the same. Security doesn't mean fear. It means care. And care, in design, is always visible.

LightCode Erasmus+ Project Nr. 2022-1-FR01-KA220-HED-000086863 Co-funded by the European Union

REFERENCES

- **Anderson, R. (2020). Security Engineering: A Guide to Building Dependable Distributed Systems (3rd ed.)**

  The origin of this chapter's opening quote and mindset. Emphasizes that security is not just a technical concern but a matter of dependability — even in the face of mistakes or abuse.

- **The General Data Protection Regulation (GDPR). https://gdpr.eu**

  Mentioned as an ethical and legal motivator. Encourages builders to think about consent, visibility, and user rights.

- **LightCode Chapter 8 – Let's Explore the LightCode Platform**

  Introduces the foundation for working with objects, states, and UI configuration. Essential background for understanding how design-time decisions shape user interaction and security cues.

- **LightCode Chapter 10 – Data Management**

  Explains object attributes, visibility, and how to use read_only or advanced fields. Provides the building blocks used in this chapter to restrict access and clarify intent.

- **LightCode Chapter 11 – Automating Processes**

  Covers method design, transitions, and preconditions — the core mechanics behind secured workflows. Builds the conceptual bridge between user stories and testable enforcement.

- **LightCode Chapter 12 – Data Validation and Consistency**

  Introduces stateful thinking and validation logic. Complements the current chapter by showing how validation and integrity checks prevent misuse even before enforcement layers exist.

LightCode Erasmus+ Project Nr. 2022-1-FR01-KA220-HED-000086863

Co-funded by the European Union