Palantir Application Developer Specialist Certification Exam Guide

1. Exam Coverage

The exam tests your ability to **design, build, secure, and operate production applications** in Foundry. You'll demonstrate your ability to own relevant decisions, including where each piece of logic lives, what a change costs the applications already consuming it, and which layer is at fault when something breaks. You are graded on **design, decision, and diagnosis** (not mere recall), and on defending your approach in light of constraints.

You should be able to:

- Model ontology types based on the workflow they enable, and defend your choices against alternative model options.
- Implement the right logic on the right ontology or application primitive to improve functionality, performance, or security.
- Apply security across users, applications, actions, functions, and integrations.
- Write ontology functions and/or applications that surface custom error messages through code and no-code methods.
- Anticipate change impacts and enable migrations that don't lose user edits or result in downtime.

Format & Grading

During this **three-hour**, **hands-on** exam, you will build, extend, and stress-test real Foundry applications in Ontologize's dedicated test environment. Question prompts include task guidance, resources, and acceptance criteria. You will be asked to plan, build, modify, and reflect on your solutions, with corresponding artifacts required for each of these activities. **AI FDE** will be unavailable during portions of the exam.

Ontologize administers and proctors the exam live over Zoom. After registering at ontologize.com/exams, you will receive additional instructions for participating in the exam.

Your work is graded on whether the required design components are present, whether what you build behaves correctly against functional expectations, and whether you can explain and defend your decisions. Points are weighted by domain (not all are equal) and task effort.

If you do not pass, your report shows your rubric performance by domain. Use your weakest domains to focus your next preparation, then return to the relevant sections of this guide and practice those skills in Foundry.

Role boundary

You own	You do not primarily own
The ontology an application reads and writes to, including action types, value types, shared properties, and interfaces	Data pipelines
Functions, automations, and outbound integrations	Model training and AI system design
Workshop applications, object views, and OSDK applications	Platform administration
The application's permission model, and least privilege across its write paths	Enterprise policy and shared identity operations
Application release, monitoring, and recovery	Operating shared observability platforms

2. Exam domains

The exam is organized around ten domains described in greater detail below.

#	Domain
1	Application Data Modeling and Semantics

#	Domain
2	Persistence, Transactions, and Data-Access Logic
3	API Contracts, Generated Clients, and Functional Integrations
4	Application Architecture, Modular Code Organization, and Build Approach
5	Workflow and Application-State Behavior
6	Application-Level Security and Access Control
7	Client Application and Decision Interface Construction
8	Interface Usability, Interaction Design, and Accessibility
9	Runtime, Deployment, and Live Application Operations
10	Testing and Quality Verification

Domain 1: Application Data Modeling and Semantics

Design ontology artifacts to support application workflows, and defend your design decision against alternatives.

- Design an object model backwards from the workflow it supports.
- Determine the grain of an object type and defend the normalization choice.
- Enforce ontology constraints where the platform applies them (e.g., value types and shared properties, conditionally required action parameters).
- Apply security measures that constrain the objects and properties users/groups can see.
- Assess schema change impacts throughout the workflow chain and safely make application/ontology updates.

Self-check: Can I plan and roll-out updates to schemas, security, and ontology primitives with minimal operational impact to production applications?

Domain 2: Persistence, Transactions, and Data-Access Logic

Move data in and out of the Ontology through controlled, auditable paths:

- Select the best ontology read path for a given access pattern.

- Configure the best write-back pattern for the workflow needs (e.g., batch edits over an object set rather than per-object calls, function-backed actions with branched logic).
- Plan schema changes with the right sequence, using branching and object monitors to safely implement them.
- Diagnose slow reads or writes in the place where they execute (e.g., round trips caused by Workshop variables that do not reuse an object set).

Self-check: When a workflow reads or writes slowly, can I tell whether the problem is the path I chose or the way I called it?

Domain 3: API Contracts, Generated Clients, and Functional Integrations

Expose Ontology capabilities as a contract other code can depend on:

- Design the API surface around typed ontology capabilities.
- Define the error contract callers depend on (e.g., error name, error code, and HTTP status as structured responses).
- Classify a change as compatible or breaking, and route it accordingly (e.g., adding an optional input to a function is compatible while adding a required one is not).
- Build an outbound integration that survives failure (e.g., Data Connection webhooks associated with a source, retry and backoff, an idempotency key on unsafe operations).
- Verify a generated client against the contract (e.g., SDK-specific documentation in Developer Console after regeneration).

Self-check: Can I state what a caller relies on (the types, the errors, and the version) and show that my change leaves all three intact?

Domain 4: Application Architecture, Modular Code Organization, and Build Approach

Choose the application and draw the boundaries between its parts:

- Choose the optimal application builder environment for the requirement (e.g., Workshop, OSDK application, custom widget).
- Decompose an application into parts that can be reused (e.g., a landing page routing to embedded Workshop modules, repeated widget compositions extracted into a child module).
- Define the contract between application parts (e.g., function interfaces specifying inputs and outputs).
- Extend a working application without destabilizing it (e.g., an additive use-case pattern that adds object types and links while leaving core ones untouched).

Self-check: Can I justify where I built this, and say which parts still hold when the workflow doubles in scope?

Domain 5: Workflow and Application-State Behavior

Model the process as explicit state, and put each rule on the primitive that owns it:

- Model a process as an explicit state machine (e.g., a Machinery process container per entity type).
- Determine whether a transition is allowed (e.g., comparing the object's current `State` value against the action's allowed input states).
- Place rules on the right primitives (e.g., action types for user-initiated changes, functions for reusable logic).
- Design automation effects.
- Resolve concurrent edits deliberately (e.g., the conflict behavior configured on the object type when pipeline data and user edits disagree).

Self-check: For this transition, what decides whether it is allowed, and where is that decision enforced?

Domain 6: Application-Level Security and Access Control

Apply least privilege across every actor, application, and edit:

- Choose among the platform's access primitives for a stated requirement (e.g., project roles, markings for classification, restricted views).
- Separate resource access from data access (e.g., opening an application is not the same as seeing its data).
- Configure authentication for the integration you are building (e.g., an outbound application for delegated external OAuth).
- Secure the boundary of an OSDK or public application (e.g., a Developer Console application with only the scopes the endpoints require).
- Apply least privilege to edits (e.g., actions as the only edit mechanism with permissions configured per action).

Self-check: For each actor, can I trace resource access and data access, and determine which permissions apply at execution?

Domain 7: Client Application and Decision Interface Construction

Build the interface a user works in, and determine what happens when an action fails:

- Decide whether the requirement is a dashboard or an operational application.
- Construct a multi-view workflow (e.g., pages and overlays for separate workflows, routing that carries state in the URL).
- Implement filtering and drill-down that scales (e.g., a chart or filter selection published as a reusable object set).
- Design the actions experience around what can fail (e.g., pending state, validation responses, input preserved when a function-backed action fails validation).
- Diagnose a control the user cannot use (e.g., tracing a Workshop button to its action type, then to submission criteria, permissions, and the object's current state).

Self-check: When a user cannot press a button, can I trace it from the control to the action type, its submission criteria, the permissions, and the object's current state?

Domain 8: Interface Usability, Interaction Design, and Accessibility

Design the feedback the user gets and the path back from a failure:

- Give the user immediate feedback for every action.
- Design the recovery path (e.g., descriptive failure messages on submission criteria).
- Require confirmation in proportion to consequence (e.g., confirmation on destructive or irreversible changes).
- Eliminate states the user can reach but not escape (e.g., conditional visibility driven by object state).
- Verify that the interface works without a mouse (e.g., every interactive control operable from the keyboard, focus order that follows the visual order).

Self-check: After a failed validation, does the user keep their input, understand what went wrong, and have somewhere to go?

Domain 9: Runtime, Deployment, and Live Application Operations

Release the application, watch what users experience, and recover it when it breaks:

- Package an application for release (e.g., a versioned product in Marketplace carrying its dependent resources).
- Promote a change through environments (e.g., separate Spaces for development, test, and production managed in DevOps).
- Choose the execution primitive for the workload (e.g., serverless functions for atomic request work, compute modules where a container is required, weighing latency against operational cost).
- Triage and recover from a live incident using platform evidence (e.g., deployment health and replica diagnostics, Workflow Builder traces).

Self-check: Given a live incident, can I correlate the symptom to the change that preceded it and roll back safely, including schema changes already applied?

Domain 10: Testing and Quality Verification

Verify the behavior that only appears under permissions, concurrency, and retry:

- Verify a schema-dependent change before it reaches users (e.g., a deployment branch tested against real Ontology data).
- Test the permission model as its own exercise (e.g., running the workflow as a less-privileged user).
- Convert a recurring failure into a check that catches it next time (e.g., a Data Health check or object monitor, a case added to an AIP Logic evaluation suite).
- Test behavior that only appears under concurrency or retry (e.g., two users saving the same record).
- Distinguish a real defect from a flaky or infrastructure failure before acting on it.

Self-check: Have I run this workflow as a less-privileged user, and tested what happens when the same effect is delivered twice?

3. Working Methods

This section describes the “meta-domains” of the exam. It suggests strategies for building, design decisions, and troubleshooting. Practice using these frameworks when preparing for the exam.

The application development lifecycle

1. **Understand the outcome.** What should the user decide, update, approve, or monitor?
2. **Identify the data contract:** which object types, link types, properties, and freshness guarantees are required?
3. **Design the interaction:** what is read-only, what changes state, and what needs confirmation?
4. **Implement the behavior** with functions, action types, automations, or integrations that have clear inputs and outputs.
5. **Secure it.** Apply least privilege so users see and do only what they should.
6. **Validate it** against happy paths, invalid inputs, concurrency, partial failure, retries, and permission differences.

7. **Operate it.** Monitor errors, latency, action outcomes, automation health, and downstream effects.
8. **Change it safely.** Check existing consumers and preserve compatibility where required.

Aim for a design that is elegant, correct, secure, usable, recorded, and possible to operate. Extra components cost points when a simpler build would have done the job.

Core decision areas

Use the requirement, the user, and the cost of getting it wrong to choose the simplest, defensible, safe design.

Decision	Choose based on	Preferred approach	Main risk or evidence to check
Ontology model	What identifies each object, what its properties mean, how many things a link joins, and how people work	Model the objects and links the workflow actually names, before designing any screen	Check stable identity, authoritative properties, editability, visibility, and whether the data supports the relationship
Declarative or function-backed action	Complexity of validation, calculations, branching, linked edits, and maintenance ownership	Declarative rules for direct, permission-checked edits; a function-backed action for computation, branching, or cascading changes	Validate inputs and preconditions at execution. Check permissions, side effects, retries, and audit evidence
Function, action, automation, or integration	Whether the logic reads data, changes state, reacts to events, or calls an external system	Functions for reusable logic, actions for permission-controlled changes, Automate for triggers, integrations for outside systems	Check sign-in, rate limits, whether a retry is safe, who sees failures, and whose permissions apply

Decision	Choose based on	Preferred approach	Main risk or evidence to check
Interactive or background work	User wait time, process duration, retry safety, and need for progress visibility	Keep short, predictable work interactive; move long or failure-prone work to a trackable background flow	Do not hide slow work behind a spinner. Provide status, timeout behavior, partial-failure handling, and a recovery path
Security and access	Who can see data, run actions and functions, and use credentials	Apply least privilege to users, applications, actions, functions, automations, secrets, and integrations	Read access is not write access. Separate user permissions from service or automation execution identity
Safe user experience	Consequence of error, accidental repetition, and clarity of state	Validate close to the user and again at execution; explain unavailable actions; require confirmation for high-consequence changes	Distinguish invalid input, denied access, transient failure, and completed work. Prevent duplicate submission
Change strategy	Number of consumers and whether the change is compatible	Add the new shape, migrate, then remove the old one; version the contract when callers cannot move together	Check the Dependents widget and workflow lineage before changing, not after

Troubleshooting playbook

1. **Classify the symptom:** wrong data, unavailable control, rejected action, timeout, automation failure, integration failure, or permission issue.
2. **Reproduce with the same identity and inputs.** Permission and data context change what you see.

3. **Check the objects and links:** identifiers, property values, how many things the link joins, and how fresh the data is.
4. **Inspect action and function evidence:** validation, preconditions, execution logs, traces, and returned errors.
5. **Check dependencies:** automation triggers, downstream writes, external credentials, and rate limits.
6. **Assess side effects:** partial updates, duplicate actions, and affected consumers.
7. **Repair safely.** Fix the configuration, make retries safe to repeat, and leave the controls in place.
8. **Verify the user-visible result:** state, audit trail, permissions, and recovery behavior.

Granting broad permissions or removing validation to get past an error hides the cause and leaves the defect in production.

4. Practice scenarios

1. **A user can view an object but cannot approve it.** Inspect action-specific authorization and preconditions; read access alone is not write permission.
2. **A button triggers the same external update twice after a timeout.** Make the call safe to repeat and give it a status the application can check, before adding more retries.
3. **A workflow takes several minutes and users repeatedly submit it.** Move long work to a trackable background flow, disable duplicate submission, and expose status.
4. **An action succeeds for one object but fails for another through the same interface.** Compare object state, links, preconditions, and permissions rather than assuming a UI defect.
5. **An automation writes data that the initiating user cannot see.** Distinguish the automation's execution identity from the user's visibility, and review the resulting access model.
6. **A breaking object-property change would affect several existing applications.** Inspect consumers and lineage, introduce a compatible transition or a versioned contract, and migrate deliberately.
7. **A form loses everything the user typed when validation fails.** Preserve the input and map the validation response to the application's feedback; a failed submission should not cost the user their work.

Additional hands-on practice:

Build an end-to-end Foundry workflow, and break it on purpose. Model an object type with a lifecycle, write to it through an action type with submission criteria, drive a step of the process with Automate, and add a Workshop module. Run the workflow as a user with fewer permissions, fire the same automation effect twice, submit a form that fails validation, change a property that another module depends on, and make a live change that you then have to roll back. For each one, identify the layer responsible for the problem and how you knew.

Capability review checklist

- ☐ I can model objects, links, identifiers, and properties before designing any screen.
- ☐ I can defend the grain of an object type and the normalization choice behind it.
- ☐ I can choose among functions, action types, automations, and integrations for a given rule.
- ☐ I can reason about checks, preconditions, retries, and whether a change lands completely or not at all.
- ☐ I can design a clear user experience for long-running and partially failing work.
- ☐ I can separate user permissions from execution identity and external credentials.
- ☐ I can classify an API change as compatible or breaking and route it accordingly.
- ☐ I can diagnose a failure from evidence rather than removing controls at random.
- ☐ I can prevent duplicate submissions and unsafe retries.
- ☐ I can plan a schema change around existing consumers without disruption.