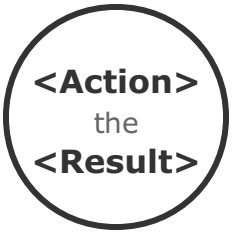
ARO

ACTION • RESULT • OBJECT

The Language Guide

Business Logic as Language



December 2025

Version 1.0

§ Chapter 1: Why ARO? — The Power of Constraints

> “The most powerful thing a language can do is say no.”

* * *

1.1 The Paradox of Less

Every programmer eventually discovers a counterintuitive truth: sometimes the best code is the code you couldn’t write.

General-purpose languages like Python, JavaScript, and Go are magnificent tools. They let you express almost anything. But “almost anything” is precisely the problem. When a language permits infinite variation, every codebase becomes a unique dialect. Reading someone else’s Express.js handler requires decoding their personal philosophy of error handling, their opinions on async/await versus callbacks, their stance on mutation.

ARO takes a different path. It deliberately limits what you can express. No loops. No conditionals. No arbitrary function definitions. Just 24 verbs, a fixed grammar, and a commitment to the happy path.

This isn’t a limitation born of laziness or naivety. It’s a design philosophy with historical precedent:

- **SQL** (1974): You can’t write loops. You describe what you want, not how to get it. Result: the most successful data language in history.
- **Make** (1976): You can’t express arbitrary control flow. You declare dependencies and recipes. Result: still building software 50 years later.
- **Terraform** (2014): You can’t write general programs. You declare desired state. Result: infrastructure-as-code became an industry.
- **Dhall** (2016): You can’t write Turing-complete programs. You get guaranteed termination. Result: configuration that actually validates.

These languages succeeded not despite their constraints but because of them. When a language says “no” to complexity, it creates space for clarity.

* * *

1.2 Anatomy of a Constrained Language

ARO makes specific choices about what it will and won't express:

What ARO Has: - 24 built-in actions (verbs like Extract, Compute, Return, Emit) - A fixed sentence structure: `<Action> the <Result> preposition the <Object>` - Feature sets that respond to events - First-class support for HTTP, files, and sockets - Native compilation to standalone binaries

What ARO Deliberately Lacks: - Traditional if/else (uses `when` guards and `match` expressions instead) - Traditional loops (uses `for each` and collection actions instead of while/recursion) - User-defined functions (feature sets aren't functions) - Complex type system (primitives built-in, complex types from OpenAPI schemas) - Exception handling (happy path only)

This sounds limiting because it is. But limitation is the point. When the 24 built-in actions aren't enough, ARO provides escape hatches—custom actions written in Swift and distributable plugins. But that's a topic for later chapters, after you've learned the language itself.

* * *

1.3 The Honest Trade-offs

We promised fairness. Here it is.

The Pros

Reduced Cognitive Load

When there's only one way to fetch data from a database, code review becomes trivial. You don't debate style. You don't argue about error handling patterns. You don't wonder if the author knows about that "better" approach. There's one approach.

A team of five developers writing ARO will produce code that looks like it was written by one person. A team of five developers writing TypeScript will produce five dialects.

Smaller Attack Surface

Simple APIs have less room for bugs. When you can't write:

```
if (user.role === 'admin' || user.department === currentDepartment) {
    // Complex permission logic with subtle edge cases
}
```

You can't write the subtle bug hiding in that conditional. ARO's permission checks happen in dedicated actions that can be audited once and trusted forever.

Enforced Patterns

ARO doesn't need a style guide. The grammar is the style guide. Every statement follows the same structure. Every feature set has the same shape. This isn't just aesthetics—it's a forcing function for architectural consistency.

Auditability

Consider this ARO feature set:

```
(createUser: User API) {
    <Extract> the <data> from the <request: body>.
    <Validate> the <data> against the <user: schema>.
    <Create> the <user> with <data>.
    <Emit> a <UserCreated: event> with <user>.
    <Return> a <Created: status> with <user>.
}
```

A business analyst can read this. A compliance officer can audit it. A new developer can understand it in seconds. The code is the documentation.

AI-Friendly

Large language models excel at constrained grammars. Ask GPT to write "a Python function that does X" and you'll get endless variation. Ask it to write "an ARO feature set that does X" and the constrained grammar guides it toward consistency.

This matters for AI-assisted development, automated code generation, and the emerging world of agents that write code.

The Cons

Ceiling of Expression

Some problems genuinely need conditionals. Calculating tax brackets. Routing based on user roles. Retrying with exponential backoff. ARO handles these by pushing complexity into custom actions—but that means escaping to Swift, which defeats some of the simplicity benefits.

If your domain is inherently conditional, you'll spend more time writing extensions than writing ARO.

Extension Overhead

When you need a custom action, you need to: 1. Write Swift code 2. Understand ARO's action protocol 3. Register the action 4. Rebuild

This is more friction than adding a function in a general-purpose language. For rapid prototyping, this overhead hurts.

Learning Curve

Thinking in actions feels unnatural at first. Programmers are trained to think in terms of functions, loops, and conditionals. ARO requires unlearning those instincts and relearning a declarative vocabulary.

The payoff comes later, but the initial investment is real.

Ecosystem Immaturity

Stack Overflow doesn't have ARO answers. There's no npm with thousands of packages. The community is small. When you hit a problem, you're often on your own.

This is the tax every new language pays. ARO is paying it now.

Abstraction Leaks

ARO's "happy path only" philosophy is beautiful until something goes wrong. When a database query fails, you get a runtime error message that describes what couldn't be done. Sometimes that's enough. Sometimes you desperately need conditional error handling.

When the abstraction leaks, you feel it acutely.

* * *

1.4 Domain Suitability

Not every domain is equally suited to ARO's constraints. Here's an honest assessment:

Domain	Suitability	Reasoning
Business Logic / CRUD APIs	Excellent	ARO's natural home. Extract data, validate, transform, persist, return. The action vocabulary maps perfectly to business operations.
DevOps / Infrastructure	Good	Declarative actions like `<Provision>`, `<Deploy>`, `<Configure>` map well to infrastructure verbs. The lack of conditionals is less painful because infrastructure should be idempotent anyway.
Build Systems	Promising	Actions as build steps, events as triggers, dependency graphs as feature set chains. Make proved this paradigm works. ARO could extend it.
IoT / Edge Computing	Interesting	Small compiled binaries. Event-driven architecture. Limited resources favor limited languages. Native compilation makes deployment feasible.
System Administration	Moderate	Scripts often need conditionals ("if file exists, do X"). ARO would require more custom actions than might be practical.
Data Science / ML	Poor	Iteration is fundamental. Exploration requires flexibility. The REPL-driven workflow clashes with ARO's compile-run cycle.
Game Development	Poor	Tight loops for physics. Real-time requirements. Mutable state everywhere. ARO's constraints actively hurt here.

* * *

1.5 When Simplicity Wins

Here's a mental model for when ARO's trade-offs favor you:

The Bug Equation

```
Bugs ∝ (API Surface) × (Complexity) × (Mutability)
```

ARO attacks all three factors: - **Smaller API surface**: 24 actions vs. infinite function possibilities - **Reduced complexity**: No control flow means fewer execution paths - **Limited mutability**: Actions transform and return; they don't mutate shared state (though explicit shared repositories exist for business domain data and can be safely used across feature sets)

Consider a real comparison. A typical Express.js endpoint handler might span 500 lines: request parsing, authentication, authorization, validation, business logic, database queries, error handling, response formatting, logging. Each line is a potential bug. Each conditional doubles the test matrix.

The equivalent ARO feature set might be 15 lines. Each line is a single action with well-defined semantics. The test matrix is manageable.

When the Trade-off Favors Constraints

1. **Team Size > 5**: Consistency becomes more valuable than flexibility when more people touch the code.
2. **Regulatory Environments**: Healthcare, finance, and government need auditable code. ARO reads like documentation.
3. **Long-Lived Systems**: Code is read more than it's written. Readability beats cleverness over a 10-year lifespan.
4. **AI-Assisted Development**: If agents will modify your code, constrained grammars make their output more predictable.
5. **High-Reliability Requirements**: Fewer execution paths mean fewer untested paths. Constraints prevent entire categories of bugs.

* * *

1.6 When to Look Elsewhere

ARO isn't the right choice for:

Real-Time Systems

Microsecond precision requires low-level control. ARO's abstraction layer adds latency you can't afford.

Heavy Algorithmic Work

Sorting algorithms. Graph traversal. Machine learning training loops. These need iteration and fine-grained control. ARO would require escaping to Swift for virtually everything.

Exploratory Programming

Some domains benefit from REPL-driven experimentation. Data analysis. Prototyping. Research. ARO's compile-run cycle creates friction that hurts exploration.

Teams with Strong Existing Patterns

If your team already has battle-tested Go services with excellent patterns, introducing ARO adds cognitive overhead without clear benefit. The value of constraints diminishes when you already have good constraints.

* * *

1.7 The Missing Parts — A Pre-Alpha Disclaimer

> ARO is pre-alpha software. This section exists because honesty is a feature.

What's Not Here Yet

Conditional Logic

ARO has no traditional if/else. Instead, it provides guarded statements with `when` clauses and `match` expressions for pattern matching. These cover most conditional needs while maintaining ARO's declarative style, though complex nested conditionals still require custom actions.

Iteration

ARO provides `for each` loops for serial iteration and `parallel for each` for concurrent processing. Collection actions like `<Filter>`, `<Transform>`, `<Sum>`, `<Sort>`, and others handle common operations declaratively. While not as flexible as general-purpose loops, these constructs cover most business processing needs.

Type System

ARO has four built-in primitive types (String, Integer, Float, Boolean) and collection types (List, Map). Complex types—records and enums—are defined in your `openapi.yaml` file's components/schemas section. This "single source of truth" approach means OpenAPI defines both your HTTP routes and your data types. Runtime type checking validates data against these schemas, with errors reported in business terms rather than technical stack traces.

Debugging Tools

No step debugger exists. Limited introspection. Debugging means reading logs and error messages. IDE integration is primitive—syntax highlighting exists; language server protocol support is in progress.

Standard Library

24 actions is a starting point. The vocabulary will grow. File operations beyond basic I/O. Database abstractions. Authentication patterns. These need to be built.

Documentation

You're reading the first attempt. Expect gaps, errors, and outdated information.

What Will Change

This is pre-alpha software. Expect:

- Action signatures may evolve incompatibly
- Preposition semantics are still being refined
- Native compilation is experimental
- Plugin API is unstable
- Error messages will improve (they're rough now)

The Commitment

This book is maintained alongside the language. We use AI-assisted tooling to keep chapters synchronized with the codebase. When ARO changes, the book follows.

For the latest version, check the repository. For the authoritative specification, read the proposals in `Proposals/`.

> This chapter reflects ARO as of December 2025.
> Language version: pre-alpha
> Book revision: 0.1

Next: Chapter 2 — The ARO Mental Model

§ Chapter 2: The ARO Mental Model

> “Every statement is a sentence. Every program is a story.”

* * *

2.1 Thinking in Sentences

Most programming languages ask you to think in terms of instructions. You assign variables, call functions, iterate over collections, and handle exceptions. The cognitive overhead accumulates with every new construct you learn. ARO takes a radically different approach: it asks you to think in terms of sentences.

Consider how you might explain a business process to a colleague. You would not say “iterate over the user collection, extract the email field, and invoke the send method on each result.” You would say “send an email to each user.” Natural language describes what should happen, not the mechanical steps to achieve it.

This observation forms the foundation of ARO’s design. The language constrains you to express operations as sentences, each following a consistent grammatical structure. This constraint is not a limitation but a clarifying force. When every line of code must fit the same pattern, you cannot hide complexity in clever abstractions or obscure syntax. The code reads like a description of the process itself.

The fundamental pattern is simple: an action verb, followed by a result noun, followed by a preposition and an object. Every statement in ARO follows this structure without exception. There are no special cases, no alternative syntaxes, no shortcuts that break the pattern. This uniformity means that once you understand how to read one ARO statement, you can read any ARO statement in any program.

* * *

2.2 The FDD Heritage

ARO’s Action-Result-Object pattern did not emerge from thin air. It traces back to 1997, when Jeff De Luca and Peter Coad faced a crisis on a massive banking project for United Overseas

Bank in Singapore. Developers from around the world could not understand the business requirements, and business people could not understand the developers. The project was failing.

De Luca and Coad asked a radical question: what if they structured the entire development process around something everyone could understand—features? They called their approach Feature-Driven Development, or FDD, and it worked. The banking project was saved.

At the heart of FDD was a deceptively simple idea. Every feature should be expressed as an action performed on a result for a business object. The formula was: Action the Result for the Object. “Calculate the total for the shopping cart.” “Validate the credentials for the user.” “Send the notification to the customer.” This was not just a naming convention. It was a language—a way for product managers, business analysts, and developers to speak the same tongue.

Features grouped into Feature Sets, collections of related functionality that together delivered a business capability. A “User Management” feature set might contain “Create the account for the user,” “Validate the credentials for the user,” “Update the profile for the user,” and “Reset the password for the user.” Visual dashboards called Parking Lots showed the status of all feature sets at a glance. Anyone—even executives without technical backgrounds—could see what was done, what was in progress, and what was coming.

In 2002, Stephen R. Palmer and John M. Felsing published “A Practical Guide to Feature-Driven Development,” documenting FDD’s principles. But by then, the Agile movement had begun. Scrum and Kanban captured the industry’s attention, and FDD faded into a footnote in methodology discussions.

FDD did not die because it was wrong. It faded because it was ahead of its time. In 1997, there was no technology to make feature-language into real code. Developers still had to translate those beautifully clear feature descriptions into Python, Java, or C#. The gap between specification and implementation remained.

Twenty-five years later, Large Language Models changed everything. AI that understands natural language made feature-language suddenly practical. When a statement like “Extract the user from the request” is both the specification and the code, there is no translation gap. The AI does not have to guess what you mean. The structure is the specification.

ARO is the realization of FDD’s vision. The Action-Result-Object pattern that De Luca and Coad invented for documentation becomes an actual programming language. Feature sets that once existed only in project plans now execute directly. The bridge between business and code, imagined in 1997, finally exists.

* * *

2.3 The Three Components

Every ARO statement consists of exactly three semantic components that work together to express a complete operation.

The first component is the **Action**, which represents what you want to do. Actions are verbs—words like Extract, Create, Return, Validate, or Store. They describe the operation in terms of its intent rather than its implementation. When you write an Extract action, you are expressing that you want to pull data out of something. You are not specifying how that extraction happens, what data structures are involved, or what error handling should occur. The verb captures the essence of the operation.

The second component is the **Result**, which represents what you get back. Every action produces something, even if that something is simply a confirmation that the operation succeeded. The result is the variable that will hold the produced value. You give it a name that describes what it represents in your domain. If you are extracting a user identifier from a request, you name the result “user-id” because that is what it is. The name becomes part of the program’s documentation, making the code self-describing.

The third component is the **Object**, which represents the input or context for the action. Objects are introduced by prepositions—words like “from,” “with,” “into,” and “against.” The choice of preposition is significant because it communicates the relationship between the action and its input. When you extract something “from” a source, the preposition indicates that data is moving from the object toward the result. When you store something “into” a repository, the preposition indicates that data is moving from the result toward the object.

These three components combine to form a complete sentence. Consider this statement:

```
<Extract> the <user-id> from the <pathParameters: id>.
```

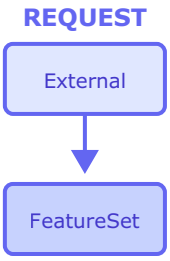
The action is Extract, telling us that we are pulling data out of something. The result is user-id, which will hold the extracted value. The object is pathParameters with a qualifier of id, indicating where the data comes from. The preposition “from” establishes that data flows from the path parameters into the user-id variable.

Reading this statement aloud produces natural English: “Extract the user-id from the path parameters id.” No translation is needed between the code and its meaning.

2.4 Understanding Semantic Roles

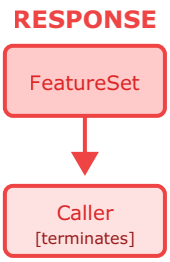
Actions in ARO are not arbitrary verbs. Each action carries a semantic role that describes the direction of data flow. The runtime automatically classifies actions based on their verbs, and this classification has important implications for how the program executes.

The first role is called **REQUEST**, which describes actions that bring data from outside the feature set into the current context. Think of request actions as inbound data flows. They reach out to external sources—HTTP requests, databases, files, or other services—and pull data into your local scope. Verbs like Extract, Retrieve, Fetch, and Read all carry the request role. When you use one of these verbs, you are declaring that you need data from somewhere else.

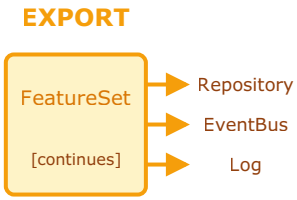


The second role is called **OWN**, which describes actions that transform data already present in the current context. These actions neither bring data in from outside nor send it out. They work entirely within the boundaries of the current feature set, taking existing values and producing new ones. Verbs like Create, Compute, Validate, Transform, and Merge carry the own role. These are the workhorse actions that implement your business logic.

The third role is called **RESPONSE**, which describes actions that send data out of the feature set to the caller. Response actions are outbound data flows that terminate the current execution path. The Return verb is the most common example, sending a response back to whoever invoked the feature set. Other verbs like Throw and Respond also carry this role.



The fourth role is called **EXPORT**, which describes actions that make data available beyond the current execution without terminating it. Unlike response actions, export actions allow execution to continue. They persist data to repositories, emit events for other handlers to process, or publish values for access within the same business activity. Verbs like Store, Emit, Publish, and Log carry the export role.



Understanding these roles helps you reason about your programs. A typical feature set begins with request actions that gather the needed data, follows with own actions that process and transform that data, includes export actions that persist results or notify other parts of the

system, and concludes with a response action that sends the final result to the caller. This pattern emerges naturally from the semantic roles.

2.5 Why Uniformity Matters

The uniform structure of ARO statements might seem restrictive at first. Why force every operation into the same grammatical pattern? The answer lies in what uniformity enables.

When every statement follows the same structure, reading code becomes effortless. You never wonder what syntax you are looking at or what special rules apply. Every line is an action, a result, a preposition, and an object. Your eyes learn to parse this pattern automatically, and soon you can scan ARO code as quickly as you scan prose.

Uniformity also benefits writing. You never face the question of how to express something. The grammar constrains you to the action-result-object pattern, and within that pattern, you simply choose the verb that matches your intent and the names that describe your data. There are no style debates about whether to use a function or a method, whether to inline an expression or extract it, or whether to use early returns or guard clauses. The grammar makes these decisions for you.

For tools, uniformity is transformative. Parsers, analyzers, code generators, and formatters all work identically across every statement because there are no special cases. An AI assistant can generate or verify ARO code with high confidence because the constrained grammar limits the space of possible outputs. Refactoring tools can manipulate code safely because the structure is completely predictable.

Perhaps most importantly, uniformity benefits teams. When five developers write ARO code, the result looks like it was written by one person. There are no personal styles, no preferred idioms, no clever tricks that only the author understands. The code is what it is, expressed in the only way the grammar permits.

2.6 The Declarative Shift

Traditional programming is imperative. You tell the computer how to do something by listing the steps it should follow. Fetch the request body. Parse the JSON. Check if the email field exists. Validate the format. Query the database. Handle the not-found case. Construct the response. Send it back. Each step is a command, and you must get every command right in the right order.

ARO is declarative. You tell the computer what you want to happen, and the runtime figures out how to make it happen. This shift has profound implications for how you think about programming.

Consider a typical operation: getting a user by their identifier. In an imperative style, you would write code that explicitly handles each step and each potential failure. In ARO, you write:

```
(getUser: User API) {
    <Extract> the <user-id> from the <pathParameters: id>.
    <Retrieve> the <user> from the <user-repository> where <id> is <user-id>.
    <Return> an <OK: status> with <user>.
}
```

This code does not explain how to extract the identifier from the path. It does not specify what happens if the identifier is missing. It does not detail how to query the repository or what to do if the user is not found. It simply states what should happen when everything works correctly.

The runtime handles everything else. If the extraction fails because the path parameter is missing, the runtime produces an appropriate error message. If the retrieval fails because no user has that identifier, the runtime produces a not-found response. You do not write error handling code because there is nothing to handle. You express the successful case, and the runtime handles the unsuccessful cases.

This is the “happy path” philosophy. Your code contains only the path through the logic when everything succeeds. The runtime, which is tested and trusted, handles the paths where things fail. This dramatically reduces the amount of code you write and eliminates entire categories of bugs that arise from incorrect error handling.

2.7 Data as Transformation

The ARO mental model encourages you to think about data as a series of transformations rather than as mutable state that you manipulate over time.

Each statement in a feature set transforms the available data. The first statement might extract a value from the request, making that value available to subsequent statements. The second statement might use that value to retrieve something from a repository, making the retrieved data available. The third statement might combine several values into a new object. The fourth might persist that object.

At each step, you are not modifying existing data. You are producing new data from existing data. The symbol table grows as execution proceeds, accumulating the results of each transformation. Nothing is overwritten or mutated. If you need a different value, you create a new binding with a new name.

This immutability has practical benefits. You can always trace where a value came from by following the chain of transformations backward. You never face the confusion of a variable changing unexpectedly because some distant code modified it. Debugging becomes straightforward because the state at any point is simply the accumulation of all previous results.

Think of a feature set as a pipeline. Data enters at one end, flows through a series of transformations, and exits at the other end. Each transformation is a pure function of its inputs, producing outputs without side effects on the local state. Export actions have external side effects—they persist data or emit events—but they do not change the local symbol table in unexpected ways.

2.8 Variables and Binding

When an action produces a result, that result is bound to a name. The binding is permanent within the scope of the feature set. You cannot rebind a name to a different value.

This design prevents a common source of bugs: the accidental reuse of a variable name for a different purpose. In many languages, you might write code like this pseudocode: “set x to 1, then later set x to 2, then later use x expecting it to be 1.” The bug is subtle and easy to overlook. ARO makes this impossible. If you try to bind a name that is already bound, the compiler rejects your code.

The practical implication is that you must choose descriptive names for your results. You cannot use generic names like “temp” or “result” for everything because you cannot reuse them. This constraint pushes you toward self-documenting code. Instead of “result,” you write “validated-user-data.” Instead of “temp,” you write “calculated-total.”

Subsequent statements reference bound names using angle brackets. When you write a statement that includes something like `with <user-data>`, you are referencing the value that was bound to the name “user-data” by a previous statement. If no previous statement bound that name, the runtime reports an error.

2.9 Comparing Approaches

To understand the ARO mental model fully, it helps to contrast it with other programming paradigms.

Imperative programming focuses on how to accomplish something. You write step-by-step instructions: do this, then do that, check this condition, loop over this collection. The computer follows your instructions exactly. The power is that you have complete control. The cost is that you must handle every detail.

Functional programming focuses on what relationships exist between inputs and outputs. You compose functions that transform data, building complex behaviors from simple, pure functions. The power is that pure functions are easy to test and reason about. The cost is that real-world programs have side effects that pure functions cannot express directly.

Object-oriented programming focuses on what entities exist and how they interact. You model your domain as objects with state and behavior, passing messages between them. The power is that objects map naturally to real-world concepts. The cost is that complex object graphs become difficult to understand and modify.

ARO takes a different approach. It focuses on what should happen in business terms. You express operations as sentences that describe business activities. The power is that the code directly reflects the business process, readable by anyone who understands the domain. The cost is that some technical operations do not fit naturally into sentence form and must be pushed into custom actions.

Each approach has its place. ARO excels at expressing business logic—the rules and processes that define what a system does. It is less suited to algorithmic work, systems programming, or exploratory data analysis. Knowing when to use ARO and when to use other approaches is part of becoming proficient with the language.

2.10 From Understanding to Practice

The mental model described in this chapter is the foundation for everything that follows. Every chapter in this guide builds on the concepts introduced here: actions and their semantic roles, results and their bindings, objects and their prepositions, the uniform structure of statements, and the declarative approach to expressing logic.

As you continue reading, keep these principles in mind. When you encounter a new feature or pattern, ask yourself how it fits into the mental model. How does this feature express data

transformation? What semantic role does this action carry? How does this pattern leverage the uniform structure of statements?

The goal is not to memorize rules but to internalize a way of thinking. Once the mental model becomes natural, writing ARO code becomes as straightforward as describing a business process in conversation. The language disappears, and only the intent remains.

* * *

Next: Chapter 3 — Getting Started

§ Chapter 3: Getting Started

> “The best way to learn a language is to write something.”

* * *

3.1 Installation

ARO is implemented in Swift and distributed as a command-line tool. The implementation requires Swift 6.0 or later, which means you will need macOS 14 or a recent Linux distribution with Swift installed. On macOS, Xcode 16 provides everything you need. On Linux, you can install Swift from swift.org.

The simplest way to get ARO is to build it from source. Clone the repository and run Swift Package Manager’s build command. For development work, a debug build is sufficient and compiles faster. For running applications you intend to deploy, a release build with optimizations produces a significantly smaller and faster binary.

After building, you will find the `aro` executable in the `.build` directory. To use it conveniently from any directory, add this location to your shell’s PATH environment variable. Once configured, running `aro --help` should display the available subcommands, confirming that the installation was successful.

The ARO command-line tool provides four primary subcommands. The `run` command compiles and executes an application in a single step, which is what you will use most during development. The `check` command validates source files without running them, useful for catching errors before execution. The `build` command compiles an application to a native binary for deployment. The `compile` command produces intermediate output for tooling integration.

* * *

3.2 Your First Application

ARO applications are directories, not single files. This might seem unusual if you are accustomed to languages where a single source file can be a complete program. The

directory-based approach exists because ARO applications typically consist of multiple feature sets spread across several files, and the runtime automatically discovers and loads all of them.

To create an application, start by making a directory. The name you choose becomes the application name. Inside this directory, create a file named `main.aro` that contains your application's entry point.

Every ARO application must have exactly one `Application-Start` feature set. This is where execution begins. The runtime looks for this specific feature set name and executes it when the application launches. Having zero entry points is an error because there would be nothing to run. Having multiple entry points is also an error because the runtime would not know which one to execute first.

A minimal entry point creates a value, does something with it, and returns a status indicating success. The `Create` action produces a new value and binds it to a name. The `Log` action writes output to the console. The `Return` action signals completion. Every feature set should end with a Return action to indicate whether it completed successfully or encountered a problem.

The second part of a feature set declaration, after the colon, is called the business activity. This is a descriptive label that explains what the feature set represents in business terms. For an entry point, common choices include "Entry Point," "Application," "System Initialization," or something more domain-specific like "Order Processing System."

When you run your application using the `aro run` command followed by the path to your application directory, the runtime discovers all ARO source files, compiles them, identifies the entry point, and executes it. If everything works correctly, you see your output and the application terminates. If something goes wrong, you see an error message explaining what happened and where.

* * *

3.3 Understanding Application Structure

The directory-based organization reflects how ARO applications are intended to be structured. A typical application has a main file containing the entry point and lifecycle handlers, plus additional files containing feature sets organized by domain or purpose.

The runtime automatically discovers all files with the `.aro` extension in the application directory. You do not need import statements or explicit file references. When the runtime

starts, it scans the directory, parses each file, validates the combined set of feature sets, and registers them with the event bus. This means you can add new feature sets simply by creating new files, and they become available immediately.

The automatic discovery has an important implication: all feature sets are globally visible within an application. A feature set in one file can emit an event that triggers a feature set in another file without any explicit connection between them. This loose coupling is intentional. It allows you to organize code however makes sense for your project without worrying about dependency graphs between files.

There is one constraint on this freedom: exactly one `Application-Start` must exist across all files. The runtime enforces this during startup and reports an error if the constraint is violated. Similarly, you can have at most one `Application-End: Success` for handling graceful shutdown and at most one `Application-End: Error` for handling crash scenarios. See Chapter 10 for complete lifecycle details.

For applications that expose HTTP APIs, you will typically include an `openapi.yaml` file in the application directory. This file defines the API contract using the OpenAPI specification. When present, the runtime uses it to configure HTTP routing, matching incoming requests to feature sets based on operation identifiers defined in the contract. Without this file, no HTTP server starts. This is deliberate: ARO follows a contract-first approach where the API specification drives the implementation rather than the other way around.

* * *

3.4 The Command-Line Interface

The `aro run` command is your primary tool during development. It takes a path to an application directory, compiles all source files, validates the application structure, and executes the entry point. The process is designed to be fast enough that you can iterate quickly, making changes and rerunning to see the effects.

The verbose flag adds detailed output showing what the runtime is doing. You can see which files were discovered, how they were parsed, which feature sets were registered, and how events flow during execution. This visibility is invaluable when debugging issues or understanding how an application behaves.

For servers and other long-running applications, you need to use the Keepalive action to prevent the application from terminating after the entry point completes. Without it, the

runtime executes the entry point and exits, which is fine for batch processes but not for services that need to wait for incoming requests.

The `aro check` command validates source files without executing them. Think of it as a sophisticated linter that catches not just syntax errors but also semantic issues like undefined variables, duplicate bindings, and type mismatches. The output categorizes issues as errors, which prevent compilation, and warnings, which indicate potential problems but do not block execution.

Running check before run can save time because compilation errors are often easier to understand when presented in isolation rather than mixed with runtime output. Many developers add a check step to their continuous integration pipelines to catch errors before code is merged.

The `aro build` command compiles an application to a native binary. Unlike interpreted execution with `run`, the build command generates LLVM IR from your ARO source, compiles it to machine code, and links the result with the ARO runtime library. The output is a standalone executable that can run on any compatible system without requiring ARO to be installed.

Native compilation is particularly useful for deployment. The resulting binaries start almost instantaneously because there is no parsing or compilation at runtime. The optimize flag enables compiler optimizations that can significantly improve performance for compute-intensive applications.

* * *

3.5 Development Workflow

A typical development session follows a predictable rhythm. You write or modify code in your editor, run the check command to catch obvious errors, then run the application to verify the behavior. When something does not work as expected, you examine the error message, adjust the code, and repeat.

The error messages are designed to be helpful. Parse errors report the file, line number, and character position where parsing failed, along with an explanation of what was expected. Semantic errors explain what semantic rule was violated and often suggest corrections based on similar identifiers in scope. Runtime errors describe what operation could not be completed, expressed in the business terms of your statements rather than implementation details.

Verbose mode becomes particularly useful when debugging event-driven behavior. Because feature sets communicate through events, it can sometimes be unclear why a particular feature set did or did not execute. The verbose output shows every event emission and every handler invocation, making the flow visible.

As your application grows, you will naturally organize feature sets into separate files. Common patterns include grouping by domain (users.aro, orders.aro), by concern (handlers.aro, notifications.aro), or by layer (api.aro, events.aro). The specific organization matters less than consistency. Choose a pattern that makes sense for your team and stick with it.

* * *

3.6 Error Messages and Debugging

ARO prioritizes readable error messages because debugging time is a significant portion of development effort. The goal is for error messages to tell you not just what went wrong but also where and why, with enough context to fix the problem.

Parse errors occur when the source code does not match the grammar. The parser reports the file, line number, and character position where parsing failed, along with a description of what it expected to find. These errors typically indicate typos, missing punctuation, or malformed statements. The statement structure is so regular that once you internalize it, parse errors become rare.

Semantic errors occur when the source code is syntactically valid but violates a semantic rule. Common examples include referencing a variable that was never defined, attempting to rebind a name that is already bound, or using an action with an incompatible preposition. The analyzer tries to provide helpful context, such as listing similar identifiers that might be what you meant.

Runtime errors occur during execution when an operation cannot be completed. These are described in terms of what the statement was trying to accomplish. If a Retrieve action cannot find the requested record, the error message says so using the names from your code, not internal implementation details. This makes runtime errors easier to understand and correlate with specific statements.

When debugging complex issues, the debug flag provides additional internal information. This includes the state of symbol tables, the contents of events, and the sequence of action

executions. This level of detail is rarely needed but can be invaluable when tracking down subtle problems.

* * *

3.7 From Here

You now understand how to install ARO, create applications, and use the command-line tools. You have seen the basic structure of an ARO program and understand why applications are directories rather than single files.

The next chapter examines the syntax of statements in detail, explaining each component of the action-result-object pattern. Understanding this pattern deeply is essential because every statement you write follows it. After that, chapter five covers feature sets, exploring how they are triggered, how they communicate through events, and how they form the building blocks of applications.

The best way to proceed is to experiment. Create a small application, add feature sets, emit events, and observe what happens. The language is designed to be discoverable. If you try something that does not work, the error message should guide you toward what does.

* * *

Next: Chapter 4 — Anatomy of a Statement

§ Chapter 4: Anatomy of a Statement

> “Grammar is the logic of speech.”

* * *

4.1 The Universal Structure

Every ARO statement follows the same grammatical pattern. This is not a guideline or a convention that you might occasionally break. It is an invariant property of the language, enforced by the parser, and fundamental to how ARO works.

The pattern consists of an action, followed by an article, followed by a result, followed by a preposition, followed by another article, followed by an object, and terminated by a period. Within this structure, there are optional elements—qualifiers, literal values, where clauses, and when conditions—but the core pattern is always present.

Understanding this pattern deeply is essential because it shapes how you think about expressing operations in ARO. When you internalize the pattern, writing ARO code becomes as natural as writing sentences. When you fight against the pattern, trying to express operations that do not fit, you struggle unnecessarily. The pattern is not a constraint to overcome but a tool to master.

Let us examine each component in detail, understanding not just what it is but why it exists and how it contributes to the expressiveness of the language.

4.2 Actions: The Verb of Your Sentence

An action is a verb enclosed in angle brackets. It tells the reader what operation the statement performs. Actions are the most prominent part of any ARO statement because they appear at

the beginning and because they carry semantic meaning that affects how the runtime behaves.

When you write an action, you are choosing from a vocabulary of approximately two dozen built-in verbs, each representing a fundamental operation. The choice of verb is significant because each verb carries a semantic role that determines the direction of data flow. When you choose Extract, you are telling the runtime that you want to pull data from an external source into the current context. When you choose Return, you are telling the runtime that you want to send data out to the caller and terminate execution.

The verbs are case-sensitive. Extract is a valid action. extract is not. This case sensitivity is deliberate: it makes actions visually distinctive from other identifiers in your code, and it aligns with the convention that actions are proper verbs deserving of capitalization.

The built-in actions cover the operations that virtually every business application needs. You can extract data from requests, retrieve data from repositories, create new values, validate inputs against schemas, transform data between formats, store data persistently, emit events for other handlers, and return responses to callers. When these built-in actions are insufficient for your needs, you can create custom actions in Swift, extending the vocabulary of the language for your specific domain.

The power of actions comes from their abstraction. When you write a Retrieve action, you are not specifying whether the data comes from an in-memory store, a relational database, a document database, or an external service. You are expressing the intent to retrieve data from a named repository. The runtime, or a custom action implementation, handles the details. This abstraction allows your ARO code to remain focused on business logic while technical concerns are handled elsewhere.

4.3 Results: The Noun You Create

The result is the variable that will hold the value produced by the action. It appears after the action and its article, enclosed in angle brackets. The result is where the output of the operation lands, giving it a name that you can reference in subsequent statements.

Choosing good result names is one of the most important skills in writing ARO code. The name you choose becomes part of the program's documentation. It appears in error messages when something goes wrong. It serves as the identifier that subsequent statements use to reference the value. A well-chosen name makes the code self-explanatory; a poorly chosen name obscures intent.

Consider the difference between naming a result “x” versus naming it “user-email-address.” The first name tells you nothing about what the value represents. The second name tells you exactly what you are dealing with. Because ARO does not allow you to rebind names, you cannot use generic names for everything. This constraint pushes you toward descriptive names, which in turn makes your code more readable.

Results can include type qualifiers, written after a colon. When you write a result like “user-id: String” you are documenting that the result is expected to be a string. Currently, ARO uses runtime typing, so these qualifiers do not affect execution. However, they serve as documentation for readers and may enable static type checking in future versions of the language. Using qualifiers is optional but recommended for results whose types are not obvious from context.

ARO allows hyphenated identifiers, which is unusual among programming languages. This feature exists because hyphenated names often read more naturally than camelCase or snake_case for business concepts. “user-email-address” reads more like natural language than “userEmailAddress” or “user_email_address.” You can choose whichever style you prefer, but the language supports hyphens for those who want them.

4.4 Objects: The Context You Operate On

The object is the input or context for the action. It appears after the preposition, enclosed in angle brackets. The object provides the data or reference that the action operates on.

Objects are introduced by prepositions, and the choice of preposition is significant. The preposition communicates the relationship between the action and its object. Different prepositions imply different types of operations, and the runtime uses this information to understand data flow.

Like results, objects can include qualifiers. When you write an object like “request: body” you are specifying that you want the body property of the request. Qualifiers allow you to navigate into nested structures. You can chain qualifiers to access deeply nested properties, writing something like “user: address.city” to access the city property of the address property of the user.

The distinction between results and objects is fundamental. Results are outputs—the values produced by actions, which become available for subsequent statements. Objects are inputs—the values consumed by actions, which must have been produced by previous statements or be available from the execution context. This input-output distinction is how data flows through a feature set.

4.5 Prepositions: The Relationships Between Things

Prepositions are small words that carry large meaning. In ARO, prepositions connect actions to their objects while communicating the nature of that connection. The language supports ten prepositions:

Preposition	Meaning	Common Actions
`from`	Source extraction	Extract, Retrieve, Fetch, Read
`with`	Accompaniment/provision	Create, Return, Emit
`for`	Purpose/target	Compute, Return, Log
`to`	Destination	Send, Write
`into`	Insertion	Store
`against`	Comparison/validation	Validate, Compare
`via`	Intermediate channel	Fetch (with proxy)
`on`	Location/attachment	Listen, Start
`at`	Position/placement	CreateDirectory, Make
`as`	Type annotation	Filter, Reduce, Map

Choosing the right preposition makes your code clearer and more accurate. When you extract a user identifier from the path parameters, “from” is the natural choice. When you create a user with provided data, “with” is the natural choice. When you store a user into a repository, “into” is the natural choice. Let the semantics of your operation guide your choice of preposition.

> **See Appendix B** for complete preposition semantics with examples.

4.6 Articles: The Grammar Connectors

Articles—“the,” “a,” and “an”—appear between the action and the result, and between the preposition and the object. They serve a grammatical purpose, making statements read like natural English sentences.

The choice of article does not affect the semantics of the statement. Whether you write “the user” or “a user” has no impact on how the statement executes. However, the choice can affect readability. In general, use “the” when referring to a specific, known thing, and use “a” or “an” when introducing something new or when the thing is one of many possible things.

For results, “the” is usually the appropriate choice because you are creating a specific binding. For objects, “the” is also usually appropriate because you are referring to something specific. For return statuses, “an” or “a” often reads more naturally—“Return an OK status” rather than “Return the OK status.”

The important thing is to be consistent within your codebase. Whether you prefer “the” everywhere or vary your articles for readability, stick with your choice so that readers can focus on the meaning rather than the grammatical choices.

4.7 Literal Values

Some statements include literal values—strings, numbers, booleans, arrays, or objects. Literal values provide concrete data within the statement rather than referencing previously bound variables.

String literals are enclosed in double quotes. You can include special characters using escape sequences: backslash-n for newline, backslash-t for tab, backslash followed by a quote for a literal quote character. Strings can contain any text and are commonly used for messages, paths, and configuration values.

Number literals can be integers or floating-point values. Integers are written as sequences of digits, optionally preceded by a minus sign for negative numbers. Floating-point numbers include a decimal point between digits. There is no distinction in syntax between integers and floats; the runtime handles numeric types appropriately.

Boolean literals are written as “true” or “false” without any enclosing symbols. They represent the two truth values and are commonly used for flags and conditions.

Array literals are enclosed in square brackets with elements separated by commas. The elements can be any valid expression, including other literals, variable references, or nested arrays. Array literals provide a convenient way to create collections inline.

Object literals are enclosed in curly braces with fields written as key-colon-value pairs separated by commas. The keys are identifiers; the values can be any valid expression. Object literals allow you to construct structured data inline, which is particularly useful for return values and event payloads.

```
<Create> the <user> with { name: "Alice", email: "alice@example.com", active: true }.
```

4.8 Where Clauses

The where clause allows you to filter or constrain operations. It appears after the object clause and begins with the keyword “where,” followed by a condition.

Where clauses are most commonly used with Retrieve actions to specify which records to fetch from a repository. When you write a where clause, you are expressing a constraint that the retrieved data must satisfy. The repository implementation uses this constraint to filter results, often translating it into a database query.

Conditions in where clauses can use equality checks with “is” or “=” and inequality checks with “!=”. They can use comparison operators for numeric values. They can combine multiple conditions with “and” and “or.” The expressive power is similar to the WHERE clause in SQL, which is intentional—many repositories are backed by databases, and the mapping should be straightforward.

Where clauses can also appear with Filter actions, where they specify which elements of a collection to include in the result. The semantics are the same: only elements satisfying the condition are included.

```
<Retrieve> the <order> from the <order-repository> where id = <order-id>.
```

4.9 When Conditions

The when condition allows you to make a statement conditional on some expression being true. It appears at the end of the statement, after any where clause, and begins with the keyword “when.”

Unlike traditional if-statements, when conditions do not create branches in control flow. A statement with a when condition either executes (if the condition is true) or is skipped (if the condition is false). There is no else clause, no alternative path. This design keeps the linear flow of ARO feature sets intact while allowing for conditional execution of individual statements.

When conditions are useful for optional operations—things that should happen only if certain prerequisites are met. For example, you might send a notification only when the user has opted into notifications, or log debug information only when debug mode is enabled.

The condition can be any boolean expression. You can reference bound variables, compare values, check for existence, and combine conditions with logical operators. The same expression syntax used elsewhere in ARO applies within when conditions.

```
<Send> the <notification> to the <user: email> when <user: notifications> is true.
```

4.10 Comments

Comments in ARO use Pascal-style syntax: an opening parenthesis followed by an asterisk, the comment text, an asterisk followed by a closing parenthesis. Comments can span multiple lines and can appear anywhere in the source where whitespace is allowed.

Comments are completely ignored by the parser. They exist solely for human readers, providing explanation, context, or temporary notes. Use comments to explain why something is done, not what is done. The code itself, with its natural-language-like structure, should explain what is happening.

4.11 Statement Termination

Every statement ends with a period. This is not optional; omitting the period is a syntax error. The period serves as an unambiguous statement terminator, making it clear where one statement ends and the next begins.

The period also reinforces the sentence metaphor. Just as English sentences end with periods, ARO statements end with periods. This small detail contributes to the natural-language feel of ARO code.

4.12 Putting It All Together

Having examined each component in isolation, let us see how they combine in complete statements of varying complexity.

A minimal statement has an action, an article, a result, a preposition, an article, and an object:

```
<Retrieve> the <users> from the <user-repository>.
```

Source: Examples/UserService/users.aro:7

A statement with a qualifier on the result and object adds more specificity:

```
<Extract> the <user-id: String> from the <pathParameters: id>.
```

A statement with a literal value provides data inline:

```
<Create> the <greeting> with "Hello, World!".
```

A statement with an expression computes a value:

```
<Compute> the <total> with <subtotal> + <tax>.
```

A statement with a where clause filters the operation:

```
<Retrieve> the <user> from the <user-repository> where <id> is <user-id>.
```

Source: Examples/UserService/users.aro:14

A statement with a when condition executes conditionally:

```
<Send> the <notification> to the <user> when <user: notifications> is true.
```

Each of these statements follows the same fundamental pattern while using optional elements to add precision and expressiveness. The pattern is the constant; the optional elements are the variables. Once you internalize the pattern, you can read and write any ARO statement fluently.

Next: Chapter 5 — Feature Sets

§ Chapter 5: Feature Sets

> “A feature set is not a function. It’s a response to the world.”

* * *

5.1 What Is a Feature Set?

A feature set is ARO’s fundamental unit of organization. It groups related statements that together accomplish a business goal. If statements are sentences, then a feature set is a paragraph—a coherent unit of meaning that expresses a complete thought about what should happen in response to some triggering condition.

The term “feature set” is deliberately chosen over alternatives like “function,” “method,” or “procedure” because it emphasizes the reactive nature of ARO code. A feature set does not run because you call it. It runs because something in the world triggered it. An HTTP request arrives. A file changes. A custom event is emitted. The application starts. These external stimuli activate feature sets, which then execute their statements in response.

This reactive model is fundamental to understanding ARO. In traditional programming, you write a main function that calls other functions in a sequence you control. In ARO, you write feature sets that respond to events, and the runtime orchestrates when they execute based on what happens. You describe what should occur when certain conditions arise; the runtime ensures your descriptions become reality when those conditions occur.

Every feature set has a two-part header enclosed in parentheses, followed by a body enclosed in curly braces. The header consists of a feature name and a business activity, separated by a colon. The body contains the statements that execute when the feature set is triggered. There is no other structure. No parameter lists, no return type declarations, no visibility modifiers. The simplicity is intentional.

* * *

5.2 The Header

The feature set header serves two purposes: it gives the feature set a unique identity, and it documents what business context the feature set belongs to.

The first part, before the colon, is the feature name. This name must be unique within the application. If two feature sets have the same name, the compiler reports an error. The name identifies this specific feature set for routing purposes—HTTP requests match against feature names that correspond to operation identifiers in the OpenAPI specification, custom events match against handler patterns that include the event name.

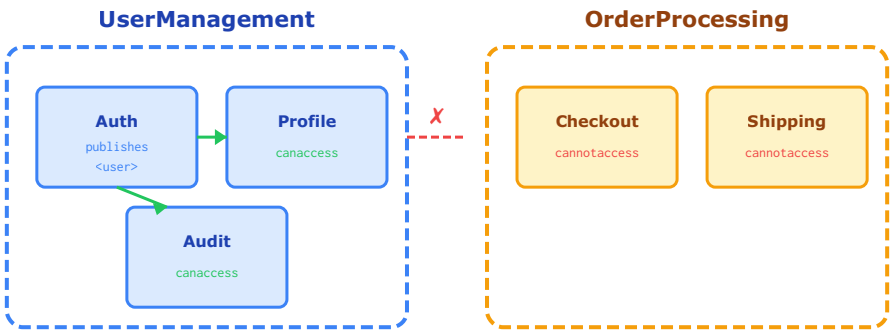
Names can take several forms. Simple identifiers like `listUsers` or `getOrder` are common for HTTP handlers because OpenAPI operation identifiers typically follow this style. Hyphenated names like `Application-Start` or `Handle-Error` read more like natural language and are common for lifecycle handlers. Descriptive phrases with spaces like `Send Welcome Email` work well for event handlers because they clearly describe the purpose.

The second part, after the colon, is the business activity. This describes the domain context or purpose of the feature set. For an HTTP API handling user operations, you might use `User API` as the business activity. For an event handler dealing with order processing, you might use `Order Processing` . For the application entry point, you might use the application name itself.

The business activity serves two purposes: documentation and scoping.

As documentation, it helps readers understand what larger goal this feature set contributes to. When you scan a file containing many feature sets, the business activities provide orientation. They answer the question: what is this code about?

As a scoping mechanism, the business activity controls variable visibility. When you publish a variable using the Publish action, that variable becomes accessible only to other feature sets with the same business activity. This enforces modularity and prevents unintended coupling between different business domains.



Publishedvariables are scoped to business activity

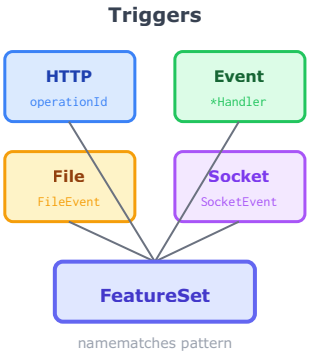
Consider an application with two business activities: `User Management` and `Order Processing`. If a feature set in `User Management` publishes a variable, only other feature sets with the same `User Management` activity can access it. Feature sets in `Order Processing` cannot see that variable, even if they use the same name. This boundary prevents unintended dependencies between unrelated domains.

Certain business activity patterns also have semantic significance. When the pattern ends with "Handler," the runtime treats the feature set as an event handler. The text before "Handler" specifies which event triggers it. This convention transforms what looks like documentation into configuration, allowing you to wire up event handling simply by naming things according to the pattern.

* * *

5.3 Triggering Patterns

Feature sets execute in response to events. The runtime maintains an event bus that routes events to matching feature sets based on their headers. The triggering mechanism matches feature set names to incoming events:



Trigger Type	Naming Pattern	Example
Lifecycle	`Application-Start` , `Application-End`	Entry point, shutdown
HTTP	OpenAPI `operationId`	`listUsers` , `createOrder`
Custom Event	`{EventName} Handler`	`UserCreated Handler`
File Event	`File Event Handler`	React to file changes
Socket Event	`Socket Event Handler`	React to socket messages

> **See Chapter 10** for application lifecycle details (startup, shutdown, Keepalive).

> **See Chapter 9** for event bus mechanics and handler patterns.

* * *

5.4 Structure and Execution

Within a feature set, statements execute in order from top to bottom. There is no branching, no looping, no early return. Execution begins with the first statement and proceeds through each subsequent statement until reaching the end or encountering an error.

This linearity might seem limiting, but it actually simplifies reasoning about code. When you look at a feature set, you know exactly what order things happen. There are no hidden control flows, no callbacks that might execute at unexpected times, no conditional branches that might skip important steps. What you see is what executes.

The `when` clause provides conditional execution without branching. A statement with a when clause executes only if the condition is true; otherwise, the statement is skipped and execution continues with the next statement. This is not a branch—there is no else path, no alternative action. Either the statement happens or it does not.

Each statement can bind a result to a name. That binding becomes available to all subsequent statements in the same feature set. If you create a value named `user` in the first statement, you can reference `user` in the second, third, and all following statements. This accumulation of bindings creates the context in which later statements operate.

Bindings are immutable within a feature set. Once you bind a name to a value, you cannot rebind it to a different value. If you try, the compiler reports an error. This constraint prevents a common class of bugs where a variable changes unexpectedly. It also pushes you toward descriptive names because you cannot reuse generic names like `temp` or `result`.

* * *

5.5 Scope and Visibility

Variables bound within a feature set are visible only within that feature set. A binding in one feature set does not affect bindings in another. Each feature set has its own isolated symbol table that begins empty and accumulates bindings as statements execute.

This isolation has important implications. If you create a value in one feature set and need to use it in another, you cannot simply reference it by name. The second feature set has no knowledge of what the first feature set bound. This prevents accidental coupling between feature sets that happen to use the same variable names.

When you need to share data between feature sets, you have two options. The first is through events: emit an event carrying the data, and have the receiving feature set extract what it needs from the event payload. This maintains loose coupling because the emitting feature set does not need to know which handlers will receive the event.

The second option is the Publish action, which makes a binding available to other feature sets within the same business activity. When you publish a value under an alias, that alias becomes accessible from any feature set with the same business activity that executes afterward. This scoping enforces modularity—different business domains cannot accidentally depend on each other's published variables. Use publishing for configuration data loaded at startup or for values that need to be shared within a domain, but use it sparingly because shared state complicates reasoning about program behavior.

The execution context provides access to information that is always available. For HTTP handlers, this includes request data: path parameters extracted from the URL, query parameters, headers, and the request body. For event handlers, this includes the event payload containing whatever data was emitted with the event. These context values are not

bound to names in advance; you extract them using the Extract action, which binds them to names you choose.

* * *

5.6 Naming Conventions

Good naming makes code readable. In ARO, feature set names serve double duty as identifiers and documentation, so choosing appropriate names is particularly important.

For HTTP handlers, names should match the operation identifiers in your OpenAPI specification exactly. This is not a convention but a requirement—the routing mechanism uses name matching to connect requests to handlers. Operation identifiers typically follow camelCase conventions: `listUsers`, `createOrder`, `getProductById`. Your feature set names should match.

For event handlers, names should be descriptive of the action being performed. The convention is a verb phrase describing the handler's purpose: `Send Welcome Email`, `Update Search Index`, `Notify Administrator`. The business activity specifies the triggering event by ending with "Handler" preceded by the event name: `UserCreated Handler`, `OrderPlaced Handler`.

For lifecycle handlers, use the reserved names exactly as specified. `Application-Start` with a business activity of your choice. `Application-End` with business activity `Success` for graceful shutdown. `Application-End` with business activity `Error` for error handling.

For internal feature sets that handle domain logic but are not directly triggered by external events, use names that describe the business operation. `Calculate Shipping`, `Validate Payment`, `Check Inventory`. These feature sets might be triggered by custom events emitted from other feature sets or might be called through other mechanisms.

* * *

5.7 File Organization

ARO applications are directories, and the runtime automatically discovers all files with the `.aro` extension in that directory. You do not need import statements or explicit file references. When you create a new file and add feature sets to it, those feature sets become part of the application immediately.

This automatic discovery encourages organizing feature sets into files by domain or purpose. A typical pattern separates lifecycle concerns from business logic: one file for application start and shutdown, other files for different domains of functionality. A user service might have a main file for lifecycle, a users file for user-related HTTP handlers, an orders file for order-related handlers, and an events file for event handlers.

The specific organization you choose matters less than consistency. Some teams prefer fine-grained files with only a few feature sets each. Others prefer coarser files that group all related functionality together. What matters is that team members can find what they are looking for and understand where new code should be added.

Because there are no imports, all feature sets are visible throughout the application. An event emitted in one file triggers handlers defined in any file. A variable published in one file is accessible from any feature set with the same business activity, regardless of which file it is defined in. This visibility is powerful but requires discipline. Establish conventions for how feature sets in different files should interact, and document those conventions so team members can follow them.

* * *

5.8 The Context Object

When a feature set executes, it has access to contextual information appropriate to how it was triggered. This information is available through special identifiers that you access using the Extract action.

HTTP handlers have access to request data. The `pathParameters` object contains values extracted from the URL path based on path templates in the OpenAPI specification. If the path template is `/users/{id}`, the `id` path parameter contains whatever value appeared in that position of the actual URL. The `queryParameters` object contains query string parameters. The `headers` object contains HTTP headers. The `request` object contains the full request, including the body.

Event handlers have access to the event that triggered them through the `event` identifier. The event object contains whatever data was included when the event was emitted. If a feature set emits a `UserCreated` event with user data, the handler can extract that user data from the event.

File event handlers have access to file system event details: the path of the file that changed, the type of change that occurred (created, modified, deleted), and other relevant metadata depending on the file system implementation.

You access context data using the Extract action with qualifiers. The expression `<pathParameters: id>` means "the id property of the pathParameters object." The expression `<event: user.email>` means "the email property of the user property of the event object." Qualifiers chain to allow navigation into nested structures.

* * *

5.9 From Here

Feature sets are the building blocks of ARO applications. They respond to events, execute statements, and either complete successfully or encounter errors. The runtime orchestrates their execution based on matching patterns between events and feature set headers.

The next chapter explores how data flows through feature sets and between them. Understanding data flow is essential for building applications that share information appropriately while maintaining the loose coupling that makes event-driven architectures powerful.

* * *

Next: Chapter 6 — Data Flow

§ Chapter 6: Data Flow

> "Data flows forward, never backward."

* * *

6.1 The Direction of Data

Every ARO statement moves data in a predictable direction. This is not merely a conceptual framework but a fundamental property of the language that the runtime enforces. Understanding data flow is essential for writing effective ARO code because it shapes how you think about structuring your feature sets.

The fundamental principle is that data enters from the outside world, transforms within the feature set, and exits to the outside world. External sources include HTTP requests, files, databases, and network connections. Internal transformations include creating new values, validating existing ones, computing results, and restructuring data. External destinations include responses to callers, persistent storage, emitted events, and outbound communications.

This directional flow creates a natural structure for feature sets. The beginning of a feature set typically contains actions that bring data in. The middle contains actions that transform that data. The end contains actions that send results out. While this is not enforced syntactically—you can technically write statements in any order—the data dependencies between statements create an inherent ordering that usually follows this pattern.

The flow is unidirectional within a single execution. Data moves forward through the statement sequence; it does not loop back. Each statement receives its inputs from the accumulated results of previous statements and produces outputs that become available to subsequent statements. This forward-only flow simplifies reasoning about program behavior because you can trace the origin of any value by looking backward through the statement sequence.

* * *

6.2 Semantic Roles

Every action in ARO has a semantic role that categorizes its data flow characteristics. The role is determined by the action's verb and affects how the runtime processes the action and validates its usage. There are four roles: REQUEST, OWN, RESPONSE, and EXPORT.

REQUEST actions bring data from outside the feature set into its local scope. When you extract data from a request, retrieve records from a repository, fetch content from a URL, or read a file, you are performing a REQUEST operation. The characteristic of REQUEST actions is that they produce new bindings by pulling data inward. After a REQUEST action executes, the symbol table contains a new entry that was not there before, and the value came from somewhere external.

OWN actions transform data that already exists within the feature set. When you create a new value from existing ones, compute a result from inputs, validate data against a schema, filter a collection, or merge objects, you are performing an OWN operation. The characteristic of OWN actions is that they read from existing bindings and produce new bindings, but all the data stays internal to the feature set. No external communication occurs.

RESPONSE actions send data out of the feature set to the caller and terminate normal execution. When you return a response to an HTTP client or throw an error, you are performing a RESPONSE operation. The characteristic of RESPONSE actions is that they move data outward and end the feature set's execution. After a RESPONSE action, no subsequent statements execute because control has returned to the caller.

EXPORT actions make data available beyond the current execution without terminating it. When you store data to a repository, emit an event for handlers to process, publish a value for other feature sets in the same business activity, log a message, or send a notification, you are performing an EXPORT operation. The characteristic of EXPORT actions is that they have side effects—they change something in the outside world—but execution continues with the next statement.

The runtime uses these roles for validation and optimization. It can detect when you try to use a value that has not been bound, when you attempt to read from a binding that a statement's role would not produce, or when you have dead code after a RESPONSE action. Understanding the roles helps you write code that the runtime can validate effectively.

* * *

6.3 The Symbol Table

As statements execute, ARO maintains a symbol table that maps variable names to values. This table is the mechanism by which data flows between statements. When a statement produces a result, the result is added to the symbol table under the specified name. When a subsequent statement references that name, the runtime looks up the value in the symbol table.

The symbol table starts empty at the beginning of each feature set execution. As each statement executes, any result it produces is added to the table. This accumulation creates the context in which later statements operate. A statement near the end of the feature set has access to all the bindings created by statements that came before it.

The symbol table is append-only within a single execution. You cannot rebind a name to a different value. If you try to create a binding for a name that already exists, the compiler or runtime reports an error. This immutability prevents a common class of bugs where variables change unexpectedly and makes it easier to reason about program behavior—when you see a reference to a name, you know it refers to exactly the value that was originally bound.

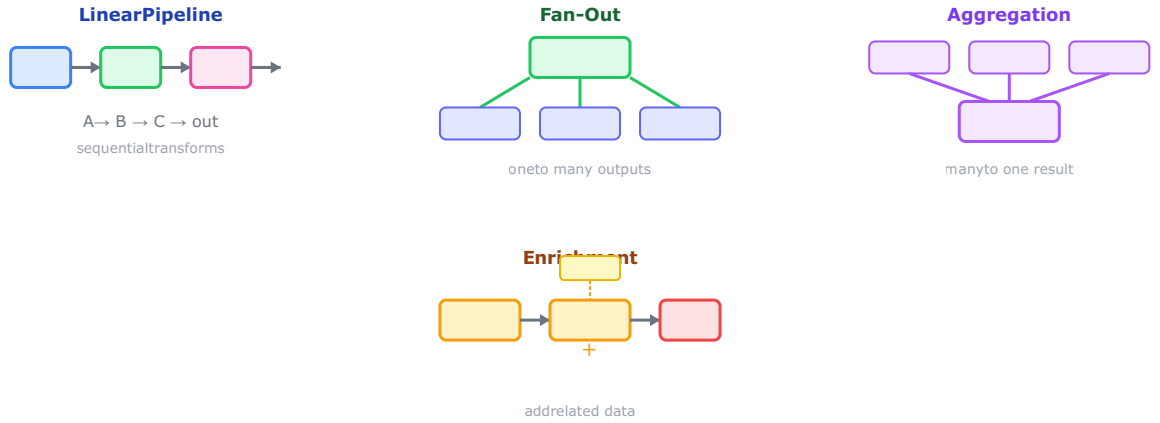
When a statement references a variable that does not exist in the symbol table, the runtime produces an error describing which variable is undefined and where the reference occurred. This is a runtime error rather than a compile-time error because the compiler cannot always determine whether a binding will exist. Some bindings depend on conditional execution or on values that only become known during execution.

The symbol table is scoped to the feature set. Bindings in one feature set do not affect bindings in another. Each feature set maintains its own independent table that exists only for the duration of that feature set's execution. This isolation prevents unintended interactions between feature sets and allows you to use descriptive names without worrying about conflicts.

* * *

6.4 Data Flow Patterns

Several common patterns emerge in how data flows through feature sets. Recognizing these patterns helps you structure your code effectively and understand existing code more quickly.



The linear pipeline is the most common pattern. Data enters at the beginning, flows through a series of transformations, and exits at the end. Each statement takes the output of the previous statement (or the original input) and produces something that the next statement needs. This creates a chain of dependencies that naturally organizes the code.

The fan-out pattern occurs when a single piece of data needs to trigger multiple independent operations. You might create a user and then want to store it, emit an event, log an audit message, and send a notification. Each of these operations uses the same user data but is otherwise independent. The statements appear in sequence but could conceptually execute in parallel because they do not depend on each other's results.

The aggregation pattern collects data from multiple sources and combines it into a single result. You might retrieve users from one repository, orders from another, and products from a third, then create a summary object that includes counts or statistics from all three. The gathering happens through multiple REQUEST actions, and the combination happens through an OWN action that references all the gathered bindings.

The enrichment pattern starts with a primary piece of data and augments it with related information from other sources. You might retrieve an order, then retrieve the customer associated with that order, then retrieve the items in that order, and finally assemble a detailed response that includes all of this related information. The key characteristic is that each subsequent retrieval depends on information from previous retrievals.

These patterns can combine in complex feature sets. A realistic API handler might aggregate data from multiple sources, enrich some of that data with additional lookups, fan out to multiple export operations, and finally return a response. Understanding the underlying patterns helps you navigate this complexity.

* * *

6.5 Cross-Feature Set Communication

Because each feature set has its own isolated symbol table, data does not automatically flow between feature sets. If one feature set creates a value and another feature set needs that value, you must explicitly communicate it through one of several mechanisms.

The Publish action makes a binding available to other feature sets within the same business activity. When you publish a value under an alias, that alias becomes accessible from any feature set with the same business activity that executes afterward. This scoping to business activity enforces modularity—feature sets in different business domains cannot accidentally depend on each other's published variables. Use publishing for configuration data, constants, or values that need to be shared within a domain, but prefer events for communication when the pattern fits.

Events provide a structured way to pass data between feature sets while maintaining loose coupling. When you emit an event with a payload, all handlers for that event type receive access to the payload. The emitting feature set does not need to know which handlers exist or what they will do with the data. The handlers extract what they need from the event payload and proceed independently. This decoupling allows you to add new behaviors by adding handlers without modifying the emitting code.

Repositories act as shared persistent storage. One feature set can store a value to a repository, and another feature set can retrieve it later. This communication is asynchronous in the sense that the retriever does not need to execute while the storer is executing. The repository holds the data between executions. This is appropriate for persistent data that outlives individual requests.

Repository names must end with `-repository`—this is not merely a convention but a requirement that enables the runtime to identify storage targets. When you write `<Store> the <user> into the <user-repository>`, the runtime recognizes `user-repository` as persistent storage because of its suffix. Names like `users` or `user-data` would not trigger repository semantics.

Repositories are scoped to business activities by default. A `user-repository` accessed by feature sets with the business activity "User Management" is separate from a `user-repository` accessed by feature sets with the business activity "Admin Tools". This scoping prevents unintended data sharing between different domains of your application.

Repositories store data as ordered lists. Each Store operation appends to the repository. A Retrieve operation returns all stored items unless you specify a filter with a where clause.

This list-based storage differs from key-value stores—you can have multiple items that match the same criteria, and you retrieve them all unless you filter.

The context object provides data that is available to handlers based on how they were triggered. HTTP handlers receive request data. Event handlers receive event payloads. This is not really communication between feature sets but rather communication from the triggering mechanism to the handler. The context is read-only; handlers cannot modify it to communicate back.

* * *

6.6 Qualified Access

When you reference a variable, you can use qualifiers to access nested properties within that variable's value. The qualifier path is written after the variable name, separated by colons. This allows you to navigate into structured data without creating intermediate bindings.

Accessing a property uses a single qualifier: referencing something like `user: name` accesses the name property of the user object. Accessing a deeply nested property chains qualifiers: referencing something like `order: customer.address.city` navigates three levels deep to get the city from the customer's address on the order.

Array indexing works similarly. You can access a specific element by index: referencing `items: 0` gets the most recently added element of the items array. Index 1 gets the second most recent, and so on. This reverse indexing matches common use cases where applications typically want to access recent data. You can combine array access with property access: referencing `items: 0: name` gets the name property of the most recent item.

Qualifiers work on the result of Extract actions, Create actions, and any other action that produces structured data. They also work on context objects like pathParameters, queryParameters, headers, and event payloads. This allows you to extract specific pieces of complex structures without binding the entire structure to an intermediate name.

The qualifier syntax reads naturally when the values have descriptive names. If you have a user with an address that has a city, then `user: address.city` reads almost like natural language describing what you want. This is another aspect of ARO's design philosophy of making code read like descriptions of intent rather than instructions to a computer.

* * *

6.7 Best Practices

Several practices help maintain clarity in data flow.

Keep the flow linear when possible. The most readable feature sets are those where data flows straight through from input to output with clear transformations along the way. When you find yourself with complex dependencies or multiple sources feeding into multiple outputs, consider whether the feature set is doing too much and might benefit from being split or from using events to separate concerns.

Name variables to reflect their state in the transformation pipeline. If you extract raw input, validate it, and then use it to create an entity, names like `raw-input`, `validated-data`, and `user` help readers understand what each value represents at that point in the flow. Names like `data1`, `data2`, and `data3` obscure this progression.

Minimize what you publish. Shared state creates dependencies between feature sets that can make code harder to understand and modify. Each published variable is a potential coupling point within its business activity. Prefer events for communication when the pattern fits, as they are more explicit about what is being communicated and to whom.

Use events for fan-out scenarios. When a single occurrence needs to trigger multiple independent actions, emitting an event and having multiple handlers is cleaner than listing all the actions inline. It also allows you to add new behaviors by adding handlers rather than modifying the original feature set.

Document complex data flows when the structure is not obvious from the code. A comment describing what data is available at each stage, or a diagram showing how data flows through the feature set, can help future readers understand non-trivial transformations.

* * *

Next: Chapter 7 — Computations

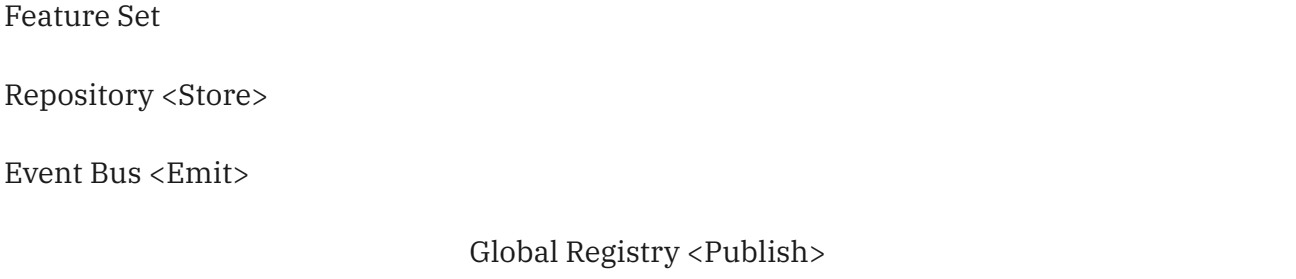
§ Chapter 6A: Export Actions

> “Know where your data goes.”

* * *

6A.1 Three Paths Out

When data leaves your feature set, it takes one of three paths. Each path serves a distinct purpose in the ARO architecture. Understanding which path to choose is essential for building well-structured applications.



Store sends data to a repository. The data persists for the application’s lifetime and can be retrieved later by any feature set. Use Store when you need to save data for future access.

Emit sends data to the event bus. Handlers that match the event type receive the data and execute independently. Use Emit when you want to trigger reactive behavior in other parts of the application.

Publish registers data in a global registry. The data becomes accessible to other feature sets but does not trigger any handlers. Use Publish when you need to share a value without triggering logic.

Each path has different characteristics that make it suitable for different scenarios. The following sections explore each path in detail.

* * *

6A.2 Store: Persistent Data

The Store action saves data to a named repository. Repositories are identified by names ending with `-repository`. This naming convention is not merely a style guide—the runtime uses this suffix to recognize storage targets.

When to Use Store

Store is the right choice when:

- You need to save data for later retrieval
- Multiple feature sets need to query the same data
- You want to maintain a collection of records
- You need audit logs or history

Basic Storage

The simplest form of Store appends a value to a repository:

```
(createUser: User API) {
    <Extract> the <user-data> from the <request: body>.
    <Create> the <user> with <user-data>.

    (* Store saves the user to the repository *)
    <Store> the <user> into the <user-repository>.

    <Return> a <Created: status> with <user>.
}
```

Each Store operation appends to the repository. If you store three users, the repository contains all three. Repositories are list-based, not key-value stores.

Retrieval

Data stored in a repository can be retrieved by any feature set:

```
(listUsers: User API) {
    (* Retrieve all users from the repository *)
    <Retrieve> the <users> from the <user-repository>.
    <Return> an <OK: status> with <users>.
}

(getUser: User API) {
    <Extract> the <id> from the <pathParameters: id>.

    (* Retrieve with a filter *)
    <Retrieve> the <user> from the <user-repository> where id = <id>.
    <Return> an <OK: status> with <user>.
}
```

The where clause filters the repository contents. Multiple conditions can be combined:

```
<Retrieve> the <orders> from the <order-repository>
    where status = "pending" and customer = <customer-id>.
```

Repository Observers

When data is stored, updated, or deleted in a repository, observers can react automatically. This is a powerful pattern for implementing side effects without coupling:

```
(* This observer runs automatically when user-repository changes *)
(Audit Changes: user-repository Observer) {
    <Extract> the <changeType> from the <event: changeType>.
    <Extract> the <entityId> from the <event: entityId>.

    <Log> the <audit: message> for the <console>
        with "[AUDIT] user-repository: " + <changeType> + " (id: " + <entityId> + ")".

    <Return> an <OK: status> for the <audit>.
}
```

The observer receives an event with details about the change:

Field	Description
`changeType`	“created”, “updated”, or “deleted”
`entityId`	ID of the affected entity (if the entity has an “id” field)
`newValue`	The new value (nil for deletes)
`oldValue`	The previous value (nil for creates)
`repositoryName`	Name of the repository

* * *

6A.3 Emit: Event-Driven Communication

The Emit action fires a domain event that triggers handlers in other feature sets. Unlike Store, the event payload is transient—it exists only during handler execution. Once all handlers complete, the event is gone.

When to Use Emit

Emit is the right choice when:

- You want to trigger side effects (notifications, analytics, etc.)
- You are building event-driven workflows or sagas
- You need loose coupling between feature sets
- Multiple handlers should react to the same occurrence

Basic Emission

Emit creates a domain event with a type and payload:

```
(createUser: User API) {
    <Extract> the <user-data> from the <request: body>.
    <Create> the <user> with <user-data>.
    <Store> the <user> into the <user-repository>.

    (* Emit notifies interested handlers *)
    <Emit> a <UserCreated: event> with <user>.

    <Return> a <Created: status> with <user>.
}
```

The event type is derived from the result descriptor. In this case, `UserCreated` becomes the event type, and handlers for "UserCreated Handler" are triggered.

Event Handlers

Handlers receive the event payload and can extract data from it:

```
(* Triggered by <Emit> a <UserCreated: event> with <user> *)
(Send Welcome Email: UserCreated Handler) {
    <Extract> the <user> from the <event: user>.
    <Extract> the <email> from the <user: email>.

    <Send> the <welcome-email> to the <email-service> with {
        to: <email>,
        subject: "Welcome!",
        template: "welcome"
    }.

    <Return> an <OK: status> for the <notification>.
}

(* Another handler for the same event *)
(Track Signup: UserCreated Handler) {
    <Extract> the <user> from the <event: user>.

    <Send> the <analytics-event> to the <analytics-service> with {
        event: "user_signup",
        properties: <user>
    }.

    <Return> an <OK: status> for the <tracking>.
}
```

Both handlers execute independently when a UserCreated event is emitted. The emitting code does not know or care which handlers exist.

Event Chains

Handlers can emit additional events, creating processing chains:

```
(* OrderPlaced triggers inventory reservation *)
(Reserve Inventory: OrderPlaced Handler) {
    <Extract> the <order> from the <event: order>.
    <Update> the <inventory> for the <order: items>.

    (* Continue the chain *)
    <Emit> an <InventoryReserved: event> with <order>.

    <Return> an <OK: status> for the <reservation>.
}

(* InventoryReserved triggers payment *)
(Process Payment: InventoryReserved Handler) {
    <Extract> the <order> from the <event: order>.
    <Send> the <charge> to the <payment-gateway> with <order>.

    (* Continue the chain *)
    <Emit> a <PaymentProcessed: event> with <order>.

    <Return> an <OK: status> for the <payment>.
}
```

This pattern enables complex workflows while keeping each handler focused on a single responsibility.

* * *

6A.4 Publish: Shared Values

The Publish action makes a value globally accessible without triggering any logic. Published values are available to other feature sets within the same business activity but do not cause handlers to execute.

When to Use Publish

Publish is rarely needed. Consider it only when:

- You load configuration at startup that multiple feature sets need
- You create a singleton resource (connection pool, service instance)

- You compute a value that multiple feature sets use without needing events

In most cases, Emit provides better decoupling. Use Publish only when you specifically need a shared value without reactive behavior.

Basic Publication

Publish registers a value under an alias:

```
(Application-Start: Config Loader) {
    <Read> the <config-data> from the <file: "./config.json">.
    <Parse> the <config: JSON> from the <config-data>.

    (* Make config available to all feature sets *)
    <Publish> as <app-config> <config>.

    <Return> an <OK: status> for the <startup>.
}
```

The alias (`app-config`) becomes the name other feature sets use to access the value.

Accessing Published Values

Published values can be referenced directly by their alias:

```
(getApiUrl: Configuration Handler) {
    (* app-config was published at startup *)
    <Extract> the <url> from the <app-config: apiUrl>.

    <Return> an <OK: status> with <url>.
}
```

Published values are scoped to the business activity. Feature sets in different business activities cannot access each other's published values.

6A.5 Decision Guide

Choosing between Store, Emit, and Publish becomes straightforward when you ask the right question.

What do you need?

Persist data for later retrieval? <Store>

Trigger handlers reactively? <Emit>

Share value without triggering logic? <Publish>

Comparison Table

Aspect	Store	Emit	Publish
Target	Repository	Event bus	Global registry
Triggers	Repository observers	Event handlers	Nothing
Data lifespan	Application lifetime	Handler execution only	Application lifetime
Access pattern	`<Retrieve>` with filters	`<Extract>` from event	Direct variable reference
Typical use	CRUD data, audit logs	Reactive workflows, notifications	Config, singletons
Coupling	Medium (via repository name)	Low (via event type)	Low (via alias name)

* * *

6A.6 Common Mistakes

Using Emit for Persistence

A common mistake is emitting an event when you need persistent data:

```
(* WRONG: Events are transient *)
(createMessage: Chat API) {
    <Extract> the <message-data> from the <request: body>.
    <Create> the <message> with <message-data>.

    (* This event disappears after handlers complete! *)
    <Emit> a <MessageCreated: event> with <message>.

    <Return> a <Created: status> with <message>.
}

(listMessages: Chat API) {
    (* ERROR: There's no way to retrieve emitted events *)
    (* The messages are gone! *)
}
```

The fix is to Store the data:

```
(* CORRECT: Store for persistence, Emit for notifications *)
(createMessage: Chat API) {
    <Extract> the <message-data> from the <request: body>.
    <Create> the <message> with <message-data>.

    (* Store the message for later retrieval *)
    <Store> the <message> into the <message-repository>.

    (* Also emit for any handlers that want to react *)
    <Emit> a <MessageCreated: event> with <message>.

    <Return> a <Created: status> with <message>.
}

(listMessages: Chat API) {
    (* Now we can retrieve stored messages *)
    <Retrieve> the <messages> from the <message-repository>.
    <Return> an <OK: status> with <messages>.
}
```

Using Store for Communication

Another mistake is storing data just to trigger an observer when Emit would be cleaner:

```
(* AWKWARD: Using store just to trigger behavior *)
(processPayment: Payment API) {
    <Extract> the <payment> from the <request: body>.

    (* Storing just to trigger the observer *)
    <Store> the <payment> into the <payment-notification-repository>.

    <Return> an <OK: status> with <payment>.
}

(Send Receipt: payment-notification-repository Observer) {
    (* This works but is awkward *)
    <Extract> the <payment> from the <event: newValue>.
    <Send> the <receipt> to the <email-service>.
}
```

The fix is to use Emit for reactive behavior:

```
(* BETTER: Use Emit for reactive communication *)
(processPayment: Payment API) {
    <Extract> the <payment> from the <request: body>.

    (* Emit for handlers that need to react *)
    <Emit> a <PaymentProcessed: event> with <payment>.

    <Return> an <OK: status> with <payment>.
}

(Send Receipt: PaymentProcessed Handler) {
    <Extract> the <payment> from the <event: payment>.
    <Send> the <receipt> to the <email-service>.
}
```

Use Store when you need persistence. Use Emit when you need reactive behavior. Use both when you need both.

Overusing Publish

Publish is sometimes overused when Emit would provide better decoupling:

```
(* QUESTIONABLE: Publishing user for other feature sets *)
(createUser: User API) {
    <Create> the <user> with <user-data>.
    <Store> the <user> into the <user-repository>.

    (* Publishing forces other feature sets to poll for this value *)
    <Publish> as <latest-user> <user>.

    <Return> a <Created: status> with <user>.
}
```

The problem is that other feature sets must know to check for the published value. With Emit, handlers are triggered automatically:

```
(* BETTER: Emit notifies interested parties automatically *)
(createUser: User API) {
    <Create> the <user> with <user-data>.
    <Store> the <user> into the <user-repository>.

    (* Handlers are triggered automatically *)
    <Emit> a <UserCreated: event> with <user>.

    <Return> a <Created: status> with <user>.
}
```

Reserve Publish for truly shared values like configuration, not for communication between feature sets.

6A.7 Complete Example: User Registration

Here is a realistic example that uses all three export actions appropriately. The scenario is user registration with email verification, welcome notifications, and analytics tracking.

main.aro — Application startup:

```
(Application-Start: User Service) {
    <Read> the <config-data> from the <file: "./config.json">.
    <Parse> the <config: JSON> from the <config-data>.

    (* Publish config for all feature sets *)
    <Publish> as <app-config> <config>.

    <Start> the <http-server> with <contract>.
    <Keepalive> the <application> for the <events>.
    <Return> an <OK: status> for the <startup>.
}
```

users.aro — User creation:

```
(createUser: User API) {
    <Extract> the <user-data> from the <request: body>.
    <Create> the <user> with <user-data>.

    (* Store for persistence — we need to retrieve users later *)
    <Store> the <user> into the <user-repository>.

    (* Emit for reactive behavior — handlers will send emails, track analytics *)
    <Emit> a <UserCreated: event> with <user>.

    <Return> a <Created: status> with <user>.
}

(getUser: User API) {
    <Extract> the <id> from the <pathParameters: id>.

    (* Retrieve stored user *)
    <Retrieve> the <user> from the <user-repository> where id = <id>.

    <Return> an <OK: status> with <user>.
}

(listUsers: User API) {
    (* Retrieve all stored users *)
    <Retrieve> the <users> from the <user-repository>.

    <Return> an <OK: status> with <users>.
}
```

handlers.aro — Event handlers:

```
(* Send welcome email when user is created *)
(Send Welcome Email: UserCreated Handler) {
    <Extract> the <user> from the <event: user>.
    <Extract> the <email> from the <user: email>.

    (* Access published config *)
    <Extract> the <from-address> from the <app-config: email.fromAddress>.

    <Send> the <welcome-email> to the <email-service> with {
        to: <email>,
        from: <from-address>,
        subject: "Welcome to our service!",
        template: "welcome"
    }.

    <Return> an <OK: status> for the <notification>.
}

(* Track signup in analytics *)
(Track Signup: UserCreated Handler) {
    <Extract> the <user> from the <event: user>.

    (* Access published config *)
    <Extract> the <analytics-key> from the <app-config: analytics.apiKey>.

    <Send> the <analytics-event> to the <analytics-service> with {
        apiKey: <analytics-key>,
        event: "user_signup",
        properties: {
            userId: <user: id>,
            email: <user: email>,
            createdAt: <user: createdAt>
        }
    }.

    <Return> an <OK: status> for the <tracking>.
}
```

observers.aro — Repository observers:

```
(* Audit all changes to user repository *)
(Audit User Changes: user-repository Observer) {
    <Extract> the <changeType> from the <event: changeType>.
    <Extract> the <entityId> from the <event: entityId>.
    <Extract> the <timestamp> from the <event: timestamp>.

    <Log> the <audit: message> for the <console>
        with "[AUDIT] " + <timestamp> + " user-repository: " + <changeType> + " (id: " + <entityId

    <Return> an <OK: status> for the <audit>.
}
```

This example demonstrates:

- **Publish** for shared configuration loaded at startup
- **Store** for user data that needs to be retrieved later
- **Emit** for triggering welcome emails and analytics
- **Repository observers** for audit logging

Each action serves its intended purpose, creating a well-structured, maintainable application.

Next: Chapter 7 — Computations

§ Chapter 7: Computations

> "Transform what you have into what you need."

* * *

7.1 The OWN Role

Among the four semantic roles in ARO—REQUEST, OWN, RESPONSE, and EXPORT—the OWN role occupies a special place. While REQUEST brings data in and RESPONSE sends it out, OWN is where the actual work happens. OWN actions transform data that already exists within the feature set, producing new values without external side effects.

The Compute action is the quintessential OWN action. It takes existing bindings, applies operations to them, and produces new bindings. No network calls, no file access, no external dependencies—just pure transformation of data that is already present. This purity makes OWN actions predictable and easy to reason about.

Consider what happens when you compute a string's length. The input string exists in the symbol table, bound to some variable name. The Compute action reads that value, performs a calculation (counting characters), and binds the result to a new variable name. The original string remains unchanged. The new binding appears in the symbol table. The flow continues.

This pattern—read, transform, bind—is the heartbeat of data processing in ARO.

* * *

7.2 Built-in Computations

ARO provides several built-in computations that cover common transformation needs:

The **length** and **count** operations count elements—characters in strings, items in arrays, keys in dictionaries. While interchangeable for most purposes, `length` is typically used for strings and `count` for collections. The **uppercase** and **lowercase** operations transform case. The **hash** operation produces an integer hash value useful for comparisons.

The **identity** operation, while seemingly trivial, serves an important purpose: it allows arithmetic expressions to be written naturally:

```
<Compute> the <total> from <price> * <quantity>.
```

Here, the expression `<price> * <quantity>` is the actual computation. The result binds to `total`. This is identity in action—the expression's result passes through unchanged.

* * *

7.3 Naming Your Results

A subtle problem arises when you need multiple results of the same operation. Consider computing the lengths of two different messages:

```
<Compute> the <length> from the <greeting>.
<Compute> the <length> from the <farewell>.  (* Overwrites! *)
```

Since ARO variables are immutable within their scope, the second statement overwrites the first. Both computations produce lengths, but only the last one survives.

The solution is the **qualifier-as-name** syntax. In this pattern, the qualifier specifies the operation while the base becomes the variable name:

```
<Compute> the <greeting-length: length> from the <greeting>.
<Compute> the <farewell-length: length> from the <farewell>.
```

Now both lengths exist with distinct names. You can compare them:

```
<Compare> the <greeting-length> against the <farewell-length>.
```

This syntax separates two concerns that were previously conflated: - The **base** (`greeting-length`) is what you want to call the result - The **qualifier** (`length`) is what operation to perform

The same pattern works with all computed operations:

```
<Compute> the <name-upper: uppercase> from the <name>.
<Compute> the <name-lower: lowercase> from the <name>.
<Compute> the <password-hash: hash> from the <password>.
```

For backward compatibility, the original syntax still works. Writing `<Compute> the <length> from <msg>.` recognizes `length` as both the variable name and the operation.

* * *

7.4 Extending Computations

The built-in operations cover common cases, but real applications often need domain-specific computations. ARO's plugin system allows you to add custom operations that integrate seamlessly with the Compute action.

Plugins implement the `ComputationService` protocol:

```
public protocol ComputationService: Sendable {
    func compute(named: String, input: Any) async throws -> any Sendable
}
```

When the Compute action executes, it first checks for a registered computation service. If one exists, it delegates to the plugin. This allows plugins to provide operations like cryptographic hashes, custom string transformations, or domain-specific calculations.

A cryptography plugin might provide:

```
<Compute> the <password-hash: sha256> from the <password>.
<Compute> the <signature: hmac> from the <message>.
```

A formatting plugin might provide:

```
<Compute> the <formatted-date: iso8601> from the <timestamp>.
<Compute> the <money-display: currency> from the <amount>.
```

The syntax remains consistent regardless of whether the operation is built-in or plugin-provided. This uniformity means you can start with built-in operations and add plugins later without changing how your code reads.

See Chapter 17 for the full plugin development guide.

7.5 Computation Patterns

Several patterns emerge in how computations are used within feature sets.

Derived values compute new data from existing bindings:

```
<Extract> the <price> from the <product: price>.
<Extract> the <quantity> from the <order: quantity>.
<Compute> the <subtotal> from <price> * <quantity>.
<Compute> the <tax> from <subtotal> * 0.08.
<Compute> the <total> from <subtotal> + <tax>.
```

Normalization ensures consistent data formats:

```
<Extract> the <email> from the <input: email>.
<Compute> the <normalized-email: lowercase> from the <email>.
```

Chained transformations build complex results step by step:

```
<Compute> the <base> from <quantity> * <unit-price>.
<Compute> the <discounted> from <base> * (1 - <discount-rate>).
<Compute> the <with-tax> from <discounted> * (1 + <tax-rate>).
```

Aggregation combines collection data:

```
<Retrieve> the <orders> from the <order-repository>.
<Compute> the <order-count: count> from the <orders>.
```

Each pattern follows the read-transform-bind rhythm. Data flows forward, transformations produce new values, and the symbol table accumulates bindings that subsequent statements can use.

* * *

Next: Chapter 8 — Understanding Qualifiers

§ Chapter 8: Understanding Qualifiers

> "The colon is context-aware: it navigates data or selects operations."

* * *

8.1 The Two Roles of Qualifiers

Throughout ARO, you encounter the colon syntax `<name: qualifier>`. This compact notation carries significant meaning, but that meaning depends on context. Understanding when the colon navigates data versus when it selects operations is essential for writing clear ARO code.

The qualifier serves two distinct purposes:

Context	Role	Example
Operations (Compute, Transform, Sort)	Selects which operation to perform	`<len: length>`
Data Access (Extract, Objects)	Navigates to nested properties	`<event: user>`

This distinction matters because the same syntax produces fundamentally different results depending on the action being performed.

* * *

8.2 Operation Qualifiers

When using actions like Compute, Validate, Transform, or Sort, the qualifier specifies which operation to apply. The base identifier becomes the variable name for the result.

The Problem: Name Collisions

Consider computing lengths of multiple strings:

```
<Compute> the <length> from the <greeting>.
<Compute> the <length> from the <farewell>.
```

Both statements attempt to bind to `length`. Since ARO variables are immutable within a scope, the second overwrites the first. You lose the greeting's length.

The Solution: Qualifier-as-Name

Separate the variable name from the operation:

```
<Compute> the <greeting-length: length> from the <greeting>.
<Compute> the <farewell-length: length> from the <farewell>.
```

Now `greeting-length` holds 12 and `farewell-length` holds 8. Both values exist simultaneously, ready for comparison:

```
<Compare> the <greeting-length> against the <farewell-length>.
```

Available Operations by Action

Action	Operations
Compute	`length`, `count`, `hash`, `uppercase`, `lowercase`, `identity`
Validate	`required`, `exists`, `nonempty`, `email`, `numeric`
Transform	`string`, `int`, `integer`, `double`, `float`, `bool`, `boolean`, `json`, `identity`
Sort	`ascending`, `descending`

Examples

```
(* Multiple computations with distinct names *)
<Compute> the <name-upper: uppercase> from the <name>.
<Compute> the <name-lower: lowercase> from the <name>.
<Compute> the <name-len: length> from the <name>.

(* Validation with named results *)
<Validate> the <email-valid: email> for the <input-email>.
<Validate> the <age-valid: numeric> for the <input-age>.

(* Transformations *)
<Transform> the <user-json: json> from the <user>.
<Transform> the <count-str: string> from the <count>.
```

* * *

8.3 Field Navigation Qualifiers

When accessing data from objects, events, or requests, the qualifier navigates to nested properties. The qualifier acts as a path into the data structure.

Single-Level Access

```
(* Extract user from event payload *)
<Extract> the <user> from the <event: user>.

(* Extract body from request *)
<Extract> the <data> from the <request: body>.

(* Extract id from path parameters *)
<Extract> the <user-id> from the <pathParameters: id>.
```

Deep Navigation

For nested structures, use dot-separated paths:

```
(* Access deeply nested data *)
<Extract> the <city> from the <user: address.city>.
<Extract> the <zip> from the <user: address.postal-code>.

(* Navigate through arrays and objects *)
<Extract> the <first-name> from the <response: data.users.0.name>.
```

Common Patterns

```
(* HTTP request handling *)
<Extract> the <auth-token> from the <request: headers.Authorization>.
<Extract> the <content-type> from the <request: headers.Content-Type>.

(* Event handling *)
<Extract> the <order> from the <event: payload.order>.
<Extract> the <customer-id> from the <event: payload.order.customer-id>.

(* Configuration access *)
<Extract> the <timeout> from the <config: server.timeout>.
<Extract> the <max-retries> from the <config: server.retry.max-attempts>.
```

* * *

8.4 Type Annotations with `as`

For data pipeline operations (Filter, Reduce, Map), you can optionally specify result types using the `as` keyword:

```
(* Without type - inferred automatically *)
<Filter> the <active-users> from the <users> where <active> is true.

(* With explicit type using 'as' *)
<Filter> the <active-users> as List<User> from the <users> where <active> is true.

(* Reduce with type for precision *)
<Reduce> the <total> as Float from the <orders> with sum(<amount>).
```

Type annotations are **optional** because ARO infers result types from the operation. Use explicit types when:

1. You need a specific numeric precision (Float vs Integer)
2. You want documentation in the code

3. You're overriding default inference

See ARO-0038 for the full specification.

8.5 The Ambiguity Case

A natural question arises: what happens when data contains a field named like an operation?

```
<Create> the <data> with { length: 42, items: [1, 2, 3] }.
```

If you write:

```
<Compute> the <len: length> from the <data>.
```

What does ARO compute? The `length` field (42) or the length of `data` (2 keys)?

The rule: Operation qualifiers apply to the action's semantic meaning, not field access. The Compute action computes `length` (the operation) on `data`, returning 2 (two keys in the dictionary).

To access the `length` field, use Extract:

```
<Extract> the <len> from the <data: length>.
```

This returns 42—the value of the `length` field.

Summary of Resolution

Statement	Interpretation	Result
`<Compute> the <len: length> from <data>.`	Compute length of data	2
`<Extract> the <len> from <data: length>.`	Extract length field	42

* * *

8.6 Best Practices

Use descriptive base names with operation qualifiers:

```
(* Good: Clear what each variable holds *)
<Compute> the <greeting-length: length> from the <greeting>.
<Compute> the <password-hash: hash> from the <password>.

(* Avoid: Unclear what 'len' or 'h' represent *)
<Compute> the <len: length> from the <greeting>.
<Compute> the <h: hash> from the <password>.
```

Keep object structures shallow when possible:

Deeply nested paths become hard to read and maintain. Consider flattening data structures or extracting intermediate values:

```
(* Hard to read *)
<Extract> the <name> from the <response: data.results.0.user.profile.name>.

(* Clearer with intermediate steps *)
<Extract> the <user> from the <response: data.results.0.user>.
<Extract> the <profile> from the <user: profile>.
<Extract> the <name> from the <profile: name>.
```

When ambiguity exists, prefer explicit actions:

If a field name collides with an operation name, use the appropriate action explicitly rather than relying on context resolution:

```
(* Explicit about intent *)
<Extract> the <len-value> from the <obj: length>.     (* Get the field *)
<Compute> the <obj-size: count> from the <obj>.       (* Count the keys *)
```

In summary, qualifiers serve two purposes: navigating data structures and selecting operations.

```
<Extract> the <city> from the <user: address.city>.
<Compute> the <name-upper: uppercase> from the <name>.
```

* * *

Next: Chapter 9 — The Happy Path

§ Chapter 9: The Happy Path

> “Optimism is a strategy, not naivety.”

* * *

7.1 The Philosophy

ARO code expresses only the happy path—what should happen when everything works correctly. There is no try/catch mechanism, no error handling blocks, no defensive programming patterns. You write the successful case, and the runtime handles everything else.

This is not an oversight or a limitation. It is a deliberate design choice based on the observation that error handling code often obscures the business logic it surrounds. When you read traditional code, you spend significant mental effort distinguishing between the actual work and the error handling scaffolding. The happy path philosophy eliminates this noise by moving error handling entirely to the runtime.

The approach works because the structure of ARO statements provides enough information to generate meaningful error messages automatically. Every statement expresses what it intends to accomplish in business terms: extracting a user identifier from path parameters, retrieving a user from a repository, returning an OK status with data. When any of these operations fails, the runtime constructs an error message from the statement itself, describing what could not be done using the same business language.

This means that error handling is not absent—it is automated. The runtime catches every failure, logs it with appropriate context, and returns or propagates an appropriate error response. You get consistent error handling across your entire application without writing any error handling code.

* * *

7.2 How Errors Work

When an action cannot complete successfully, ARO generates an error message derived from the failed statement. The message describes the operation in business terms rather than technical ones. If a statement attempts to retrieve a user from a repository and no matching user exists, the error message says exactly that: “Cannot retrieve the user from the user-repository where id = 42.”

For HTTP handlers, the runtime translates these errors into appropriate HTTP responses. A failed retrieval becomes a 404 response. A failed validation becomes a 400 response. A permission denial becomes a 403 response. The runtime infers the appropriate status code from the error context, and the response body contains a structured representation of the error including the descriptive message.

The key insight is that the error message describes what the code was trying to accomplish, not what went wrong internally. This makes error messages immediately understandable to anyone who knows the business domain. A message like “Cannot retrieve the user from the user-repository where id = 42” tells you exactly what the application was trying to do. You do not need to understand implementation details to understand the error.

The runtime also logs every error with full context: which feature set was executing, which statement failed, what values were involved, the timestamp, and a request identifier for correlation. This logging happens automatically for every failure, ensuring that debugging information is always available when you need it.

* * *

7.3 Error Message Generation

ARO constructs error messages systematically from the components of the failed statement. The message typically follows the pattern “Cannot [action] the [result] [preposition] the [object]” with any where clause or other qualifiers appended.

The action verb provides the main operation that failed. The result name identifies what was being produced or acted upon. The preposition clarifies the relationship between the action and its object. The object identifies the source, destination, or context of the operation. If the statement includes a where clause for filtering, that clause appears in the error message to help identify which specific item could not be found.

This systematic construction means that choosing good names in your code automatically produces good error messages. If you name your result “user-profile” and your repository “user-repository,” the error message will say “Cannot retrieve the user-profile from the user-repository,” which is immediately comprehensible. If you use names like “x” and “data,” the error message becomes “Cannot retrieve the x from the data,” which tells you nothing useful.

This connection between code quality and error message quality creates a positive incentive. Writing readable code with descriptive names not only helps human readers understand the code but also produces better error messages that help during debugging and troubleshooting.

Error Message Examples

Here is ARO code followed by the runtime error messages it produces when operations fail:

Code:

```
(getUser: User API) {
    <Extract> the <id> from the <pathParameters: id>.
    <Retrieve> the <user> from the <user-repository> where id = <id>.
    <Return> an <OK: status> with <user>.
}
```

When user ID 530 does not exist:

```
Runtime Error: Cannot retrieve the user from the user-repository where id = 530
  Feature: getUser
  Statement: <Retrieve> the <user> from the <user-repository> where id = <id>
```

When pathParameters does not contain id:

```
Runtime Error: Cannot extract the id from the pathParameters
  Feature: getUser
  Statement: <Extract> the <id> from the <pathParameters: id>
  Cause: Key 'id' not found in pathParameters
```

Another example with validation:

```
(createOrder: Order API) {
    <Extract> the <data> from the <request: body>.
    <Validate> the <data> against the <order-schema>.
    <Store> the <order> in the <order-repository>.
    <Return> a <Created: status> with <order>.
}
```

When validation fails:

```
Runtime Error: Cannot validate the data against the order-schema
  Feature: createOrder
  Statement: <Validate> the <data> against the <order-schema>
  Cause: Validation failed
```

When the repository is unavailable:

```
Runtime Error: Cannot store the order in the order-repository
  Feature: createOrder
  Statement: <Store> the <order> in the <order-repository>
  Cause: Connection refused
```

The pattern is consistent: the statement's natural language structure becomes the error message, with context about which feature failed and what caused the failure.

* * *

7.4 Why This Works

The happy path philosophy reduces code dramatically. Traditional error handling often quadruples the size of a function. Every operation that might fail needs a corresponding error check. Every error check needs logging, response construction, and potentially cleanup. A simple three-step operation can become twenty or thirty lines when fully protected with error handling.

ARO's approach eliminates all of this boilerplate. A three-step operation remains three statements. The code expresses exactly what it accomplishes with no noise from error handling. Readers can understand the business logic at a glance because there is nothing else to distract them.

The approach also produces consistent error responses across the entire application. In traditional codebases, different developers write error messages differently. One might say “User not found,” another “user_not_found,” another “404.” Status codes vary for similar errors. Some errors include stack traces, others do not. This inconsistency makes APIs harder to use and debug.

With ARO, every error follows the same format because every error is generated by the same mechanism. Clients can rely on consistent response structures. Operators can rely on consistent log formats. The application behaves predictably in all error scenarios because the runtime, not the developer, determines how errors are expressed.

Automatic logging eliminates another common failure mode. In traditional code, developers sometimes forget to log errors, especially in rarely-executed code paths. ARO logs every failure automatically with full context, ensuring that debugging information exists when you need it.

* * *

7.5 When Happy Path Hurts

The happy path philosophy is not universally applicable. Several categories of problems fit poorly with this approach.

Custom error messages are difficult when validation rules require specific feedback. If a password must contain at least eight characters, a number, and a special character, and the user’s password fails all three requirements, the generic “Cannot validate the password against the password-rules” message does not tell the user how to fix the problem. Custom actions can provide more detailed error messages, but this pushes complexity out of ARO and into Swift.

Conditional error handling becomes awkward when different failures require different responses. If duplicate email and invalid email format both fail validation but require different instructions to the user, ARO cannot distinguish between them at the language level. The runtime treats all validation failures identically.

Retry logic has no direct expression in ARO because there are no loops. If an external service is temporarily unavailable and the operation should be retried with exponential backoff, that logic must be implemented in a custom action. The ARO statement sees only success or failure; it cannot express “try again.”

Partial failures present a fundamental mismatch with the happy path model. If you need to send an email, update analytics, and sync with a CRM, and you want to continue even if some of these fail, ARO's approach of stopping on first failure does not work. All-or-nothing semantics are built into the model.

Graceful degradation, where an application continues with reduced functionality when some components fail, similarly does not fit. ARO assumes that every statement is required for successful completion. There is no way to mark some statements as optional or to provide fallback behavior when they fail.

* * *

7.6 Strategies for Complex Error Handling

When you need error handling that goes beyond the automatic behavior, several strategies can help.

Custom actions are the primary escape hatch. By implementing an action in Swift, you gain full access to traditional error handling patterns. You can provide detailed validation error messages, implement retry logic, handle multiple error conditions differently, and perform any other error-related processing. The ARO statement that invokes your action sees only the final result—success with a value, or failure with an error that propagates according to normal rules.

Event-based error handling allows recovery code to execute in a separate feature set. When certain errors occur, the runtime can emit events that trigger handlers. These handlers can log additional context, notify administrators, attempt compensating actions, or perform other recovery activities. The original feature set still fails, but the handlers provide an opportunity for side effects related to that failure.

The when clause provides conditional execution that can skip non-critical operations. A statement with a when clause that evaluates to false is simply not executed rather than causing an error. This is useful for optional notifications, analytics updates, or other operations that should not block the main flow if preconditions are not met.

Separating concerns into multiple feature sets can isolate failures. If you have operations that should succeed or fail independently, triggering them via events rather than executing them inline means each can complete or fail without affecting the others. Event handlers execute in their own isolated contexts.

* * *

7.7 The Happy Path Contract

When you write ARO code, you implicitly accept a contract about how error handling works. You express what should happen when inputs are valid and systems are working. The runtime handles what happens when things fail. Custom actions provide escape hatches for scenarios that require more sophisticated error handling.

This contract works well for CRUD operations, data pipelines, event handling, and straightforward API endpoints—scenarios where the success path is clear and the failure mode is essentially “it didn’t work.” The automatic error messages are usually sufficient because there is typically only one kind of failure: the operation could not be completed.

The contract works less well for distributed transactions where you need coordination across multiple systems, retry-heavy workflows where transient failures should be retried automatically, complex validation with many specific error cases that each require different user guidance, and real-time systems that must degrade gracefully rather than failing entirely.

Knowing when to use ARO and when to use custom actions or other approaches is part of becoming proficient with the language. The happy path philosophy is a powerful default that eliminates enormous amounts of boilerplate for common scenarios. Recognizing the scenarios where it does not fit is equally important.

* * *

7.8 Best Practices

Trust the runtime’s error handling. Do not add defensive checks for conditions that the runtime already handles. If the runtime will produce an appropriate error when a required value is missing, there is no need to add a validation statement that checks for its presence. The redundant check adds code without adding value.

Use descriptive names because they become part of error messages. The quality of your variable and repository names directly affects the quality of automatically generated error messages. Names like “user-profile,” “order-repository,” and “email-address” produce clear, understandable error messages. Names like “x,” “data,” and “repo” produce error messages that tell you nothing useful.

Keep feature sets focused so that errors have clear context. When a feature set performs a single coherent operation, an error in that feature set has obvious meaning. When a feature set performs many unrelated operations, errors become harder to interpret because the context is muddied.

Design your domain model so that validation can be expressed through structure rather than explicit checks. If invalid data cannot be created in the first place because the types enforce validity, you eliminate entire categories of validation errors from your code.

* * *

Next: Chapter 10 — The Event Bus

§ Chapter 10: The Event Bus

> “In an event-driven system, everything is a reaction.”

* * *

9.1 Events, Not Calls

ARO is fundamentally event-driven. Feature sets do not call each other directly—they react to events. This is a crucial architectural distinction that shapes how you design and reason about ARO applications.

In traditional programming, you invoke functions by name. The caller knows about the callee and depends on it directly. If the callee changes its signature or behavior, the caller must be updated. This coupling creates a web of dependencies that grows denser as applications expand.

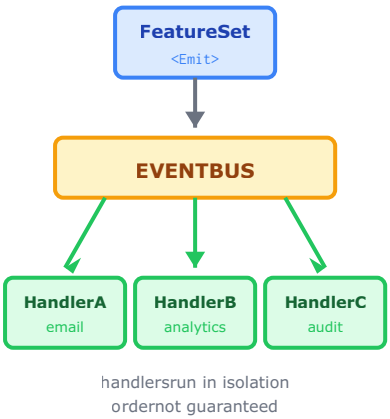
ARO takes a different approach. When a feature set needs something to happen afterward, it emits an event describing what occurred. Other feature sets can register as handlers for that event type. The emitting feature set does not know which handlers exist or what they will do. It simply publishes information about what happened and moves on.

This decoupling has profound implications. You can add new behaviors by adding handlers without modifying existing code. You can remove behaviors by removing handlers without affecting the emitting code. Multiple independent subsystems can react to the same event without coordination. The architecture remains loose and extensible.

The event bus is the runtime component that makes this work. It receives events from Emit actions, matches them to handlers based on naming patterns, and dispatches them for execution. The bus is invisible to your code—you simply emit events and register handlers, and the runtime handles the routing.

* * *

9.2 How the Event Bus Works



The event bus maintains a registry of handlers organized by event type. When the application starts, the runtime scans all feature sets, identifies those whose business activity matches the handler pattern, and registers them with the bus. This registration happens automatically based on naming conventions.

When a feature set executes an Emit action, the runtime creates an event object containing the event name and payload data. This object is delivered to the event bus, which looks up all handlers registered for that event type. Each matching handler receives the event and executes independently.

The bus provides delivery guarantees within a single application instance. When you emit an event, all registered handlers will eventually execute. However, the order of execution is not guaranteed—handlers may run in any sequence. If you need guaranteed ordering, you must express it through event chaining where each handler emits an event that triggers the next step.

Handler execution is isolated. Each handler runs in its own context with its own symbol table. A failure in one handler does not affect other handlers for the same event. The emitting feature set is also isolated—it completes regardless of whether handlers succeed or fail. This fire-and-forget semantics makes event emission a non-blocking operation that does not wait for handlers to complete.

* * *

9.3 Event Matching

Handlers are matched to events based on their business activity. The standard pattern is that the business activity ends with “Handler” preceded by the event name. When a feature set declares its business activity as “UserCreated Handler,” it becomes a handler for events named “UserCreated.”

This naming-based matching is simple and transparent. By reading the feature set declaration, you know exactly which events it handles. By searching the codebase for “UserCreated Handler,” you find all handlers for that event type. No configuration files or external registrations are needed—the code itself declares its event relationships.

The runtime supports several built-in event types generated by services rather than user code. File system services emit File Event when files change. Socket services emit Socket Event when messages arrive. Timer services emit Timer Event on schedule. HTTP services generate events that trigger feature sets matching OpenAPI operation identifiers. These built-in events follow the same matching rules as custom events.

You can have multiple handlers for the same event. When UserCreated is emitted, every feature set with “UserCreated Handler” in its business activity will execute. This allows you to add behaviors incrementally. One handler might send a welcome email. Another might update analytics. A third might create an audit record. Each handler does one thing well, and together they compose the complete response to the event.

* * *

9.4 Emitting Events

The Emit action publishes an event to the bus. The action takes an event type and a payload. The event type appears in the result position with an “event” qualifier. The payload follows the “with” preposition and can be any value—a simple variable, an object literal, or a complex expression.

The event type becomes the name used for handler matching. If you emit an event with type “OrderPlaced,” handlers with business activity “OrderPlaced Handler” will receive it. Choose event names that describe what happened in business terms rather than technical terms. “CustomerRegistered” is better than “RecordInserted.” “PaymentDeclined” is better than “ErrorOccurred.”

The payload is the data that handlers will receive. Handlers access this data using the Extract action with the “event” identifier. Include everything that handlers might need to do their work, but avoid including sensitive information that not all handlers should access. The payload is delivered unchanged to every handler, so all handlers see the same data.

A single feature set can emit multiple events. This is common when different subsystems need to react to different aspects of an operation. Creating an order might emit OrderCreated for order processing, PaymentRequired for the payment system, and InventoryUpdate for warehouse management. Each subsystem handles the event relevant to its domain.

* * *

9.5 Accessing Event Data

Within a handler, the event is available through the special “event” identifier. You use the Extract action to pull specific data out of the event payload into local bindings.

The event object contains the payload that was provided when the event was emitted. If the emitter provided a user object as the payload, you can extract that user with appropriate qualifiers. If the emitter provided an object literal with multiple fields, you can extract each field individually.

Beyond the payload, events carry metadata about their origin. The event identifier provides a unique value for correlating logs and traces. The timestamp records when the event was emitted. The source identifies which feature set emitted the event. This metadata is useful for debugging and auditing.

The extraction patterns for events follow the same qualifier syntax used elsewhere in ARO. You chain qualifiers with colons to navigate into nested structures. If the payload contains an order with a customer with an email, you can extract that email directly using multiple qualifiers.

* * *

9.6 Multiple Handlers and Execution

When multiple handlers register for the same event type, all of them execute when that event is emitted. This parallel reaction is one of the most powerful aspects of event-driven architecture because it allows independent modules to respond to the same stimulus without coordinating with each other.

Each handler runs independently. They do not share state. A failure in one handler does not prevent other handlers from running. If one handler encounters an error, that error is logged, but the other handlers continue normally. This isolation makes handlers resilient—a bug in one handler does not bring down the entire system.

The order of handler execution is not specified. The runtime may execute handlers in any order, and that order may vary between executions. If your handlers must execute in a specific sequence, you need to express that through event chaining rather than relying on implicit ordering.

Handlers may execute concurrently if the runtime determines that they are independent. This parallelism can improve performance, but it also means handlers must not assume

exclusive access to shared resources. Design handlers to be safe for concurrent execution, avoiding race conditions in shared state.

* * *

9.7 Event Chains

Events can trigger handlers that emit additional events, creating chains of processing. This pattern allows you to break complex workflows into discrete steps that execute in sequence.

The chain is established through the event naming. When the first event triggers a handler that emits a second event, handlers for that second event run next. Each step in the chain is a separate feature set with its own isolation and error handling.

Event chains are useful for orchestrating multi-step processes. An order processing workflow might start with OrderCreated, which triggers inventory checking. If inventory is available, InventoryReserved triggers payment processing. If payment succeeds, PaymentProcessed triggers fulfillment. Each step emits the event that triggers the next step.

The advantage of chains over monolithic handlers is modularity. Each step can be developed, tested, and modified independently. You can add steps by adding handlers. You can modify a step's implementation without affecting other steps. The overall workflow emerges from the composition of independent parts.

Be cautious of circular chains where A triggers B triggers A. This creates an infinite loop that will exhaust resources. Design your event flows to be acyclic, with clear beginning and end points.

> **Compiler Check**: The ARO compiler detects circular event chains at compile time. If your handlers form a cycle (for example, `Alpha Handler` emits `Beta` and `Beta Handler` emits `Alpha`), you will receive an error:
>
> ```
> error: Circular event chain detected: Alpha -> Beta -> Alpha
>   hint: Event handlers form an infinite loop that will exhaust resources
>   hint: Consider breaking the chain by using different event types or adding termination condit
> ```
>
> This static analysis catches cycles before your code runs, preventing runtime infinite loops.

* * *

9.8 Error Handling in Events

Error handling for events differs from synchronous execution. When a handler fails, the error is logged with full context, but the failure does not propagate to the emitter or to other handlers. Each handler succeeds or fails independently.

This design reflects the fire-and-forget nature of event emission. The emitting feature set has already moved on by the time handlers execute. It cannot meaningfully handle handler failures because it has already returned its result. The isolation is intentional—it prevents cascading failures and keeps the emitter's behavior predictable.

For scenarios where handler success is critical, you need different patterns. You might use synchronous validation before emitting the event, checking conditions that would cause handler failure. You might use compensating events where failure handlers emit events that trigger recovery. You might move critical operations into the emitting feature set itself rather than relying on handlers.

The runtime logs all handler failures. You can configure alerts based on these logs to notify operators when handlers are failing. The logs include the event type, handler name, error message, and full context, providing the information needed to diagnose and fix issues.

* * *

9.9 Best Practices

Name events for business meaning rather than technical operations. The event name should describe what happened in domain terms that non-technical stakeholders would understand. "CustomerRegistered" tells you about a business occurrence. "DatabaseRowInserted" tells you about an implementation detail.

Keep handlers focused on single responsibilities. A handler that sends email and updates analytics and notifies administrators is doing too much. Split these into three handlers that each do one thing well. The event bus will invoke all of them, and each will be easier to understand, test, and maintain.

Design handlers for idempotency when possible. Events might be delivered more than once in some scenarios—retries after transient failures, replays for recovery, or duplicate

emissions due to bugs. If handlers can safely process the same event multiple times without causing incorrect behavior, your system is more resilient.

Avoid circular event chains. If event A triggers handler B which emits event A, you have an infinite loop. The compiler will catch these cycles and report them as errors during `aro check`. Map out your event flows to ensure they are directed acyclic graphs. Each event should lead forward through the workflow, not backward to create cycles.

Document event contracts. The payload of an event is a contract between emitters and handlers. Document what fields are included, their types, and their meanings. When you change an event's structure, update all handlers to accommodate the change.

* * *

9.10 Repository Observers

Repository observers are a specialized form of event handlers that react to changes in repository data. When items are stored, updated, or deleted from a repository, observers automatically receive the change details including both old and new values.

The observer pattern for repositories enables powerful reactive architectures. You can implement audit logging that captures every change with full before-and-after context. You can synchronize data across systems when items change. You can enforce business rules that react to modifications.

To create a repository observer, name your feature set's business activity with the pattern "{repository-name} Observer." When any feature set stores to or deletes from that repository, your observer executes with full change context.

The observer receives an event payload with the repository name, change type (created, updated, or deleted), entity ID if available, and both old and new values. For creates, the old value is nil. For deletes, the new value is nil. For updates, both are present, allowing you to compare what changed.

user-repository <Store> | <Delete>

RepositoryChangedEvent

Audit Observer old + new values

Sync Observer changeType

automatic reactive patterns

This built-in observer mechanism works with the repository semantics. Updates are detected by matching on the entity's id field. When you store an item with an id that already exists, the store becomes an update, and observers see both the previous and new versions of that item.

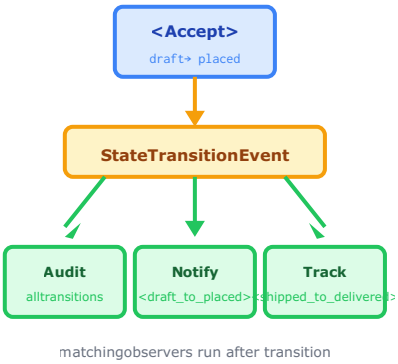
Repository observers follow the same isolation rules as other event handlers. Each observer runs independently with its own context. Failures in one observer do not affect others. The original store or delete operation completes regardless of observer success or failure.

The design enables domain-driven patterns where each bounded context can observe relevant repositories without coupling to the code that modifies them. Audit systems observe without being called. Synchronization happens automatically. The architecture remains loose and extensible.

* * *

9.11 State Observers

State observers are a specialized form of event handler that react to state transitions. When the Accept action successfully transitions a state field, it emits a StateTransitionEvent that observers can handle.



State observers bridge the gap between state machines and event-driven architecture. The Accept action manages the "what"—validating and applying state transitions. Observers manage the "then what"—reacting to those transitions with side effects like audit logging, notifications, or analytics.

The observer pattern is declared through business activity naming. A feature set with business activity "status StateObserver" observes all transitions on fields named "status." Adding a transition filter like "status StateObserver" restricts the observer to only that specific transition. A feature set with just "StateObserver" observes all state transitions regardless of field name.

```
(* Observe ALL status transitions *)
(Audit Order Status: status StateObserver) {
    <Extract> the <orderId> from the <transition: entityId>.
    <Extract> the <fromState> from the <transition: fromState>.
    <Extract> the <toState> from the <transition: toState>.

    <Log> the <audit: message> for the <console>
        with "[AUDIT] Order ${orderId}: ${fromState} -> ${toState}".

    <Return> an <OK: status> for the <audit>.
}

(* Observe ONLY when order is placed *)
(Notify Order Placed: status StateObserver<draft_to_placed>) {
    <Extract> the <orderId> from the <transition: entityId>.
    <Log> the <notification> for the <console>
        with "Order ${orderId} has been placed!".
    <Return> an <OK: status> for the <notification>.
}

(* Observe ONLY when order ships *)
(Send Shipping Notification: status StateObserver<paid_to_shipped>) {
    <Extract> the <order> from the <transition: entity>.
    <Extract> the <tracking> from the <order: trackingNumber>.

    <Log> the <notification> for the <console>
        with "Order shipped! Tracking: ${tracking}".

    <Return> an <OK: status> for the <notification>.
}
```

Within an observer, the transition data is available through the “transition” identifier. You extract specific fields using the familiar qualifier syntax. The fromState and toState fields tell you what changed. The fieldName and objectName provide context about where the change occurred. The entityId offers a convenient way to identify the affected entity. The entity field gives you the complete object after the transition if you need additional context.

Observers execute after the Accept action succeeds. If Accept throws because the current state does not match the expected state, no event is emitted and no observers run. This ensures observers only react to valid transitions. Observers themselves run in isolation—if one observer fails, other observers for the same transition still execute. The failure is logged but does not affect the original Accept action or other observers.

The separation between Accept and observers creates a clean architecture. Accept is synchronous and transactional—it validates and applies the state change atomically. Observers are asynchronous and independent—they react to the change with side effects that

do not affect the core state transition. This separation makes the system more testable, more maintainable, and more extensible.

* * *

Next: Chapter 11 — Application Lifecycle

§ Chapter 11: Application Lifecycle

> “Every program has a beginning, a middle, and an end.”

* * *

9.1 The Three Phases

ARO applications have three distinct lifecycle phases: startup, execution, and shutdown. Each phase has specific responsibilities and corresponding feature sets that can handle them.

The startup phase initializes resources, establishes connections, and prepares the application to do work. This is when configuration is loaded, services are started, and the application transitions from inert code to a running system. Startup must complete successfully for the application to proceed.

The execution phase is where the application does its actual work. For batch applications, this might be a single sequence of operations. For servers and daemons, this is an ongoing process of handling events, requests, and other stimuli. The execution phase can last indefinitely for long-running applications.

The shutdown phase cleans up resources, closes connections, and prepares the application to terminate. This is when pending work is completed, buffers are flushed, and the application transitions from a running system back to inert code. Proper shutdown prevents resource leaks and data loss.

Each phase has a corresponding feature set that you can define to handle its responsibilities. The startup phase uses `Application-Start`, which is required. The shutdown phase uses `Application-End: Success` for normal shutdown and `Application-End: Error` for error shutdown, both of which are optional.

* * *

9.2 Application-Start

Every ARO application must have exactly one feature set named `Application-Start`. This is where execution begins. The runtime looks for this feature set when the application launches and executes it to initialize the application. Without an Application-Start feature set, there is nothing to execute, and the application cannot run.

Having multiple Application-Start feature sets is also an error. If you spread your application across multiple files and accidentally define Application-Start in more than one of them, the runtime reports the conflict and refuses to start. This constraint ensures that there is always exactly one unambiguous entry point.

The business activity (the text after the colon) can be anything descriptive of your application. Common choices include the application name, a description of its purpose, or simply "Application" or "Entry Point." This text has no semantic significance for Application-Start—it is purely documentation.

The startup feature set typically performs several initialization tasks. Loading configuration from files or environment variables is common. Starting services like HTTP servers, database connections, or file watchers is typical for server applications. Publishing values that other feature sets in the same business activity will need is another common task. Each of these tasks is expressed as statements in the feature set.

The startup feature set must return a status to indicate whether initialization succeeded. If any statement fails during startup, the runtime logs the error, invokes the error shutdown handler if one exists, and terminates the application with a non-zero exit code. A successful startup means the application is ready to do work.

* * *

9.3 The Keepalive Action

For applications that need to continue running after startup to process ongoing events, the Keepalive action is essential. Without it, the runtime executes the startup feature set, reaches the return statement, and terminates the application. This is fine for batch applications that do their work during startup, but servers and daemons need to stay running.

The Keepalive action blocks execution at the point where it appears. It allows the event loop to continue processing events—HTTP requests, file system changes, socket messages, timer events, and custom events—while the startup feature set waits. The application remains active, handling events as they arrive.

When the application receives a shutdown signal, either from the user pressing Ctrl+C (SIGINT) or from the operating system sending SIGTERM, the Keepalive action returns. Execution resumes from where it left off, and any statements after the Keepalive execute. Then the return statement completes the startup feature set, which triggers the shutdown phase.

Applications that do not use Keepalive execute their startup statements and immediately terminate. This is appropriate for command-line tools that perform a specific task and exit, data processing scripts that run to completion, or any application where continued execution is not needed. The absence of Keepalive does not indicate an error—it simply indicates that the application has no ongoing work to do.

* * *

9.4 Application-End: Success

The success shutdown handler runs when the application terminates normally. This means the user sent a shutdown signal, the application called for shutdown programmatically, or any other clean termination occurred. It is an opportunity to perform cleanup that should happen on every normal exit.

The handler is optional. If you do not define one, the application terminates without any cleanup phase. For simple applications that do not hold external resources, this is fine. For applications with database connections, open files, or other resources that should be closed properly, defining a success handler is important.

Typical cleanup tasks include stopping services so they stop accepting new work, draining any pending operations so they complete rather than being lost, closing database connections so they are returned to connection pools, flushing log buffers so no messages are lost, and performing any other resource release that should happen on shutdown.

The handler should be designed to complete reasonably quickly. Shutdown has a default timeout, and if the handler takes too long, the process is terminated forcibly. If you have long-running cleanup tasks, consider whether they can be shortened or made asynchronous.

The shutdown handler receives no special input—unlike error shutdown, there is no error context because nothing went wrong. It simply performs its cleanup and returns a status indicating that shutdown completed successfully.

* * *

9.5 Application-End: Error

The error shutdown handler runs when the application terminates due to an unhandled error. This means an exception occurred that was not caught by any handler, a fatal condition was detected, or some other error situation triggered abnormal termination.

Unlike success shutdown, error shutdown provides access to the error that caused the termination. You can extract this error from the shutdown context and use it for logging, alerting, or diagnostic purposes. The error contains information about what went wrong, where it happened, and any associated context.

The handler is optional, but defining one is strongly recommended for production applications. Without it, errors cause silent termination with no opportunity for cleanup or notification. With it, you can ensure that administrators are alerted, logs contain sufficient information for diagnosis, and resources are released even in error scenarios.

Cleanup during error shutdown should be defensive. Some resources might be in inconsistent states due to the error. Cleanup code should be prepared for failures and should continue even if some cleanup steps fail. The goal is best-effort cleanup, not guaranteed perfect cleanup.

The distinction between success and error shutdown allows you to handle these cases differently. Success shutdown might wait for pending work to complete. Error shutdown might skip that wait and proceed directly to resource release. Success shutdown might log a friendly goodbye message. Error shutdown might log a detailed error report.

* * *

9.6 Shutdown Signals

The operating system communicates with processes through signals. ARO handles the common shutdown signals appropriately.

SIGINT is sent when the user presses Ctrl+C in the terminal. ARO treats this as a request for graceful shutdown. The Keepalive action returns, and the success shutdown handler executes. This allows the user to stop a running application cleanly.

SIGTERM is the standard signal for requesting process termination. Process managers, container orchestrators, and system shutdown sequences typically send SIGTERM before escalating to forced termination. ARO handles SIGTERM the same as SIGINT—graceful shutdown with the success handler.

SIGKILL cannot be caught or handled. When a process receives SIGKILL, the operating system terminates it immediately. There is no opportunity for cleanup. This is the last resort for stopping a process that does not respond to SIGTERM. Applications should not rely on SIGKILL for normal operation—if your application requires SIGKILL to stop, something is wrong with its shutdown handling.

The shutdown process has a timeout. If the shutdown handlers do not complete within a reasonable time (typically 30 seconds), the process is terminated forcibly. This prevents hung shutdown handlers from keeping the process alive indefinitely. Design your handlers to complete quickly enough to finish before the timeout.

* * *

9.7 Startup Errors

If the Application-Start feature set fails, the application cannot proceed. The runtime logs the error with full context, invokes the error shutdown handler if one exists, and terminates the process with a non-zero exit code.

Common startup failures include configuration files that do not exist or contain invalid data, services that cannot be reached such as databases or external APIs, permissions that prevent the application from accessing needed resources, and port conflicts where a server cannot bind to its configured port.

The error messages for startup failures follow the same pattern as other ARO errors. They describe what the statement was trying to accomplish in business terms. “Cannot read the config from the file with config.json” tells you exactly what failed. The error includes additional context about why it failed—file not found, permission denied, or similar.

Designing for startup resilience involves validating assumptions early. If your application requires a configuration file, failing fast during startup is better than failing later when the

configuration is first used. The startup feature set is the appropriate place to verify that all prerequisites are met.

* * *

9.8 Best Practices

Always use Keepalive for server applications. If your application starts an HTTP server, file watcher, socket listener, or any other service that should run continuously, the Keepalive action is necessary to prevent immediate termination after startup.

Define both shutdown handlers for production applications. The success handler ensures clean shutdown during normal operation. The error handler ensures that error conditions are logged and resources are released even when things go wrong.

Log lifecycle events for operational visibility. Logging at startup provides confirmation that the application started successfully and with what configuration. Logging at shutdown helps diagnose whether shutdown completed cleanly. These logs are invaluable for debugging operational issues.

Clean up resources in reverse order of acquisition. If you start the database, then the cache, then the HTTP server during startup, stop the HTTP server, then the cache, then the database during shutdown. This order ensures that dependent resources are still available when cleanup needs them.

Keep shutdown handlers fast. Long shutdown times frustrate operators and can cause problems with process managers that expect quick termination. If you have work that takes a long time to complete, consider whether it can be deferred or done asynchronously rather than during shutdown.

* * *

Next: Chapter 12 — Custom Events

§ Chapter 12: Custom Events

> “Design your domain events like you design your domain model.”

* * *

10.1 Domain Events

Custom events represent significant occurrences in your business domain. Unlike the built-in events that the runtime generates for file changes, socket messages, and HTTP requests, custom events are defined by you to capture business-meaningful happenings within your application.

A domain event records that something important occurred. A new user registered. An order was placed. A payment was received. Inventory fell below a threshold. Each event captures a moment in time when the state of the business changed in a way that other parts of the system might care about.

The power of domain events comes from their role in decoupling. The code that causes an event does not need to know what will happen afterward. It simply announces that something occurred. Other code can react to that announcement. This separation allows you to add new reactions without modifying the original code, to test event producers and consumers independently, and to understand each part of the system in isolation.

Designing good domain events requires thinking about your business domain. What are the significant state changes? What information would observers need to react appropriately? How should events relate to each other? These questions parallel the questions you ask when designing a domain model, which is why event design and domain modeling often go hand in hand.

* * *

10.2 Emitting Events

The Emit action publishes an event to the event bus. The event has a type and a payload. The type is specified in the result position with an “event” qualifier. The payload follows the “with” preposition.

The event type becomes the name used for matching handlers. Choose event names that describe what happened rather than what should happen. Use past tense to emphasize that the event records something that already occurred. “CustomerRegistered” and “OrderPlaced” and “PaymentReceived” are good names. “RegisterCustomer” and “PlaceOrder” are commands, not events—they describe actions to be taken, not facts that have been recorded.

Event names should be specific and business-meaningful. “UserUpdated” is vague—what was updated? The user’s email? Their password? Their role? More specific names like “UserEmailChanged,” “UserPasswordReset,” and “UserRoleUpdated” tell handlers exactly what happened, allowing them to react appropriately.

The payload is the data that travels with the event. It should contain everything that handlers might need to do their work. For a UserCreated event, include the full user object or at least the properties that handlers will need. For an OrderPlaced event, include the order details, customer information, and anything else relevant to order processing.

The payload should be self-contained. Handlers should not need to make additional queries to understand the event. If a handler needs the customer’s email address to send a notification, include the email in the event payload rather than forcing the handler to retrieve it separately.

* * *

10.3 Handling Events

A feature set becomes an event handler when its business activity matches the handler pattern. The pattern consists of the event name followed by “Handler.” A feature set with business activity “UserCreated Handler” handles UserCreated events. A feature set with business activity “OrderPlaced Handler” handles OrderPlaced events.

Within a handler, the event is available through the “event” identifier. You use Extract actions to pull data out of the event into local bindings. The event object contains the payload that was provided when the event was emitted, plus metadata about the event itself.

The payload structure depends on what the emitter provided. If the emitter passed a single object as the payload, you can extract properties from that object. If the emitter passed an object literal with multiple fields, you can extract each field by name. The structure of the event is an implicit contract between emitters and handlers—handlers must know what to expect.

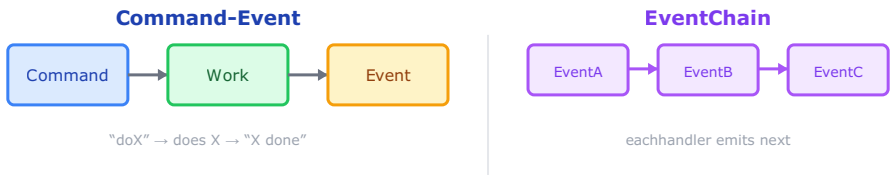
Event metadata includes an identifier for correlating logs and traces, a timestamp recording when the event was emitted, and a source identifying which feature set emitted the event. This metadata is useful for debugging, auditing, and understanding event flows through the system.

Handlers should be focused on single responsibilities. A handler that sends email and updates analytics and notifies administrators is doing too much. Split these into separate handlers that each do one thing well. The event bus will invoke all handlers for the event, and each handler can be developed and tested independently.

* * *

10.4 Event Patterns

Several patterns emerge in how events are used to structure applications.



The command-event pattern separates the action that causes a change from the event that records it. An HTTP handler receives a command (create a user), performs the work (validate data, store user), and emits an event (UserCreated). The command is imperative—it asks for something to happen. The event is declarative—it states what happened. This separation clarifies responsibilities and enables loose coupling.

Event chains occur when handlers emit additional events. An OrderPlaced event might trigger an inventory handler that emits InventoryReserved, which triggers a payment handler that emits PaymentProcessed, which triggers a fulfillment handler. Each step in the chain is a separate handler with its own isolation, error handling, and potential for independent evolution.

The saga pattern uses event chains to implement long-running processes that span multiple steps. A refund saga might involve reversing a payment, restoring inventory, updating the order status, and notifying the customer. Each step emits an event that triggers the next step. If a step fails, compensation events can trigger rollback of previous steps.

Complete Saga Example: Order Processing

Here is a complete order processing saga showing event-driven choreography:

```
(* Step 1: HTTP handler creates order and starts the saga *)
(createOrder: Order API) {
    <Extract> the <order-data> from the <request: body>.
    <Create> the <order> with <order-data>.
    <Store> the <order> in the <order-repository>.

    (* Emit event to start the saga *)
    <Emit> an <OrderPlaced: event> with <order>.

    <Return> a <Created: status> with <order>.
}

(* Step 2: Reserve inventory when order is placed *)
(Reserve Inventory: OrderPlaced Handler) {
    <Extract> the <order> from the <event: order>.
    <Extract> the <items> from the <order: items>.

    (* Reserve each item in inventory *)
    <Retrieve> the <inventory> from the <inventory-service> for <items>.
    <Update> the <inventory> with { reserved: true }.
    <Store> the <inventory> in the <inventory-service>.

    (* Continue the saga *)
    <Emit> an <InventoryReserved: event> with <order>.
}

(* Step 3: Process payment after inventory is reserved *)
(Process Payment: InventoryReserved Handler) {
    <Extract> the <order> from the <event: order>.
    <Extract> the <amount> from the <order: total>.
    <Extract> the <payment-method> from the <order: paymentMethod>.

    (* Charge the customer *)
    <Send> the <charge-request> to the <payment-gateway> with {
        amount: <amount>,
        method: <payment-method>
    }.

    (* Continue the saga *)
    <Emit> a <PaymentProcessed: event> with <order>.
}

(* Step 4: Ship order after payment succeeds *)
(Ship Order: PaymentProcessed Handler) {
    <Extract> the <order> from the <event: order>.

    (* Update order status and create shipment *)
    <Update> the <order> with { status: "shipped" }.
    <Store> the <order> in the <order-repository>.
    <Send> the <shipment-request> to the <shipping-service> with <order>.

    (* Final event in the happy path *)
    <Emit> an <OrderShipped: event> with <order>.
```

```
}

(* Notification handler - runs in parallel with saga *)
(Notify Customer: OrderShipped Handler) {
    <Extract> the <order> from the <event: order>.
    <Extract> the <email> from the <order: customerEmail>.

    <Send> the <shipping-notification> to the <email-service> with {
        to: <email>,
        template: "order-shipped",
        order: <order>
    }.

    <Return> an <OK: status> for the <notification>.
}
```

This saga demonstrates: - **Event chain**: OrderPlaced → InventoryReserved → PaymentProcessed → OrderShipped - **Decoupling**: Each handler focuses on one step, unaware of the others - **Fan-out**: Multiple handlers can listen to the same event (e.g., OrderShipped triggers both shipping and notifications)

Fan-out occurs when multiple handlers react to the same event. An OrderPlaced event might trigger handlers for inventory, payment, notifications, analytics, and fraud checking. All these handlers run when the event is emitted. Each handler focuses on its specific concern, and together they implement the complete response to a new order.

* * *

10.5 Event Design Guidelines

Good event design requires thinking about both producers and consumers.

Include sufficient context in event payloads. Handlers should have what they need without additional queries. If a UserUpdated event only contains the user identifier, every handler must retrieve the user to learn what changed. If the event includes the changes, previous values, who made the change, and when, handlers can react immediately.

Use past tense consistently. Events record what happened, not what should happen. “UserCreated” states a fact. “CreateUser” requests an action. The distinction matters because it clarifies the nature of the communication—events are announcements, not requests.

Be specific rather than generic. “UserUpdated” could mean many things. “UserEmailChanged” is unambiguous. Specific events allow handlers to know exactly what

occurred and whether they should react. A handler that only cares about email changes can ignore password resets if they are separate events.

Treat event payloads as immutable. The payload is a snapshot of state at the moment the event was emitted. Handlers should not expect to modify the payload or to have modifications affect other handlers. Each handler receives an independent view of the event.

Design for evolution. Events are contracts between producers and consumers. Changing an event's structure can break consumers. When you add fields, make them optional so existing consumers continue to work. When you remove fields, ensure no consumers still depend on them. Version events if incompatible changes are necessary.

* * *

10.6 Error Handling in Events

Event handlers run in isolation. If one handler fails, other handlers for the same event still run. The emitting feature set is not affected by handler failures—it continues with its own execution regardless of what handlers do.

This isolation reflects the fire-and-forget nature of event emission. The emitter announces what happened and moves on. It does not wait for handlers to complete, does not receive their results, and does not fail if they fail. This makes event emission a non-blocking operation and prevents cascading failures.

For scenarios where handler success is important, additional patterns help. Compensation events can trigger recovery when things fail. A PaymentFailed event can trigger handlers that cancel the order and notify the customer. The failure handler runs as a reaction to the failure event, providing a mechanism for recovery without coupling the original operation to error handling.

The runtime logs all handler failures with full context. Operators can monitor these logs to detect failing handlers. Alerts can trigger when failure rates exceed thresholds. The information in the logs—event type, handler name, error message, timestamp, correlation identifier—supports diagnosis and debugging.

Designing handlers for idempotency provides resilience. If a handler can safely process the same event multiple times without incorrect behavior, temporary failures can be recovered by reprocessing the event. This is particularly valuable in distributed systems where exactly-once delivery is difficult to guarantee.

* * *

10.7 Best Practices

Name events from the perspective of the domain, not the infrastructure. “CustomerJoinedLoyaltyProgram” is a domain event. “DatabaseRowInserted” is an infrastructure event. Domain events communicate business meaning; infrastructure events communicate implementation details. Prefer domain events because they remain stable as implementations change.

Document the contract between event producers and consumers. The payload structure is an implicit contract—producers must provide what consumers expect. Documenting this contract makes the expectation explicit. Include what fields are present, their types, and their semantics. When the contract changes, communicate the change to all affected parties.

Use events for cross-cutting concerns. Audit logging, analytics, notifications, and other concerns that touch many parts of the application are natural fits for events. The code that creates a user does not need to know about audit logging—it just emits UserCreated, and an audit handler captures it.

Test handlers in isolation. Because handlers are independent feature sets with well-defined inputs (the event), they are straightforward to test. Construct a mock event with the expected payload, invoke the handler, and verify the behavior. This unit testing approach scales to complex systems.

Avoid circular event chains. If event A triggers a handler that emits event B, and event B triggers a handler that emits event A, you have an infinite loop. The ARO compiler detects these cycles at compile time and reports them as errors, so you will catch this problem before your code runs. Map your event flows to ensure they form directed acyclic graphs with clear start and end points.

* * *

10.8 Compiler Validation

The ARO compiler performs static analysis on your event handlers to detect potential issues before runtime.

Circular Event Chain Detection: The compiler builds a graph of event flows by analyzing which handlers emit which events. If a cycle is detected (for example, `Alpha Handler` emits `Beta` and `Beta Handler` emits `Alpha`), the compiler reports an error:

```
error: Circular event chain detected: Alpha -> Beta -> Alpha
  hint: Event handlers form an infinite loop that will exhaust resources
  hint: Consider breaking the chain by using different event types or adding termination condition
```

This check examines all Emit statements, including those inside Match statements and ForEach loops, treating any potential emission path as part of the event flow graph.

Breaking Cycles: If you need handlers to communicate back and forth, consider these approaches: - Use a termination condition based on data in the event payload - Design a linear workflow where each step moves forward, not backward - Introduce a new event type that represents a terminal state - Move the repeated logic into a single handler rather than chaining

The goal is to ensure that every event chain has a clear end point where no further events are emitted.

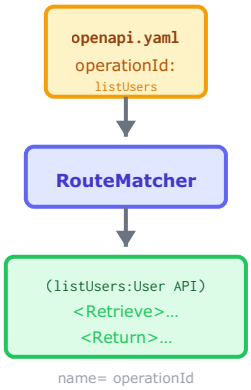
Next: Chapter 13 — OpenAPI Integration

§ Chapter 13: OpenAPI Integration

> "Your contract is your router."

* * *

11.1 Contract-First Development



ARO embraces contract-first API development. Rather than defining routes in code and hoping documentation stays synchronized, you define your API in an OpenAPI specification file, and ARO uses that specification to configure routing automatically. The specification is the source of truth; the code implements what the specification declares.

This approach inverts the typical relationship between code and documentation. In most frameworks, you write code and generate documentation from it. In ARO, you write the specification and implement handlers for it. The specification comes first, defining what the API looks like from a client's perspective. The implementation comes second, fulfilling the promises made by the specification.

The connection between specification and code is the operation identifier. Each operation in your OpenAPI specification has an operationId that uniquely identifies it. When a request arrives, ARO determines which operation it matches based on the path and method, looks up the operationId, and triggers the feature set with that name. The feature set name becomes the operationId; the operationId becomes the feature set name.

This design ensures that your API cannot drift from its documentation. If the specification declares an operation, you must implement a feature set with that name, or clients receive an error. If you implement a feature set that does not correspond to an operation, it never receives HTTP requests. The specification and implementation are bound together.

* * *

11.2 The OpenAPI Requirement

ARO's HTTP server depends on the presence of an OpenAPI specification file in the application directory. The runtime checks for specification files in this order of precedence:

1. `openapi.yaml` (preferred)
2. `openapi.yml`
3. `openapi.json`

The first file found is used as the API contract. Without any of these files, no HTTP server starts. No port is opened. No requests are received.

This requirement is deliberate. It enforces the contract-first philosophy at the framework level. You cannot accidentally create an undocumented API because the documentation is required for the API to exist. You cannot forget to update documentation when changing routes because changing routes means changing the specification.

The requirement also simplifies the runtime. There is no route registration API, no decorator syntax for paths, no configuration file for endpoints. The OpenAPI specification provides all of this information in a standard format that tools throughout the industry understand. Your API specification can be viewed in Swagger UI, validated with standard tools, and used to generate client libraries, all because it follows the OpenAPI standard.

If you are building an application that does not expose an HTTP API—a file processing daemon, a socket server, a command-line tool—you simply omit the openapi.yaml file. The application runs normally; it just does not handle HTTP requests.

* * *

11.3 Operation Identifiers

The operationId is the key that connects HTTP routes to feature sets. When you define an operation in your OpenAPI specification, you assign it an operationId. When you implement the handler in ARO, you create a feature set with that identifier as its name.

Operation identifiers should be descriptive verbs that indicate what the operation does. Common conventions include verb-noun patterns like listUsers, createOrder, getProductById, and deleteComment. The identifier should make sense when read in isolation, as it will appear in logs, error messages, and feature set declarations.

Each operation in your specification must have a unique operationId. The OpenAPI standard requires this, and ARO relies on it for routing. If two operations shared an identifier, there would be ambiguity about which feature set should handle which request. The uniqueness constraint eliminates this possibility.

When a request arrives that matches a path and method in your specification, ARO looks up the corresponding operationId and searches for a feature set with that name. If found, the feature set executes with the request context available for extraction. If not found, ARO returns a 501 Not Implemented response indicating that the operation exists in the specification but has no handler.

* * *

11.4 Route Matching

ARO matches incoming HTTP requests to operations through a two-step process. First, it matches the request path against the path templates in the specification. Second, it matches the HTTP method against the methods defined for that path. The combination of path and method identifies a unique operation.

Path templates can include parameters enclosed in braces. A template like /users/{id} matches paths like /users/123 or /users/abc. When a match occurs, the actual value from the URL is extracted and made available as a path parameter. The parameter name in the template (id in this example) becomes the key for accessing the value.

Multiple methods can be defined for the same path. The /users path might support GET for listing users and POST for creating users. Each method has its own operationId and its own feature set. A GET request to /users triggers listUsers; a POST request to /users triggers createUser. The path is the same, but the operations are different.

Requests that do not match any path receive a 404 response. Requests that match a path but use an undefined method receive a 405 Method Not Allowed response. These responses are generated automatically based on the specification; you do not write code to handle unmatched routes.

* * *

11.5 Automatic Server Startup

The HTTP server starts automatically when an `openapi.yaml` file is present in your application directory. There is no explicit Start action required for HTTP services. When the runtime discovers the OpenAPI specification during application initialization, it reads the file, configures routing based on its contents, and begins accepting requests on the default port (8080).

After the server starts, you use the Keepalive action to keep the application running and processing requests. Without Keepalive, the application would start the server and immediately terminate:

```
(Application-Start: User API) {
    <Log> the <startup: message> for the <console> with "API starting...".
    <Keepalive> the <application> for the <events>.
    <Return> an <OK: status> for the <startup>.
}
```

You can configure the port on which the server listens using environment variables or configuration files. This flexibility allows you to run multiple services on different ports or to conform to container orchestration requirements.

The server starts synchronously during initialization. If the port is already in use or binding fails for any other reason, the startup fails with an appropriate error. This fail-fast behavior ensures you know immediately if the server cannot start, rather than discovering the problem later when requests fail.

* * *

11.6 Request Context

When a feature set handles an HTTP request, it has access to information about that request through special context identifiers. You use the Extract action to pull specific pieces of information into local bindings.

Path parameters are values extracted from the URL based on the path template. If your template is /users/{id} and the request URL is /users/123, the path parameter “id” has the value “123”. You access this through the pathParameters identifier with the parameter name as a qualifier.

Query parameters are the key-value pairs in the URL's query string. A request to /users?limit=10&offset=20 has query parameters "limit" and "offset". You access these through the queryParameters identifier. Query parameters are optional by default; extracting a parameter that was not provided produces an empty or missing value rather than an error.

The request body is the content sent with POST, PUT, and PATCH requests. For JSON content, the runtime parses the body into a structured object that you can extract and navigate. You access the body through the request identifier with "body" as the qualifier.

Headers are the HTTP headers sent with the request. Authentication tokens, content types, and other metadata arrive as headers. You access these through the headers identifier with the header name as a qualifier. Header names are case-insensitive per the HTTP specification.

* * *

11.7 Response Mapping

ARO maps return statements to HTTP responses based on the status qualifier you provide. The qualifier determines the HTTP status code, and the payload becomes the response body.

Common status qualifiers include OK for 200 responses, Created for 201 responses when a resource is created, Accepted for 202 when processing is deferred, and NoContent for 204 when there is no response body. Error statuses include BadRequest for 400, NotFound for 404, and Conflict for 409.

The payload you provide with the response becomes the response body, typically serialized as JSON. You can return a single object, an array, or an object literal that you construct inline. The runtime handles serialization and sets appropriate content-type headers.

If your feature set fails rather than returning normally, the runtime generates an error response. The status code depends on the type of failure—not found errors become 404, validation errors become 400, internal errors become 500. The response body contains the error message generated from the failed statement.

* * *

11.8 Validation

OpenAPI specifications can include schemas that define the structure and constraints of request bodies and responses. ARO can validate incoming requests against these schemas, rejecting invalid requests before they reach your feature set.

Automatic validation can be enabled when starting the server. With validation enabled, the runtime checks each incoming request body against the schema defined in the specification. If the request does not conform, the client receives a 400 response with details about which validations failed.

Manual validation is an alternative when you want more control. You extract the request body and then validate it explicitly using the Validate action with a reference to the schema. This approach lets you perform additional processing before or after validation, or handle validation failures in custom ways.

Schema validation provides a first line of defense against malformed requests. It ensures that your feature set receives data in the expected structure with the expected types. This eliminates the need for defensive type checking in your business logic and catches problems at the API boundary where they can be reported clearly to clients.

* * *

11.9 Best Practices

Design your API specification before writing implementation code. Think about what resources your API exposes, what operations clients need to perform, and what data structures are involved. Write this design down in OpenAPI format. Then implement feature sets to fulfill the specification.

Choose operation identifiers that describe what the operation does in clear, consistent terms. Use verb-noun patterns like listUsers, createOrder, getProductById. Avoid generic names like “handle” or “process” that do not convey meaning. The identifier appears in your feature set declarations, in logs, and in error messages, so clarity matters.

Group related operations using tags in your OpenAPI specification. Tags help organize documentation and make the specification easier to navigate. A user management API might tag all user-related operations with “Users” and all authentication operations with “Auth.”

Document the possible responses for each operation. Clients need to know not just the success response but also what error responses they might receive and under what conditions. This documentation helps API consumers handle all cases appropriately.

Keep your specification and implementation synchronized. When you change the API, update the specification first, then update the implementation. When you add new operations, add them to the specification first. The contract should always accurately reflect what the API does.

* * *

Next: Chapter 14 — HTTP Feature Sets

§ Chapter 14: HTTP Feature Sets

> “Every endpoint tells a story.”

* * *

12.1 HTTP Handler Basics

HTTP feature sets handle requests routed through the OpenAPI contract. When a request matches a path and method defined in your specification, the corresponding feature set executes with the request data available for extraction.

The feature set name must exactly match the operationId from your OpenAPI specification. This matching is case-sensitive. If your specification defines an operation with operationId “getUser”, you must create a feature set named “getUser”—not “GetUser”, not “get-user”, not “getuser”. The names must be identical.

A typical HTTP handler follows a predictable pattern. It extracts data from the request, performs some processing, and returns a response. The extraction pulls relevant information from path parameters, query parameters, headers, or the request body. The processing might retrieve data from repositories, create new entities, validate inputs, or compute results. The response returns an appropriate status code with optional data.

The business activity (the text after the colon in the feature set declaration) can be anything descriptive. Common choices include the name of the API or service, a description of the resource being managed, or a broader category that groups related operations. This text appears in logs and helps readers understand the context.

* * *

12.2 CRUD Operations

Most HTTP APIs implement CRUD operations—Create, Read, Update, and Delete—for their resources. Each operation has a characteristic pattern in ARO.

List operations retrieve collections of resources. They typically extract pagination parameters from the query string, retrieve matching records from a repository, and return the collection. The response often includes metadata about the total count, current page, and whether more records exist.

Read operations retrieve a single resource by identifier. They extract the identifier from the path, retrieve the matching record, and return it. If no record matches, the runtime generates a not-found error that becomes a 404 response.

Create operations make new resources. They extract data from the request body, validate it against a schema, create the new entity, store it in a repository, and return it with a Created status. They often emit an event so other parts of the system can react to the new resource.

Update operations modify existing resources. They extract the identifier from the path and the update data from the body, retrieve the existing record, merge the updates, store the result, and return the updated entity. Full updates (PUT) replace all fields; partial updates (PATCH) modify only the provided fields.

Delete operations remove resources. They extract the identifier from the path, delete the matching record from the repository, and return a NoContent status indicating success. They often emit an event so other parts of the system can react to the deletion.

* * *

12.3 Request Data Access

HTTP handlers have access to various parts of the incoming request through special context identifiers. Each type of request data has its own identifier and access pattern.

Path parameters are values embedded in the URL path based on the template defined in your OpenAPI specification. A template like /users/{id}/orders/{orderId} defines two path parameters: “id” and “orderId”. You extract these using the pathParameters identifier with the parameter name as a qualifier.

Query parameters are the key-value pairs in the URL’s query string. A request to /search?q=widgets&page=2 has query parameters “q” and “page”. You extract these using the queryParameters identifier. Query parameters are typically optional; extracting one that was not provided yields an empty value rather than an error.

The request body contains data sent with POST, PUT, and PATCH requests. For JSON content, the runtime parses the body into a structured object. You extract it using the request

identifier with “body” as the qualifier. You can then extract individual fields from the body using additional qualifiers.

Headers contain metadata about the request. Authentication tokens typically arrive in the Authorization header. Content type information appears in the Content-Type header. Custom headers can carry application-specific data. You extract headers using the headers identifier with the header name as a qualifier.

The full request object provides access to additional information including the HTTP method, the original path, and all headers as a collection. This is useful for debugging or when you need information not available through the specific accessors.

* * *

12.4 Response Patterns

ARO provides a variety of status qualifiers for different response scenarios. Choosing the right status communicates the outcome clearly to clients.

Success responses indicate the operation completed as expected. OK (200) is the standard success status for retrieving data or completing an operation. Created (201) indicates a new resource was created, typically used with POST requests. Accepted (202) indicates the request was received for asynchronous processing. NoContent (204) indicates success with no response body, typically used for DELETE operations.

Error responses indicate something prevented successful completion. BadRequest (400) indicates the client sent invalid data. Unauthorized (401) indicates authentication is required. Forbidden (403) indicates the client lacks permission for the operation. NotFound (404) indicates the requested resource does not exist. Conflict (409) indicates the request conflicts with current state, such as creating a duplicate.

The response body can be a single object, an array, or an object literal constructed inline. For single resources, return the entity directly. For collections, return an array or a structured object with the array and metadata. For errors, return a structured object with error details.

Collection responses often include pagination metadata. Beyond the data array, include the total count, current page, page size, and whether more pages exist. This information helps clients navigate through large result sets.

* * *

12.5 Authentication and Authorization

Many APIs require authentication to identify the caller and authorization to verify permissions. ARO handles these concerns through request data extraction and validation.

Authentication typically involves extracting a token from the Authorization header, validating it against an authentication service or by verifying its signature, and extracting identity information such as a user identifier or role list. The validated identity becomes available for use in subsequent statements.

Authorization checks whether the authenticated identity has permission for the requested operation. This might involve checking a role list against required roles, querying a permissions service, or applying custom authorization logic. If authorization fails, you return a Forbidden status.

The pattern is to extract and validate authentication early in the feature set, before performing any business logic. This ensures that unauthenticated or unauthorized requests fail fast with appropriate error responses rather than partially executing before failing.

Custom actions can encapsulate complex authentication and authorization logic. A ValidateToken action might handle JWT verification, signature checking, and expiration validation. A CheckPermission action might query a permissions database or evaluate policy rules. These actions keep your feature sets focused on business logic.

* * *

12.6 Nested Resources

APIs often model relationships between resources through nested URL paths. A user has orders; an order belongs to a user. The path /users/{userId}/orders represents the orders belonging to a specific user.

Nested resource paths involve multiple path parameters. You extract each parameter by name. The parent identifier constrains which records to consider; the child identifier (if present) identifies a specific record within that constraint.

When listing nested resources, include the parent identifier in your repository query. When creating nested resources, include the parent identifier in the stored record. When retrieving or modifying specific nested resources, verify that the child belongs to the specified parent.

The nesting expresses ownership and access control. A request for /users/123/orders/456 should only succeed if order 456 actually belongs to user 123. Your where clause should include both constraints to enforce this relationship.

Complete Nested Resource Example

Here is a complete example for managing order items as a nested resource under orders:

OpenAPI specification (partial):

```
paths:
  /orders/{orderId}/items:
    get:
      operationId: listOrderItems
    post:
      operationId: createOrderItem
  /orders/{orderId}/items/{itemId}:
    get:
      operationId: getOrderItem
    put:
      operationId: updateOrderItem
    delete:
      operationId: deleteOrderItem
```

ARO handlers:

```
(* List all items for an order *)
(listOrderItems: Order API) {
    <Extract> the <order-id> from the <pathParameters: orderId>.

    (* Verify order exists *)
    <Retrieve> the <order> from the <order-repository> where id = <order-id>.

    (* Get items for this order *)
    <Retrieve> the <items> from the <item-repository> where orderId = <order-id>.

    <Return> an <OK: status> with <items>.
}

(* Get a specific item, verifying it belongs to the order *)
(getOrderItem: Order API) {
    <Extract> the <order-id> from the <pathParameters: orderId>.
    <Extract> the <item-id> from the <pathParameters: itemId>.

    (* Retrieve with both constraints to enforce ownership *)
    <Retrieve> the <item> from the <item-repository>
        where id = <item-id> and orderId = <order-id>.

    <Return> an <OK: status> with <item>.
}

(* Create a new item for an order *)
(createOrderItem: Order API) {
    <Extract> the <order-id> from the <pathParameters: orderId>.
    <Extract> the <item-data> from the <request: body>.

    (* Verify order exists before adding item *)
    <Retrieve> the <order> from the <order-repository> where id = <order-id>.

    (* Create item with parent reference *)
    <Create> the <item> with {
        orderId: <order-id>,
        productId: <item-data>.productId,
        quantity: <item-data>.quantity,
        price: <item-data>.price
    }.

    <Store> the <item> in the <item-repository>.

    (* Recalculate order total *)
    <Emit> an <OrderItemAdded: event> with { order: <order>, item: <item> }.

    <Return> a <Created: status> with <item>.
}

(* Delete an item from an order *)
(deleteOrderItem: Order API) {
    <Extract> the <order-id> from the <pathParameters: orderId>.
    <Extract> the <item-id> from the <pathParameters: itemId>.
```

```
    (* Verify ownership before deletion *)
    <Retrieve> the <item> from the <item-repository>
        where id = <item-id> and orderId = <order-id>.

    <Delete> the <item> from the <item-repository>.

    <Emit> an <OrderItemRemoved: event> with { orderId: <order-id>, itemId: <item-id> }.

    <Return> a <NoContent: status> for the <deletion>.
}
```

Key patterns in this example: - **Parent verification**: Always verify the parent exists before creating nested resources - **Ownership enforcement**: Include both parent and child IDs in where clauses - **Event emission**: Notify other handlers when nested resources change

Deep nesting (more than two levels) can make URLs unwieldy and logic complex. Consider whether deep nesting is necessary or whether flatter alternatives would serve your API design better.

* * *

12.7 Best Practices

Keep handlers focused on single responsibilities. A handler that retrieves a user should not also retrieve their orders, payments, and statistics. If clients need aggregated data, provide a separate endpoint or use a pattern like GraphQL that's designed for flexible queries.

Use consistent naming across your API. If you name one operation "listUsers", name similar operations "listOrders" and "listProducts", not "getOrders" or "findProducts". Consistency makes your API predictable and easier to learn.

Emit events for side effects rather than performing them inline. When creating a user, emit a UserCreated event rather than directly sending welcome emails, updating analytics, and notifying administrators. Event handlers keep side effects separate and make the core logic clearer.

Validate inputs early. Extract and validate request data at the beginning of your feature set, before performing any business logic. This ensures that invalid requests fail immediately with clear error messages rather than partially executing.

Return appropriate status codes. Use Created for successful POST requests that create resources. Use NoContent for successful DELETE requests. Use specific error codes rather than generic 500 responses. Clients rely on status codes to understand outcomes.

* * *

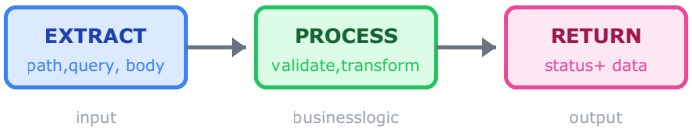
Next: Chapter 15 — Request/Response Patterns

§ Chapter 15: Request/Response Patterns

> "Input, transform, output. The rhythm of every API."

* * *

13.1 The Request-Response Cycle



Every HTTP handler follows a fundamental cycle: receive a request, process it, and return a response. This cycle structures how you think about and write HTTP feature sets in ARO.

The cycle begins with extraction. Data arrives in the request through various channels—path parameters embedded in the URL, query parameters in the query string, headers containing metadata, and a body containing the primary payload. Your feature set pulls out the pieces it needs using Extract actions.

The middle of the cycle is processing. This is where your business logic lives. You might validate the extracted data against schemas or business rules. You might retrieve existing data from repositories. You might create new entities, compute derived values, or transform data between formats. You might store results or emit events.

The cycle concludes with the response. You return a status indicating the outcome and optionally include data in the response body. The status code communicates success or failure; the body provides details. The runtime serializes your response and sends it to the client.

This cycle is the same regardless of what your API does. A simple health check endpoint and a complex multi-step transaction both follow the same pattern: extract, process, return.

* * *

13.2 Extraction Patterns

Extraction is the first step in handling any request. You need to get data out of the request and into local bindings where you can work with it.

Simple extraction pulls a single value from a known location. Extracting a user identifier from path parameters, a search query from query parameters, or an authentication token from headers are all simple extractions. Each uses the Extract action with the appropriate context identifier and qualifier.

Nested extraction navigates into structured data. The request body is often a JSON object with nested properties. You can extract the entire body and then extract individual fields from it, or you can use chained qualifiers to navigate directly to nested values.

Default values handle optional data. Query parameters are typically optional—clients may or may not include them. When you extract an optional parameter, you might get an empty value. You can use the Create action with an “or” clause to provide a default when the extracted value is missing.

Multiple extractions gather all the data you need. A complex handler might extract several path parameters, multiple query parameters, the request body, and one or more headers. Each extraction creates a binding that subsequent statements can use.

The pattern is to perform all extractions early in the feature set, before any processing logic. This makes it clear what data the handler needs and ensures that missing required data causes immediate failure rather than partial processing.

* * *

13.3 Validation Patterns

Validation ensures that extracted data meets expectations before you use it in business logic. Invalid data caught early produces clear error messages; invalid data caught late produces confusing failures.

Schema validation checks that data conforms to a defined structure. OpenAPI specifications include schemas for request bodies, and ARO can validate against these schemas. The Validate action compares data against a schema and fails if the data does not conform.

Business rule validation checks constraints beyond structure. A quantity must be positive. A date range must have the start before the end. An email must be in a valid format. These rules express business requirements that pure schema validation cannot capture.

Cross-field validation checks relationships between multiple values. A password confirmation must match the password. A shipping address is required when the delivery method is not pickup. These validations involve multiple extracted values and their relationships.

Custom validation actions encapsulate complex validation logic. When validation rules are elaborate or shared across multiple handlers, implementing them as custom actions keeps your feature sets focused on the business flow rather than validation details.

* * *

13.4 Transformation Patterns

Transformation is the heart of request processing. You take input data and produce output data through various operations.

Entity creation transforms raw input into domain objects. You extract unstructured data from the request, perhaps validate it, and create a typed entity. The created entity has a well-defined structure and possibly additional computed properties.

Data enrichment augments core data with related information. You retrieve a primary entity and then retrieve additional entities referenced by the primary one. The enriched result combines the primary entity with its related data.

Aggregation computes summary values from collections. You retrieve a set of records and compute totals, counts, averages, or other aggregate values. The response includes these computed values rather than or in addition to the raw records.

Format transformation converts between representations. You might transform an internal entity into an API response format, convert between date representations, or restructure nested data into a flat format.

Each transformation takes bound values as input and produces new bindings as output. The sequence of transformations builds up the data needed for the response.

* * *

13.5 Response Patterns

Response patterns determine how you communicate outcomes to clients. The combination of status code and response body tells clients what happened and provides any resulting data.

Success with data returns a status indicating success along with the relevant data. For retrievals, this is typically OK with the retrieved entity. For creations, this is typically Created with the new entity. The data might be a single object, an array, or a structured object containing data and metadata.

Success without data indicates the operation completed but there is nothing to return. Delete operations typically use NoContent because the deleted resource no longer exists. Some update operations might also return NoContent if the updated state is not needed by the client.

Collection responses return multiple items, often with pagination metadata. Beyond the array of items, include information about the total count, current page, page size, and whether additional pages exist. This metadata helps clients navigate large result sets.

Error responses indicate what went wrong. The status code categorizes the error—client error versus server error, not found versus forbidden. The response body provides details including an error message and possibly additional context like field-level validation errors.

Structured responses maintain consistency across endpoints. Rather than returning raw data for success and structured objects for errors, consider always returning a consistent structure with “data” and “error” fields, or “data” and “meta” fields. Consistency makes your API easier for clients to consume.

* * *

13.6 Common Patterns

Several patterns recur across APIs and have established solutions in ARO.

Get-or-create retrieves an existing resource if it exists or creates a new one if it does not. This pattern is useful for idempotent operations where clients want to ensure a resource exists without caring whether it was already present.

Upsert updates an existing resource if found or creates it if not. Unlike get-or-create, upsert applies updates to existing resources rather than returning them unchanged. The identifier might be a natural key like an email address rather than a generated identifier.

Bulk operations process multiple items in a single request. Creating, updating, or deleting multiple resources at once reduces round trips compared to processing each item individually. The response might summarize results rather than returning all processed items.

Search with filters handles complex queries. Rather than defining separate endpoints for each query variation, a single search endpoint accepts filter parameters that constrain the results. The handler builds a query from the provided filters and executes it against an index or database.

* * *

13.7 Response Headers

Beyond the status code and body, responses can include headers that provide additional metadata or instructions to clients.

Content disposition headers control how browsers handle downloaded files. For file downloads, you set the Content-Disposition header to indicate that the response should be saved as a file with a particular name.

Cache control headers tell clients and intermediaries how long to cache the response. Setting appropriate cache headers reduces load on your server and improves client performance for responses that do not change frequently.

Custom headers can carry application-specific metadata. Rate limit information, correlation identifiers, and pagination links are examples of data that might travel in headers rather than the body.

The Return action can include header specifications that the runtime applies to the HTTP response. Headers are key-value pairs that augment the response status and body.

* * *

13.8 Best Practices

Extract and validate early. Get all the data you need from the request at the beginning of your handler. Validate it immediately after extraction. This pattern ensures that invalid requests fail fast with clear errors.

Use meaningful response structures. Consistent response shapes across your API make client development easier. Consider standard patterns like wrapping data in a “data” field and including metadata in a “meta” field.

Be consistent across endpoints. If one endpoint returns pagination in a particular format, all endpoints with pagination should use the same format. If one endpoint includes error details in a particular structure, all endpoints should use the same structure.

Document your response shapes. Clients need to know what to expect from your API. The OpenAPI specification should document not just the types but also the structure and meaning of responses. Good documentation reduces client development time and support requests.

Handle edge cases explicitly. What happens when a list endpoint finds no matching items—an empty array or a 404? What happens when an optional related resource is missing? Decide these behaviors intentionally and implement them consistently.

* * *

Next: Chapter 16 — Built-in Services

§ Chapter 16: Built-in Services

> "Batteries included."

* * *

16.1 Available Services

ARO provides five built-in services that handle common infrastructure concerns: an HTTP server for serving web requests, an HTTP client for making outbound requests, a file system service for reading and writing files, and socket services for TCP communication.

These services are available without additional configuration or dependencies. When you need to serve HTTP requests, you start the HTTP server. When you need to make outbound API calls, you use the HTTP client. When you need to read configuration files or write data, you use the file system service. When you need low-level TCP communication, you use the socket services.

Each service follows the same pattern of interaction. You start or configure the service using an action, you interact with it through subsequent actions, and you stop it during shutdown. Services that produce events—file changes, socket messages—trigger event handlers that you define.

The services are designed to be sufficient for most application needs while remaining simple. If you need more specialized functionality, you can create custom actions that wrap specialized libraries.

* * *

16.2 HTTP Server

The HTTP server handles incoming HTTP requests based on your OpenAPI specification. It provides a production-capable server built on SwiftNIO that efficiently handles concurrent connections.

Starting the server is a single statement that tells the runtime to load the OpenAPI specification, configure routes, and begin listening for connections. The server typically starts during Application-Start and runs until shutdown. Without the Keepalive action, the application would start the server and immediately exit.

You can configure the port on which the server listens. The default is typically port 8080, but you can specify any available port. You can also specify a host address to control which network interfaces accept connections—binding to "0.0.0.0" accepts connections from any interface, while binding to "127.0.0.1" accepts only local connections.

Additional configuration options control request validation, timeout behavior, maximum body size, and CORS settings. Validation tells the server to check incoming requests against the OpenAPI schemas before routing them to handlers. Timeout settings control how long requests can run. CORS settings control cross-origin access for browser-based clients.

Stopping the server during shutdown allows it to complete in-flight requests gracefully. The server stops accepting new connections, waits for existing requests to complete, and then releases resources.

* * *

16.3 HTTP Client

The HTTP client makes outbound HTTP requests to external services. It provides a high-performance client built on AsyncHTTPClient that handles connection pooling, timeouts, and retries.

Simple GET requests use the Fetch action with a URL. The action makes the request, waits for the response, and binds the result. You can then extract the response body, status code, and headers.

POST, PUT, DELETE, and other methods use the Send action with method, body, and header configuration. The body is serialized as JSON by default. Headers can include authentication tokens, content type specifications, and other metadata.

The client supports various configuration options. Timeout settings control how long to wait for a response. Retry settings enable automatic retry on transient failures. Follow redirect settings control whether redirects are followed automatically.

Error handling follows the same happy path philosophy as other ARO operations. If a request fails—connection refused, timeout, server error—the runtime generates an appropriate error message. You do not write explicit error handling for HTTP failures.

* * *

16.4 File System Service

The file system service provides comprehensive operations for reading, writing, and managing files and directories. It handles the mechanics of file I/O while you focus on what data to read or write.

Reading Files

Reading files uses the Read action with a file path. The action reads the file contents and binds them to a result. For JSON files, the content is parsed into a structured object. For text files, the content is a string. The path can be relative to the application directory or absolute.

```
<Read> the <content> from the <file: "./README.md">.
<Read> the <config: JSON> from the <file: "./config.json">.
<Read> the <image: bytes> from the <file: "./logo.png">.
```

Writing Files

Writing files uses the Write action with data and a path. The action serializes the data and writes it to the specified location. Parent directories are created automatically if they do not exist.

```
<Write> the <report> to the <file: "./output/report.txt">.
<Write> the <data: JSON> to the <file: "./export.json">.
```

Appending to Files

The Append action adds content to the end of an existing file, creating the file if it does not exist.

```
<Append> the <log-line> to the <file: "./logs/app.log">.
```

Checking File Existence

The Exists action checks whether a file or directory exists at a given path.

```
<Exists> the <found> for the <file: "./config.json">.

when <found> is false {
    <Log> the <warning> for the <console> with "Config not found!".
}
```

Getting File Information

The Stat action retrieves detailed metadata about a file or directory, including size, modification dates, and permissions.

```
<Stat> the <info> for the <file: "./document.pdf">.
<Log> the <size> for the <console> with <info: size>.
<Log> the <modified> for the <console> with <info: modified>.
```

The result contains: name, path, size (bytes), isFile, isDirectory, created, modified, accessed, and permissions.

Listing Directory Contents

The List action retrieves the contents of a directory. You can filter by glob pattern and list recursively.

```
(* List all files in a directory *)
<Create> the <uploads-path> with "./uploads".
<List> the <entries> from the <directory: uploads-path>.

(* Filter with glob pattern *)
<Create> the <src-path> with "./src".
<List> the <aro-files> from the <directory: src-path> matching "*.aro".

(* List recursively *)
<Create> the <project-path> with "./project".
<List> the <all-files> from the <directory: project-path> recursively.
```

Each entry contains: name, path, size, isFile, isDirectory, and modified.

Creating Directories

The CreateDirectory action creates a directory, including any necessary parent directories.

```
<CreateDirectory> the <output-dir> to the <path: "./output/reports/2024">.
```

Copying Files and Directories

The Copy action copies a file or directory to a new location. Directory copies are recursive by default.

```
<Copy> the <file: "./template.txt"> to the <destination: "./copy.txt">.
<Copy> the <directory: "./src"> to the <destination: "./backup/src">.
```

Moving and Renaming

The Move action moves or renames a file or directory.

```
<Move> the <file: "./draft.txt"> to the <destination: "./final.txt">.
<Move> the <file: "./inbox/report.pdf"> to the <destination: "./archive/report.pdf">.
```

Deleting Files

The Delete action removes a file from the file system.

```
<Delete> the <file: "./temp/cache.json">.
```

File Watching

File watching monitors a directory for changes and emits events when files are created, modified, or deleted. You start watching during Application-Start by specifying the directory to monitor. When changes occur, the runtime emits File Event events that your handlers can process. This is useful for applications that need to react to external file changes—configuration reloading, data import, file synchronization.

```
<Watch> the <file-monitor> for the <directory> with "./data".
```

Event handlers are named according to the event type: `Handle File Created`, `Handle File Modified`, or `Handle File Deleted`.

Cross-Platform Behavior

File operations work consistently across macOS, Linux, and Windows. Path separators use `/` in ARO code and are translated appropriately for each platform. Hidden files (those starting with `.` on Unix or with the hidden attribute on Windows) are included in listings by default.

16.5 Socket Services

The socket server and client provide low-level TCP communication for applications that need more control than HTTP offers or that need to communicate using custom protocols.

The socket server listens for incoming TCP connections on a specified port. When clients connect and send data, the runtime emits Socket Event events. Your handlers receive the client identifier and the data, and can send responses back to specific clients or broadcast to all connected clients.

The socket client connects to remote TCP servers. You establish a connection, send data, receive responses, and close the connection when done. This is useful for communicating with services that use custom TCP protocols.

Socket communication is lower level than HTTP. You are responsible for message framing, serialization, and protocol handling. The services provide the transport; you provide the protocol logic.

For most web applications, HTTP is the appropriate choice. Use sockets when you need persistent connections, when you need to implement a specific protocol, or when HTTP overhead is unacceptable for your performance requirements.

16.6 Service Lifecycle

Services have a lifecycle that mirrors the application lifecycle. They start during Application-Start, run during the application's lifetime, and stop during Application-End.

Starting services is typically one of the first things you do during startup. You start the HTTP server to begin accepting requests. You start file monitoring to begin watching for changes. You start socket servers to begin accepting connections.

After starting services, you use the Keepalive action to keep the application running. Without it, the application would complete the startup sequence and exit. Keepalive blocks until a shutdown signal arrives, allowing services to process events.

When shutdown occurs, you stop services in your Application-End handler. Stopping in reverse order of starting is a common practice—resources started last are often dependencies of resources started first, so they should be stopped first.

The error shutdown handler (Application-End: Error) should also stop services, attempting best-effort cleanup even when an error has occurred. Services might be in inconsistent states, so cleanup should be defensive.

* * *

16.7 Service Configuration

Services accept configuration options that control their behavior. These options are passed when starting or using the service.

HTTP server configuration includes port and host for network binding, validation for request checking, timeout for request duration limits, body size limits, and CORS settings for browser access control. These options shape how the server behaves and what requests it accepts.

HTTP client configuration includes timeout for response waiting, retry for transient failure recovery, and redirect handling. These options affect outbound request behavior.

File system configuration includes encoding for text handling and directory creation settings. These options affect how files are read and written.

Configuration can be hardcoded in your ARO statements or loaded from external files during startup. Loading from external files allows configuration to vary between environments without code changes.

* * *

16.8 Practical Example: Services in Action

Here is a complete example demonstrating multiple built-in services working together. This application watches a directory for configuration changes and uses the HTTP client to report them to an external monitoring service.

```
(* Config Monitor - Watch files and report changes via HTTP *)

(Application-Start: Config Monitor) {
    <Log> the <message> for the <console> with "Starting configuration monitor...".

    (* Load the monitoring endpoint from environment or config *)
    <Create> the <webhook-url> with "https://monitoring.example.com/webhook".

    (* Start watching the config directory *)
    <Watch> the <file-monitor> for the <directory> with "./config".

    <Log> the <message> for the <console> with "Watching ./config for changes...".

    (* Keep running until shutdown signal *)
    <Keepalive> the <application> for the <events>.

    <Return> an <OK: status> for the <startup>.
}

(Report Config Change: File Event Handler) {
    (* Extract the changed file path *)
    <Extract> the <path> from the <event: path>.
    <Extract> the <event-type> from the <event: type>.

    <Log> the <message> for the <console> with "Config changed:".
    <Log> the <message> for the <console> with <path>.

    (* Build notification payload *)
    <Create> the <notification> with {
        file: <path>,
        change: <event-type>,
        timestamp: "now"
    }.

    (* Send to monitoring webhook *)
    <Send> the <notification> to the <webhook-url>.

    <Log> the <message> for the <console> with "Change reported to monitoring service.".

    <Return> an <OK: status> for the <event>.
}

(Application-End: Success) {
    <Log> the <message> for the <console> with "Config monitor stopped.".
    <Return> an <OK: status> for the <shutdown>.
}
```

This example shows:

- **File system service**: The `Watch` action starts monitoring the `./config` directory
- **HTTP client**: The `Send` action posts change notifications to an external webhook

- **Lifecycle management**: `Keepalive` keeps the app running, `Application-End` provides graceful shutdown
- **Event handling**: The File Event Handler processes each file change event

> **See also:** `Examples/FileWatcher` and `Examples/HTTPClient` for standalone examples of each service.

* * *

16.9 Best Practices

Start all services during Application-Start. Centralizing service startup in one place makes it clear what your application depends on and ensures consistent initialization order.

Stop all services during Application-End. Both success and error handlers should attempt to stop services, releasing resources cleanly. The error handler should be defensive because resources might be in inconsistent states.

Use Keepalive for any application that runs services. Without it, the application starts services and immediately exits. The Keepalive action keeps the application running to process incoming events.

Handle service errors through event handlers when appropriate. Some services emit error events that you can handle to log failures, attempt recovery, or alert operators. The happy path philosophy applies, but observability is still important.

Configure services appropriately for your environment. Development might use permissive CORS settings and verbose logging. Production might use restrictive settings and minimal logging. Load configuration from environment-specific files rather than hardcoding.

* * *

Next: Chapter 17 — Custom Actions

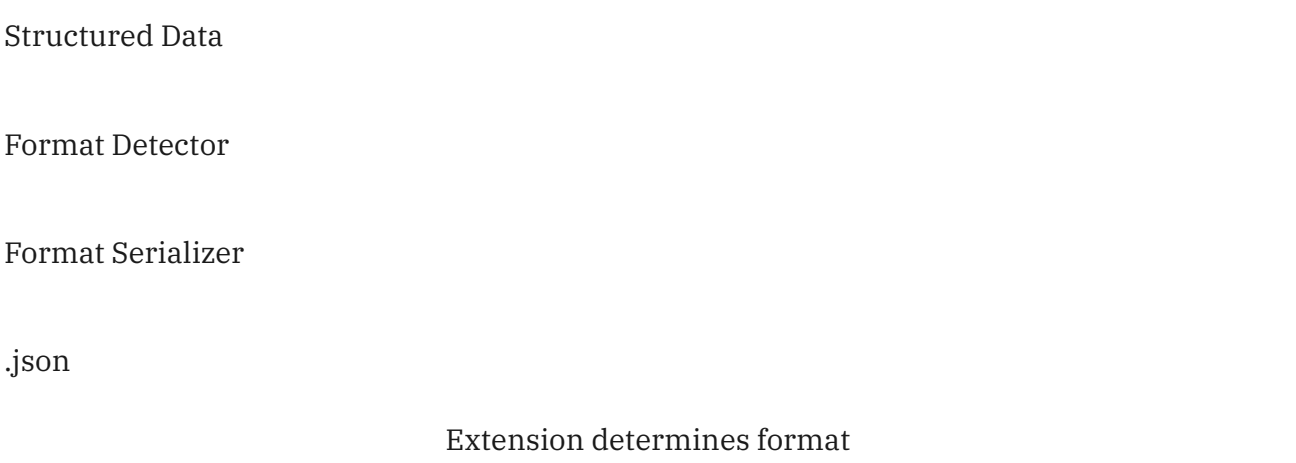
§ Chapter 16B: Format-Aware File I/O

> “Structured data in, structured data out.”

* * *

16B.1 The Format Detection Pattern

When you write data to a file, ARO examines the file extension and automatically serializes your data to the appropriate format. When you read a file, ARO parses the content back into structured data. This eliminates the boilerplate of manual serialization and deserialization.



Extension determines format

The file extension is the key. Write to `users.json` and get JSON. Write to `users.csv` and get CSV. Write to `users.yaml` and get YAML. The same data, formatted appropriately for each destination.

```
(* Same data, three different formats *)
<Write> the <users> to "./output/users.json".
<Write> the <users> to "./output/users.csv".
<Write> the <users> to "./output/users.yaml".
```

This pattern follows ARO's philosophy of reducing ceremony. The file path already tells you the intended format. Making you specify it again would be redundant.

16B.2 Supported Formats

ARO supports thirteen file formats out of the box. Each format has specific characteristics that make it suitable for different use cases.

Extension	Format	Best For
`.json`	JSON	APIs, configuration, data exchange
`.jsonl` , `.ndjson`	JSON Lines	Streaming, logging, large datasets
`.yaml` , `.yml`	YAML	Human-readable configuration
`.toml`	TOML	Application configuration
`.xml`	XML	Legacy systems, enterprise integration
`.csv`	CSV	Spreadsheets, data import/export
`.tsv`	TSV	Tab-delimited data
`.md`	Markdown	Documentation tables
`.html` , `.htm`	HTML	Web output, reports
`.txt`	Plain Text	Simple key-value data
`.sql`	SQL	Database backup, migration
`.log`	Log	Application logs, audit trails
`.env`	Environment	Configuration with uppercase keys
`.obj` or unknown	Binary	Raw data, unknown formats

* * *

16B.3 Writing Structured Data

The Write action serializes your data according to the file extension. Each format has its own serialization rules.

JSON and JSON Lines

JSON is the most common format for structured data. ARO produces pretty-printed JSON with sorted keys for consistency and readability.

```
<Create> the <users> with [
    { "id": 1, "name": "Alice", "email": "alice@example.com" },
    { "id": 2, "name": "Bob", "email": "bob@example.com" }
].

<Write> the <users> to "./output/users.json".
```

Output (users.json):

```
[
  {
    "email": "alice@example.com",
    "id": 1,
    "name": "Alice"
  },
  {
    "email": "bob@example.com",
    "id": 2,
    "name": "Bob"
  }
]
```

JSON Lines (`.jsonl` or `.ndjson`) writes one JSON object per line with no extra whitespace. This format is ideal for streaming and logging because each line is independently parseable.

```
<Write> the <events> to "./logs/events.jsonl".
```

Output (events.jsonl):

```
{"email":"alice@example.com","id":1,"name":"Alice"}
{"email":"bob@example.com","id":2,"name":"Bob"}
```

YAML and TOML

YAML produces human-readable output that is easy to edit by hand. It uses indentation to show structure and avoids the visual noise of brackets and quotes.

```
<Write> the <config> to "./settings.yaml".
```

Output (settings.yaml):

```
-   email: alice@example.com
    id: 1
    name: Alice
-   email: bob@example.com
    id: 2
    name: Bob
```

TOML is similar but uses explicit section headers. It is particularly popular for application configuration files.

```
<Write> the <users> to "./config/users.toml".
```

Output (users.toml):

```
[[users]]
id = 1
name = "Alice"
email = "alice@example.com"

[[users]]
id = 2
name = "Bob"
email = "bob@example.com"
```

CSV and TSV

Comma-separated values (CSV) is the universal format for spreadsheet data. ARO writes a header row with field names, followed by data rows.

```
<Write> the <report> to "./export/report.csv".
```

Output (report.csv):

```
email,id,name
alice@example.com,1,Alice
bob@example.com,2,Bob
```

Tab-separated values (TSV) works the same way but uses tabs instead of commas. This is useful when your data contains commas.

Values containing the delimiter character, quotes, or newlines are automatically escaped with quotes.

XML

XML output uses the variable name from your Write statement as the root element. This provides meaningful document structure without requiring additional configuration.

```
<Write> the <users> to "./data/users.xml".
```

Output (users.xml):

```
<?xml version="1.0" encoding="UTF-8"?>
<users>
  <item>
    <id>1</id>
    <name>Alice</name>
    <email>alice@example.com</email>
  </item>
  <item>
    <id>2</id>
    <name>Bob</name>
    <email>bob@example.com</email>
  </item>
</users>
```

Notice how `<users>` becomes the root element because that was the variable name in the Write statement.

Markdown and HTML

Markdown produces pipe-delimited tables suitable for documentation. HTML produces properly structured tables with thead and tbody elements.

```
<Write> the <summary> to "./docs/summary.md".
<Write> the <report> to "./output/report.html".
```

Markdown output:

```
| id | name | email |
|----|------|-------|
| 1 | Alice | alice@example.com |
| 2 | Bob | bob@example.com |
```

SQL

SQL output produces INSERT statements for database migration or backup. The table name comes from the variable name.

```
<Write> the <users> to "./backup/users.sql".
```

Output (users.sql):

```
INSERT INTO users (id, name, email) VALUES (1, 'Alice', 'alice@example.com');
INSERT INTO users (id, name, email) VALUES (2, 'Bob', 'bob@example.com');
```

String values are properly escaped to prevent SQL injection in the output.

Log

Log output produces date-prefixed entries, ideal for application logs and audit trails. Each entry receives an ISO8601 timestamp.

```
<Write> the <message> to "./app.log" with "Server started".
<Append> the <entry> to "./app.log" with "User logged in".
```

Output (app.log):

```
2025-12-29T10:30:45Z: Server started
2025-12-29T10:30:46Z: User logged in
```

When writing an array, each element becomes a separate log entry:

```
<Create> the <events> with ["Startup complete", "Listening on port 8080"].
<Write> the <events> to "./events.log".
```

Output:

```
2025-12-29T10:30:45Z: Startup complete
2025-12-29T10:30:45Z: Listening on port 8080
```

Environment Files

Environment files use KEY=VALUE format with uppercase keys. Nested objects are flattened with underscore separators.

```
<Create> the <config> with {
    "database": { "host": "localhost", "port": 5432 },
    "apiKey": "secret123"
}.
<Write> the <config> to "./.env".
```

Output (.env):

```
API_KEY=secret123
DATABASE_HOST=localhost
DATABASE_PORT=5432
```

Environment files can also be read back:

```
<Read> the <settings> from "./.env".
(* Returns { "API_KEY": "secret123", "DATABASE_HOST": "localhost", ... } *)
```

Plain Text

Plain text output produces key=value pairs, one per line. Nested objects use dot notation.

```
<Create> the <config> with { "host": "localhost", "port": 8080, "debug": true }.
<Write> the <config> to "./output/config.txt".
```

Output (config.txt):

```
host=localhost
port=8080
debug=true
```

* * *

16B.4 Reading Structured Data

Reading files reverses the process. ARO examines the file extension and parses the content into structured data that you can work with.

.csv

Format Detector

Format Parser

Structured Data

Extension determines parser

```
(* JSON becomes object or array *)
<Read> the <config> from "./settings.json".

(* JSONL becomes array of objects *)
<Read> the <events> from "./logs/events.jsonl".

(* CSV becomes array with headers as keys *)
<Read> the <records> from "./data.csv".

(* YAML becomes object or array *)
<Read> the <settings> from "./config.yaml".
```

Each format parses to an appropriate data structure:

Format	Parses To
JSON	Object or Array
JSON Lines	Array of Objects
YAML	Object or Array
TOML	Object
XML	Object (nested)
CSV	Array of Objects
TSV	Array of Objects
Plain Text	Object
Environment	Object
Binary	Raw Data

Bypassing Format Detection

Sometimes you want to read a file as raw text without parsing. Use the `String` qualifier to bypass format detection:

```
(* Parse JSON to structured data *)
<Read> the <config> from "./settings.json".

(* Read raw JSON as string - no parsing *)
<Read> the <raw-json: String> from "./settings.json".
```

This is useful when you need to inspect the raw content or pass it to an external system without modification.

* * *

16B.5 CSV and TSV Options

CSV and TSV formats support additional configuration options. These options control how data is formatted when writing and how content is parsed when reading.

Option	Type	Default	Description
`delimiter`	String	`,` (CSV) / `\t` (TSV)	Field separator
`header`	Boolean	`true`	Include/expect header row
`quote`	String	`"`	Quote character

Custom Delimiter

Some systems use semicolons or other characters as field separators:

```
<Write> the <data> to "./export.csv" with { delimiter: ";" }.
```

Header Row Control

You can suppress the header row when writing:

```
<Write> the <data> to "./export.csv" with { header: false }.
```

When reading, if your CSV has no header row, specify this so the parser knows to treat the first line as data:

```
<Read> the <data> from "./import.csv" with { header: false }.
```

* * *

16B.6 Round-Trip Patterns

A common pattern is reading data from one format and writing to another. ARO makes this seamless because the data structure is format-independent.

Data Transformation Pipeline

```
(Application-Start: Data Transformer) {
    (* Read from CSV *)
    <Read> the <records> from "./input/data.csv".

    (* Process the data *)
    <Log> the <count> for the <console> with "Processing records...".

    (* Write to multiple formats *)
    <Write> the <records> to "./output/data.json".
    <Write> the <records> to "./output/data.yaml".
    <Write> the <records> to "./output/report.md".

    <Log> the <done> for the <console> with "Transformation complete!".
    <Return> an <OK: status> for the <transform>.
}
```

Multi-Format Export

When you need to provide data in multiple formats for different consumers:

```
(exportData: Export API) {
    <Retrieve> the <users> from the <user-repository>.

    (* Web API consumers get JSON *)
    <Write> the <users> to "./export/users.json".

    (* Analysts get CSV for spreadsheets *)
    <Write> the <users> to "./export/users.csv".

    (* Documentation gets Markdown *)
    <Write> the <users> to "./docs/users.md".

    (* Archive gets SQL for database restore *)
    <Write> the <users> to "./backup/users.sql".

    <Return> an <OK: status> with "Exported to 4 formats".
}
```

* * *

16B.7 Best Practices

Choose the right format for your use case. JSON and YAML are best for configuration and API data. CSV is best for spreadsheet workflows. JSON Lines is best for logging and streaming. SQL is best for database backup.

Use consistent extensions. Stick to standard extensions like `.json`, `.yaml`, `.csv`. Avoid custom extensions unless you need binary format handling.

Consider human readability. YAML and Markdown are easy to read and edit by hand. JSON and XML preserve structure but are harder to edit manually.

Handle round-trips carefully. Some formats lose information during round-trips. Markdown tables become strings when read back. SQL output cannot be parsed back into objects. Plan your data flow accordingly.

Use JSON Lines for large datasets. Regular JSON must be fully loaded into memory. JSON Lines can be processed line by line, making it suitable for large files and streaming scenarios.

* * *

Next: Chapter 17 — Custom Actions

§ Chapter 17: Custom Actions

> "When 24 actions aren't enough, write your own."

* * *

17.1 When to Create Custom Actions

ARO's built-in actions cover common operations, but real applications often need capabilities beyond the standard set. Custom actions extend the language with domain-specific operations while maintaining ARO's declarative style.

Create custom actions when you need to integrate with external services. Database drivers, message queues, third-party APIs, and specialized protocols all require code that cannot be expressed in ARO alone. A custom action wraps the integration logic and exposes it through a clean verb in your ARO code.

Create custom actions when you need complex business logic that does not fit ARO's linear statement model. Algorithms, complex validations, recursive processing, and stateful operations are better expressed in Swift. The custom action handles the complexity; the ARO statement expresses the intent.

Create custom actions when you want to wrap existing Swift or Objective-C libraries. Rich ecosystems of libraries exist for image processing, machine learning, cryptography, and countless other domains. Custom actions bridge these libraries into ARO.

Create custom actions when you want domain-specific operations that make your ARO code more expressive. Instead of multiple generic statements, a single statement with a domain-specific verb can express complex operations clearly.

* * *

17.2 The Escape Hatch Pattern

No constrained language survives contact with reality without an extension mechanism. This is the pattern that made other constrained languages successful: Terraform has providers, Ansible has modules, Make has recipes—and ARO has actions and services.

The contract is clear: ARO gives you rails; you build the track extensions when the rails don't go where you need.

ARO provides two extension mechanisms:

Mechanism	What It Adds	Syntax	Best For
Custom Actions	New verbs	`<Geocode> the <coords> from <addr>.`	Domain-specific operations
Custom Services	External integrations	`<Call> from <postgres: query>`	Systems with multiple methods

Custom actions, covered in this chapter, let you add new verbs to the language. When you implement a custom action, you can write statements like `<Geocode> the <coordinates> from the <address>` that feel native to ARO.

Custom services, covered in Chapter 17B, let you integrate external systems through the `Call` action. Services provide multiple methods under a single service name: `<Call> from <postgres: query>`, `<Call> from <postgres: insert>`.

Plugins, covered in Chapter 18, let you package and share both actions and services with the community.

The rest of this chapter focuses on implementing custom actions—the fundamental building block of ARO's extensibility.

* * *

17.3 The Action Protocol

Every custom action implements the ActionImplementation protocol. This protocol defines the structure that the runtime expects: a role indicating data flow direction, a set of verbs that

trigger the action, valid prepositions that can appear with the action, and an execute method that performs the work.

The role categorizes the action by its data flow direction. REQUEST actions bring data from outside the feature set inward—fetching from APIs, querying databases, reading external state. OWN actions transform data already present in the feature set—computing, validating, reformatting. RESPONSE actions send data out and terminate execution—returning results, throwing errors. EXPORT actions send data out without terminating—storing, emitting events, logging.

The verbs set contains the words that trigger this action. When the parser sees one of these verbs in an action statement, it resolves to this action implementation. Choose verbs that read naturally and describe what the action does. A geocoding action might use “Geocode” as its verb.

The valid prepositions set constrains how the action can be used syntactically. If your action makes sense with “from” (extracting from a source) but not “into” (inserting into a destination), include only “from” in the valid prepositions. This helps catch misuse at parse time.

The execute method does the actual work. It receives descriptors for the result and object from the statement, plus an execution context that provides access to bound values and methods for binding new values.

* * *

17.4 Accessing the Context

The execution context is your interface to the ARO runtime. Through it, you access values bound by previous statements, bind new values for subsequent statements, and interact with the event system.

Getting values retrieves bindings created by earlier statements. The require method returns a value and throws if the binding does not exist—use this for required inputs. The get method returns an optional value—use this for optional inputs where missing values are acceptable.

Setting values creates bindings that subsequent statements can access. The bind method associates a value with a name. You typically bind the result using the identifier from the result descriptor, but you can bind additional values if your action produces multiple outputs.

The object descriptor contains information about the statement's object clause. The identifier is the object name. The qualifier, if present, is the path after the object name. For a statement referencing "user: address", the identifier is "user" and the qualifier is "address".

The event bus is accessible through the context for emitting events. Your action can emit domain events that trigger handlers elsewhere in the application.

* * *

17.5 Implementing an Action

Implementing a custom action follows a consistent pattern. You define a struct that conforms to ActionImplementation, specify the required static properties, and implement the execute method.

Begin by choosing an appropriate role based on what the action does with data. If it pulls data from an external source, use REQUEST. If it transforms existing data, use OWN. If it sends data out and ends execution, use RESPONSE. If it has side effects but allows execution to continue, use EXPORT.

Choose verbs that describe the action clearly. The verb appears in ARO statements, so it should read naturally. For a geocoding service, "Geocode" is clear. For a payment processor, "Charge" or "ProcessPayment" might work.

Specify which prepositions make sense for your action. Think about how the statement will read. "Geocode the coordinates from the address" suggests "from" is appropriate. "Encrypt the ciphertext with the key" suggests "with" is appropriate.

In the execute method, retrieve inputs from the context, perform your operation, bind outputs, and return the primary result. Handle errors by throwing exceptions—the runtime converts these to ARO error messages.

* * *

17.6 Error Handling

Custom actions report errors by throwing Swift exceptions. The runtime catches these exceptions and converts them to ARO error messages that follow the happy path philosophy.

Throw descriptive errors that explain what went wrong. Include relevant values in the error message so users can understand the failure. “Cannot geocode address ‘invalid street’: service returned no results” is more helpful than “Geocoding failed.”

Consider the different failure modes your action might encounter. Network errors, validation failures, resource not found, permission denied—each deserves a distinct error message. Users who see these messages should understand both what happened and, ideally, what they might do about it.

The runtime integrates your errors with its error handling system. HTTP handlers convert errors to appropriate status codes. Logging includes the full error context. The error message you throw becomes part of the user-visible output.

* * *

17.7 Async Operations

Custom actions can be asynchronous, which is essential for I/O operations. The execute method is declared async, so you can await async operations within it.

Network requests, database queries, file operations, and external API calls should all use async/await rather than blocking. This allows the runtime to handle other work while waiting for I/O to complete.

Swift’s async/await syntax integrates naturally. You await async calls, handle their results, and proceed. The ARO runtime manages the execution, ensuring that statement ordering is maintained even with concurrent underlying operations.

Timeouts and cancellation should be considered for long-running operations. If your action might take a long time, consider implementing timeout logic or checking for cancellation signals.

* * *

17.8 Thread Safety

Actions must be thread-safe because the runtime may execute them concurrently. Swift’s Sendable protocol, which ActionImplementation requires, helps enforce this.

Design actions to be stateless when possible. Actions that store state between invocations create potential for race conditions. If state is necessary, use appropriate synchronization mechanisms.

Dependencies like database connection pools or HTTP clients should be thread-safe. Most well-designed Swift libraries provide thread-safe interfaces. If you are uncertain about a dependency's thread safety, consult its documentation.

Avoid mutable shared state. If your action needs configuration, receive it during initialization and store it immutably. If your action needs to accumulate results, use the context's binding mechanism rather than internal state.

* * *

17.9 Registration

Custom actions must be registered with the action registry before they can be used. Registration tells the runtime which action implementation handles which verbs.

Registration typically happens during application initialization, before any ARO code executes. For embedded applications, you register actions in your Swift startup code. For plugin-based actions, the plugin system handles registration.

Registration is straightforward: you call the register method on the shared action registry, passing your action type. The runtime examines the type's static properties to learn its verbs and incorporates it into action resolution.

Once registered, your action's verbs become available in ARO code. Statements using those verbs route to your implementation. The integration is seamless—users of your action need not know whether it is built-in or custom.

* * *

17.10 Best Practices

Single responsibility keeps actions focused and testable. Each action should do one thing well. If an action is doing multiple distinct operations, consider splitting it into multiple actions.

Choose verbs that read naturally in ARO statements. The statement “Geocode the coordinates from the address” should sound like a sentence describing what happens. Avoid generic verbs like “Process” or “Handle” that do not convey meaning.

Provide helpful error messages. When your action fails, users see your error message. Invest effort in making these messages clear and actionable.

Document your actions. Other developers using your actions need to know what verbs are available, what prepositions are valid, what inputs are expected, and what outputs are produced. This documentation might live in code comments, separate documentation files, or both.

Test actions independently. Because actions have a well-defined interface, they are straightforward to unit test. Create mock contexts, invoke the execute method, and verify the results.

* * *

Next: Chapter 17B — Custom Services

§ Chapter 17B: Custom Services

> "When you need to talk to the outside world."

* * *

17B.1 Actions vs Services: The Key Distinction

Before diving into custom services, it is essential to understand how they differ from custom actions. This distinction shapes how you design extensions to ARO.

Custom Actions add new verbs to the language. When you create a custom action, you define a new way of expressing intent. The Geocode action lets you write `<Geocode> the <coordinates> from the <address>`. The verb "Geocode" becomes part of your ARO vocabulary.

Custom Services are external integrations invoked through the `<Call>` action. Services do not add new verbs—they add new capabilities accessible through the existing Call verb. You write `<Call> the <result> from the <postgres: query>` to invoke the postgres service's query method.

The pattern reveals the distinction:

```
(* Custom Action - new verb *)
<Geocode> the <coordinates> from the <address>.

(* Custom Service - Call verb with service:method *)
<Call> the <users> from the <postgres: query> with { sql: "SELECT * FROM users" }.
```

When to Use Each

Use Case	Choose	Why
Domain-specific operation that feels like a language feature	Action	Reads naturally: `<Validate> the <order>`
External system integration (database, queue, API)	Service	Uniform pattern: `<Call> from <service: method>`
Reusable across many projects	Service	Services are more portable
Single, focused operation	Action	Cleaner syntax for one thing
Multiple related operations	Service	One service, many methods

The Design Philosophy

Actions extend the language. Services extend the runtime.

When you add an action, you are saying "this operation is so fundamental to my domain that it deserves its own verb." A payment system might have `<Charge>`, `<Refund>`, and `<Authorize>` as actions because these are core operations that should read as naturally as built-in actions.

When you add a service, you are saying "this external system provides capabilities my application needs." A database driver, message queue client, or cloud storage integration are services because they wrap external systems with multiple operations.

* * *

17B.2 The Service Protocol

Services implement the `AROService` protocol. This protocol defines the contract between your service and the ARO runtime.

```
public protocol AROService: Sendable {
    /// Service name used in ARO code (e.g., "postgres", "redis")
    static var name: String { get }

    /// Initialize the service
    init() throws

    /// Execute a method call
    func call(
        _ method: String,
        args: [String: any Sendable]
    ) async throws -> any Sendable

    /// Cleanup resources (optional)
    func shutdown() async
}
```

The protocol is simpler than ActionImplementation because services have a uniform invocation pattern. All service calls go through the `call` method; the method parameter distinguishes different operations.

17B.3 Implementing a Service

Let us build a PostgreSQL service to illustrate the implementation pattern.

Step 1: Define the Service Structure

```
import PostgresNIO

public struct PostgresService: AROService {
    public static let name = "postgres"

    private let pool: PostgresConnectionPool

    public init() throws {
        // Load configuration from environment
        let config = PostgresConnection.Configuration(
            host: ProcessInfo.processInfo.environment["DB_HOST"] ?? "localhost",
            port: Int(ProcessInfo.processInfo.environment["DB_PORT"] ?? "5432") ?? 5432,
            username: ProcessInfo.processInfo.environment["DB_USER"] ?? "postgres",
            password: ProcessInfo.processInfo.environment["DB_PASSWORD"] ?? "",
            database: ProcessInfo.processInfo.environment["DB_NAME"] ?? "app"
        )

        self.pool = try PostgresConnectionPool(configuration: config)
    }
```

Step 2: Implement the Call Method

The call method dispatches to different operations based on the method parameter:

```
    public func call(
        _ method: String,
        args: [String: any Sendable]
    ) async throws -> any Sendable {
        switch method.lowercased() {
        case "query":
            return try await executeQuery(args)
        case "execute":
            return try await executeStatement(args)
        case "insert":
            return try await executeInsert(args)
        default:
            throw ServiceError.unknownMethod(method, service: Self.name)
        }
    }

    private func executeQuery(_ args: [String: any Sendable]) async throws -> [[String: Any]] {
        guard let sql = args["sql"] as? String else {
            throw ServiceError.missingArgument("sql")
        }

        let rows = try await pool.query(sql)
        return rows.map { row in
            // Convert PostgreSQL row to dictionary
            var dict: [String: Any] = [:]
            for (column, value) in row {
                dict[column] = value
            }
            return dict
        }
    }

    private func executeStatement(_ args: [String: any Sendable]) async throws -> [String: Any] {
        guard let sql = args["sql"] as? String else {
            throw ServiceError.missingArgument("sql")
        }

        try await pool.execute(sql)
        return ["success": true]
    }

    private func executeInsert(_ args: [String: any Sendable]) async throws -> [String: Any] {
        guard let table = args["table"] as? String,
              let data = args["data"] as? [String: Any] else {
            throw ServiceError.missingArgument("table or data")
        }

        let columns = data.keys.sorted().joined(separator: ", ")
        let placeholders = (1...data.count).map { "$\($0)" }.joined(separator: ", ")
        let sql = "INSERT INTO \(table) (\(columns)) VALUES (\(placeholders)) RETURNING id"

        let result = try await pool.query(sql, values: data.values.map { $0 })
```

```
        return ["id": result.first?["id"] ?? 0]
    }
```

Step 3: Implement Shutdown

```
    public func shutdown() async {
        await pool.close()
    }
}
```

17B.4 Using Your Service in ARO

Once registered, your service is available through the Call action:

```
(List Active Users: User Management) {
    <Call> the <users> from the <postgres: query> with {
        sql: "SELECT * FROM users WHERE active = true"
    }.

    <Return> an <OK: status> with <users>.
}

(Create User: User Management) {
    <Extract> the <data> from the <request: body>.

    <Call> the <result> from the <postgres: insert> with {
        table: "users",
        data: <data>
    }.

    <Extract> the <id> from the <result: id>.
    <Return> a <Created: status> with { id: <id> }.
}

(Update User Status: User Management) {
    <Extract> the <id> from the <pathParameters: id>.
    <Extract> the <status> from the <request: status>.

    <Call> the <result> from the <postgres: execute> with {
        sql: "UPDATE users SET status = $1 WHERE id = $2",
        params: [<status>, <id>]
    }.

    <Return> an <OK: status> for the <update>.
}
```

* * *

17B.5 Service Registration

Services must be registered before use. Registration happens during application initialization:

```
// In your Swift application startup
import AORuntime

func initializeServices() throws {
    // Register custom services
    try ServiceRegistry.shared.register(PostgresService())
    try ServiceRegistry.shared.register(RedisService())
    try ServiceRegistry.shared.register(RabbitMQService())
}
```

For plugin-based services (covered in Chapter 18), registration happens automatically when the plugin loads.

* * *

17B.6 Error Handling in Services

Services report errors by throwing Swift exceptions. The runtime converts these to ARO error messages:

```
public enum ServiceError: Error, CustomStringConvertible {
    case unknownMethod(String, service: String)
    case missingArgument(String)
    case connectionFailed(String)
    case queryFailed(String)

    public var description: String {
        switch self {
        case .unknownMethod(let method, let service):
            return "Unknown method '\(method)' for service '\(service)'"
        case .missingArgument(let arg):
            return "Missing required argument: \(arg)"
        case .connectionFailed(let reason):
            return "Connection failed: \(reason)"
        case .queryFailed(let reason):
            return "Query failed: \(reason)"
        }
    }
}
```

When a service throws, the ARO runtime produces an error like:

```
Cannot call postgres:query - Connection failed: Connection refused to localhost:5432
```

* * *

17B.7 Complete Example: Redis Service

Here is a complete service implementation for Redis:

```
import RediStack

public struct RedisService: AROService {
    public static let name = "redis"

    private let connection: RedisConnection

    public init() throws {
        let host = ProcessInfo.processInfo.environment["REDIS_HOST"] ?? "localhost"
        let port = Int(ProcessInfo.processInfo.environment["REDIS_PORT"] ?? "6379") ?? 6379

        self.connection = try RedisConnection.make(
            configuration: .init(hostname: host, port: port)
        ).wait()
    }

    public func call(
        _ method: String,
        args: [String: any Sendable]
    ) async throws -> any Sendable {
        switch method.lowercased() {
        case "get":
            guard let key = args["key"] as? String else {
                throw ServiceError.missingArgument("key")
            }
            let value = try await connection.get(RedisKey(key)).get()
            return value.string ?? ""

        case "set":
            guard let key = args["key"] as? String,
                  let value = args["value"] else {
                throw ServiceError.missingArgument("key or value")
            }
            let stringValue = String(describing: value)
            try await connection.set(RedisKey(key), to: stringValue).get()
            return ["success": true]

        case "del", "delete":
            guard let key = args["key"] as? String else {
                throw ServiceError.missingArgument("key")
            }
            let count = try await connection.delete(RedisKey(key)).get()
            return ["deleted": count]

        case "incr":
            guard let key = args["key"] as? String else {
                throw ServiceError.missingArgument("key")
            }
            let newValue = try await connection.increment(RedisKey(key)).get()
            return ["value": newValue]

        case "expire":
            guard let key = args["key"] as? String,
```

```
                let seconds = args["seconds"] as? Int else {
                throw ServiceError.missingArgument("key or seconds")
            }
            try await connection.expire(RedisKey(key), after: .seconds(Int64(seconds))).get()
            return ["success": true]

        default:
            throw ServiceError.unknownMethod(method, service: Self.name)
        }
    }

    public func shutdown() async {
        try? await connection.close().get()
    }
}
```

Usage in ARO:

```
(Cache User: Caching) {
    <Extract> the <user-id> from the <user: id>.

    <Call> the <result> from the <redis: set> with {
        key: "user:" + <user-id>,
        value: <user>
    }.

    (* Set expiration to 1 hour *)
    <Call> the <expire-result> from the <redis: expire> with {
        key: "user:" + <user-id>,
        seconds: 3600
    }.

    <Return> an <OK: status> for the <cache>.
}

(Get Cached User: Caching) {
    <Extract> the <user-id> from the <request: id>.

    <Call> the <cached> from the <redis: get> with {
        key: "user:" + <user-id>
    }.

    <Return> an <OK: status> with <cached>.
}
```

* * *

17B.8 Best Practices

Name services clearly. The service name appears in ARO code, so it should be short and recognizable. Use “postgres” not “postgresqlDatabaseService”.

Keep methods focused. Each method should do one thing. “query” runs a query. “execute” runs a statement without results. “insert” inserts and returns the ID. Do not combine operations.

Document methods. Users of your service need to know what methods are available, what arguments each expects, and what it returns. Consider adding a “help” method that returns documentation.

Handle connection lifecycle. Services often maintain connections to external systems. Initialize connections in `init()`, reuse them across calls, and close them in `shutdown()`.

Use environment variables for configuration. Host, port, credentials, and other settings should come from environment variables, not hardcoded values. This allows the same code to work in development and production.

Provide meaningful errors. When a method fails, the error message should explain what went wrong and ideally suggest a fix. “Connection refused to localhost:5432” is better than “Connection error”.

* * *

17B.9 Services vs Built-in Actions

Some capabilities in ARO are implemented as built-in actions rather than services. HTTP requests use the `Request` action, not a service:

```
(* Built-in Request action - NOT a service *)
<Request> the <weather> from "https://api.weather.com/current".

(* Service call pattern - for external integrations *)
<Call> the <users> from the <postgres: query> with { sql: "..." }.
```

The distinction: built-in actions are part of the ARO language and have dedicated syntax. Services are external integrations that share the uniform Call pattern.

Capability	Implementation	Syntax
HTTP requests	Built-in action	`<Request> the <data> from <url>.`
File operations	Built-in action	`<Read> the <data> from <path>.`
Database queries	Custom service	`<Call> from <postgres: query>`
Message queues	Custom service	`<Call> from <rabbitmq: publish>`
Cloud storage	Custom service	`<Call> from <s3: upload>`

* * *

Next: Chapter 18 — Plugins

§ Chapter 18: Plugins

> “Package and share your extensions.”

* * *

18.1 What Are Plugins?

Plugins are Swift packages that provide custom actions and services for ARO applications. They allow you to package extensions together, share them across projects, and distribute them to other developers through Swift Package Manager.

The plugin system exists because real applications often need specialized capabilities that the built-in features do not provide. Plugins provide a structured way to create, distribute, and use these extensions.

A plugin can contain two types of extensions:

Type	Adds	Invocation Pattern	Example
Actions	New verbs	`<Verb> the <result> from <object>.`	`<Geocode> the <coords> from <address>.`
Services	External integrations	`<Call> from <service: method>`	`<Call> from <zip: compress>`

The distinction matters for plugin design. Actions extend the language vocabulary—each action adds a new verb. Services extend runtime capabilities—they share the `Call` verb but provide different methods.

* * *

18.2 Plugin Structure

A plugin follows standard Swift package conventions with ARO-specific requirements. The package contains source files, a registration function, and optionally tests and documentation.

The package manifest declares the plugin as a dynamic library product. This enables runtime loading where the plugin is discovered and loaded as a shared library.

For Action Plugins: - Implement the `ActionImplementation` protocol - Registration function calls `ActionRegistry.shared.register(YourAction.self)` - Each action adds a new verb to ARO

For Service Plugins: - Use the C-callable interface with `aro_plugin_init` - Return JSON metadata describing services and their symbols - Each service is called via `<Call> from <service: method>`

* * *

18.3 Creating an Action Plugin

Action plugins add new verbs to ARO. Here is a complete example of a Geocoding action plugin.

Directory Structure:

```
GeocodePlugin/
├── Package.swift
├── Sources/GeocodePlugin/
│   ├── GeocodeAction.swift
│   └── Registration.swift
└── Tests/GeocodePluginTests/
    └── GeocodeActionTests.swift
```

Package.swift:

```
// swift-tools-version:5.9
import PackageDescription

let package = Package(
    name: "GeocodePlugin",
    platforms: [.macOS(.v13)],
    products: [
        .library(name: "GeocodePlugin", type: .dynamic, targets: ["GeocodePlugin"])
    ],
    dependencies: [
        .package(url: "https://github.com/your-org/ARORuntime.git", from: "1.0.0")
    ],
    targets: [
        .target(name: "GeocodePlugin", dependencies: ["ARORuntime"])
    ]
)
```

GeocodeAction.swift:

```
import ARORuntime

public struct GeocodeAction: ActionImplementation {
    public static let role: ActionRole = .request
    public static let verbs: Set<String> = ["Geocode"]
    public static let validPrepositions: Set<Preposition> = [.from]

    public init() {}

    public func execute(
        result: ResultDescriptor,
        object: ObjectDescriptor,
        context: ExecutionContext
    ) async throws -> any Sendable {
        // Get the address from context
        let address: String = try context.require(object.identifier)

        // Call geocoding API (simplified)
        let coordinates = try await geocode(address)

        // Bind result
        context.bind(result.identifier, value: coordinates)

        return coordinates
    }

    private func geocode(_ address: String) async throws -> [String: Double] {
        // Implementation using a geocoding service
        // Returns ["latitude": 37.7749, "longitude": -122.4194]
    }
}
```

Registration.swift:

```
import AARORuntime

@_cdecl("aro_plugin_register")
public func registerPlugin() {
    ActionRegistry.shared.register(GeocodeAction.self)
}
```

Usage in ARO:

```
(Get Location: Address Lookup) {
    <Create> the <address> with "1600 Amphitheatre Parkway, Mountain View, CA".

    (* Custom action - new verb *)
    <Geocode> the <coordinates> from the <address>.

    <Log> the <message> for the <console> with <coordinates>.
    <Return> an <OK: status> with <coordinates>.
}
```

18.4 Creating a Service Plugin

Service plugins provide external integrations called via the `Call` action. Here is a complete example of a Zip service plugin.

Directory Structure:

```
ZipPlugin/
├── Package.swift
└── Sources/ZipPlugin/
    └── ZipService.swift
```

Package.swift:

```
// swift-tools-version:5.9
import PackageDescription

let package = Package(
    name: "ZipPlugin",
    platforms: [.macOS(.v13)],
    products: [
        .library(name: "ZipPlugin", type: .dynamic, targets: ["ZipPlugin"])
    ],
    dependencies: [
        .package(url: "https://github.com/marmelroy/Zip.git", from: "2.1.0")
    ],
    targets: [
        .target(name: "ZipPlugin", dependencies: ["Zip"])
    ]
)
```

ZipService.swift:

```
import Foundation
import Zip

// Plugin initialization - returns service metadata as JSON
@_cdecl("aro_plugin_init")
public func pluginInit() -> UnsafePointer<CChar> {
    let metadata = """
    {"services": [{"name": "zip", "symbol": "zip_call"}]}
    """
    return UnsafePointer(strdup(metadata)!)
}

// Main entry point for the zip service
@_cdecl("zip_call")
public func zipCall(
    _ methodPtr: UnsafePointer<CChar>,
    _ argsPtr: UnsafePointer<CChar>,
    _ resultPtr: UnsafeMutablePointer<UnsafeMutablePointer<CChar>?>
) -> Int32 {
    let method = String(cString: methodPtr)
    let argsJSON = String(cString: argsPtr)

    do {
        let args = try parseJSON(argsJSON)
        let result = try executeMethod(method, args: args)
        resultPtr.pointee = encodeJSON(result).withCString { strdup($0) }
        return 0
    } catch {
        resultPtr.pointee = "{\"error\": \"\(error)\"}".withCString { strdup($0) }
        return 1
    }
}

private func executeMethod(_ method: String, args: [String: Any]) throws -> [String: Any] {
    switch method.lowercased() {
    case "compress", "zip":
        guard let files = args["files"] as? [String],
              let output = args["output"] as? String else {
            throw PluginError.missingArgument
        }
        let fileURLs = files.map { URL(fileURLWithPath: $0) }
        try Zip.zipFiles(paths: fileURLs, zipFilePath: URL(fileURLWithPath: output), password: nil
        return ["success": true, "output": output, "filesCompressed": files.count]

    case "decompress", "unzip":
        guard let archive = args["archive"] as? String else {
            throw PluginError.missingArgument
        }
        let destination = args["destination"] as? String ?? "."
        try Zip.unzipFile(URL(fileURLWithPath: archive), destination: URL(fileURLWithPath: destina
        return ["success": true, "destination": destination]

    default:
```

```
        throw PluginError.unknownMethod(method)
    }
}

enum PluginError: Error {
    case missingArgument, unknownMethod(String)
}
```

Usage in ARO:

```
(Compress Files: Archive) {
    (* Service call - uses Call action *)
    <Call> the <result> from the <zip: compress> with {
        files: ["file1.txt", "file2.txt"],
        output: "archive.zip"
    }.

    <Log> the <message> for the <console> with <result>.
    <Return> an <OK: status> for the <compression>.
}
```

* * *

18.5 Choosing Between Action and Service Plugins

When designing a plugin, consider which approach fits better:

Choose Action Plugin When:

- The operation feels like a language feature
- You want natural syntax: `<Geocode>`, `<Encrypt>`, `<Validate>`
- The operation is single-purpose
- Readability is paramount

```
(* Natural, domain-specific syntax *)
<Geocode> the <coordinates> from the <address>.
<Encrypt> the <ciphertext> from the <plaintext> with <key>.
<Validate> the <result> for the <order>.
```

Choose Service Plugin When:

- You are wrapping an external system with multiple operations
- You want a uniform interface: `<Call> from <service: method>`
- The integration has many related methods
- Portability across projects matters

```
(* Uniform service pattern *)
<Call> the <result> from the <postgres: query> with { sql: "..." }.
<Call> the <result> from the <postgres: insert> with { table: "...", data: ... }.
<Call> the <result> from the <zip: compress> with { files: [...] }.
<Call> the <result> from the <zip: decompress> with { archive: "..." }.
```

Comparison Table

Aspect	Action Plugin	Service Plugin
Syntax	`<Verb> the <result>...`	`<Call> from <service: method>`
Protocol	`ActionImplementation`	C-callable with JSON
Methods	One per action	Multiple per service
Registration	`ActionRegistry.register()`	`aro_plugin_init` JSON
Best for	Domain operations	External integrations

* * *

18.6 Plugin Loading

Plugins load in two ways: compile-time linking and runtime discovery.

Compile-time linking adds the plugin as a package dependency. When you build your application, the plugin is linked in. The registration function runs during initialization.

Runtime discovery scans a `plugins/` directory for compiled libraries. During startup, ARO loads each library and calls its registration function. This allows adding plugins without recompiling.

For runtime loading, place compiled `.dylib` (macOS), `.so` (Linux), or `.dll` (Windows) files in `./plugins/`. ARO loads them automatically during Application-Start.

18.7 Plugin Design Guidelines

Cohesion: Group related functionality. A database plugin provides all database operations. A compression plugin provides all compression methods.

Naming: Use distinctive names. For actions, prefix with your domain if conflicts are possible. For services, choose clear names that describe the integration.

Configuration: Use environment variables for credentials and connection strings. This keeps configuration separate from code and allows different values per environment.

Error messages: Provide clear, actionable errors. Include what failed, why, and ideally what to do about it.

18.8 Documentation

Document your plugin thoroughly:

- **Actions**: List verbs, valid prepositions, expected inputs, outputs, errors
- **Services**: List methods, arguments for each, return values
- **Configuration**: Environment variables, required setup
- **Examples**: Show typical usage in ARO code

A README should provide quick start instructions. Users should be productive within minutes of adding your plugin.

18.9 Publishing

Publish plugins through Git repositories. Swift Package Manager resolves dependencies from URLs.

1. Create a Git repository for your plugin
2. Tag releases following semantic versioning
3. Document installation in your README
4. Announce in relevant communities

Example installation instruction:

```
// In your Package.swift
dependencies: [
    .package(url: "https://github.com/your-org/GeocodePlugin.git", from: "1.0.0")
]
```

* * *

18.10 Best Practices

Test thoroughly. Plugins may be used in unexpected ways. Test edge cases and error conditions.

Version carefully. Breaking changes require major version bumps. Users depend on stability.

Keep dependencies minimal. Each dependency is a dependency for your users. Heavy dependencies cause conflicts.

Document everything. Users should not need to read source code to use your plugin.

* * *

Next: Chapter 19 — Native Compilation

§ Chapter 19: Native Compilation

> "From script to standalone binary."

* * *

19.1 What Is Native Compilation?

ARO can compile applications to standalone native binaries that run without the ARO runtime installed. This transforms your application from an interpreted script into a self-contained executable that can be deployed anywhere the target platform runs.

The native compilation process translates ARO source code into LLVM IR (intermediate representation), compiles it to machine code, and links the result with a precompiled runtime library. The output is a native executable—a binary file that the operating system can run directly without any interpreter.

This capability matters for deployment scenarios. When you deploy an interpreted ARO application, you must ensure the ARO runtime is available on the target system. When you deploy a native binary, you deploy just the binary (and any required data files like openapi.yaml). This simplifies deployment, reduces dependencies, and can improve startup time.

Native binaries also provide some intellectual property protection. While not impossible to reverse engineer, native binaries obscure your application logic more than source code would. For applications where source visibility is a concern, native compilation provides a degree of protection.

* * *

19.2 The Build Command

The aro build command compiles an ARO application to a native binary. You provide the path to your application directory, and the compiler produces an executable.

The basic invocation takes just the application path. The output executable is named after your application directory. If you want a different name, use the output option to specify it explicitly.

Optimization options control how aggressively the compiler optimizes the generated code. The optimize flag enables performance optimizations that may increase compile time but produce faster code. The size flag optimizes for smaller binary size. The strip flag removes debug symbols, reducing binary size but making debugging harder.

The release flag combines optimization, size optimization, and symbol stripping for production builds. This produces the smallest and fastest binaries at the cost of debugging capability. Use release for deployment; use unoptimized builds during development when you might need to debug.

Verbose output shows what the compiler is doing at each step: discovering source files, parsing, generating LLVM IR, compiling to machine code, and linking. This visibility helps diagnose build problems and understand what the compiler does.

* * *

19.3 The Compilation Pipeline

.aro AST validated .ll

*.o MyApp ARO Runtime Library libaro.a

Native compilation proceeds through a series of transformations that convert ARO source code into an executable binary.

The process begins with discovery. The compiler scans your application directory for ARO source files and auxiliary files like the OpenAPI specification. It validates that exactly one Application-Start feature set exists.

Parsing converts source text into abstract syntax trees. Each source file is parsed independently, producing AST representations of the feature sets and statements it contains. Parse errors at this stage indicate syntax problems in your source code.

Semantic analysis validates the parsed code. It checks that referenced variables are defined, that actions are used with valid prepositions, and that data flows correctly through feature sets. Semantic errors indicate logical problems that cannot be detected from syntax alone.

Code generation produces LLVM IR from the validated AST. Each ARO statement becomes one or more calls to the runtime library. The generated code follows a straightforward translation pattern that preserves the semantics of the original ARO code.

Compilation uses the LLVM toolchain to compile the generated IR into object code. Optimization happens at this stage—LLVM can apply its full range of optimizations to the generated code.

Linking combines the object code with the ARO runtime library to produce the final executable. The runtime library contains implementations of all the built-in actions, the event bus, HTTP client and server, and other infrastructure your application depends on.

* * *

19.4 Runtime Requirements

Native binaries link against the ARO runtime library, which provides implementations of actions and services. This library is included in every binary.

The OpenAPI specification file must still be present at runtime for applications that serve HTTP requests. The specification defines routing, and the runtime reads it when the HTTP server starts. Deploy the openapi.yaml file alongside your binary.

Any configuration files or data files your application reads must also be deployed. The native binary does not embed these files; it reads them at runtime just as the interpreted version would.

Dynamically loaded plugins are not supported in native builds. All actions must either be built-in or statically linked at compile time. If you use plugins, they must be included as compile-time dependencies rather than loaded at runtime.

* * *

19.5 Binary Size and Performance

Native binaries have characteristic size and performance profiles that differ from interpreted execution.

Binary size depends on the complexity of your application and whether optimizations are enabled. Release builds with stripping produce the smallest binaries.

Startup time improves significantly with native binaries. Interpreted execution must parse source files and compile them to an internal representation before running. Native binaries skip this phase, starting execution immediately. For applications that start frequently—command-line tools, serverless functions—this improvement is meaningful.

Runtime performance for I/O-bound workloads (most ARO applications) is similar between interpreted and native execution. The bottleneck is usually I/O—network requests, database queries, file operations—not the execution of ARO statements. For compute-heavy workloads, native compilation may provide some improvement.

Memory usage is typically lower for native binaries because they do not maintain the interpreter infrastructure. This can be significant for memory-constrained environments.

* * *

19.6 Deployment

Native binaries simplify deployment because they have minimal runtime dependencies. The binary, the OpenAPI specification (if using HTTP), and any data files are all you need to deploy.

Containerization with Docker works well with native binaries. A multi-stage build can use the full ARO development image for compilation and a minimal base image for the final container. The resulting container contains only the binary and required files, producing small, efficient images.

Systemd and other service managers can run native binaries directly. Create a service unit file that specifies the binary location, working directory, user, and restart behavior. The binary behaves like any other system service.

Cloud deployment to platforms that accept binaries—EC2, GCE, bare metal—is straightforward. Upload the binary and supporting files, configure networking and security,

and run the binary. Platform-specific considerations like health checks and logging integrations apply as they would to any application.

* * *

19.7 Debugging

Debugging native binaries requires different tools than debugging interpreted execution. The runtime's verbose output is not available; instead, you use traditional native debugging tools.

Compile without the strip flag to retain debug symbols. These symbols map binary locations back to source locations, enabling meaningful stack traces and debugger operation.

System debuggers like lldb on macOS and gdb on Linux can attach to your binary, set breakpoints, examine memory, and step through execution. The code you debug is the compiled machine code rather than the original ARO code, but the relationship is straightforward enough to follow.

Core dumps capture the state of a crashed binary for post-mortem analysis. Enable core dumps in your environment, and when a crash occurs, use the debugger to examine the core file and understand what happened.

Logging becomes more important when detailed runtime output is not available. Include logging statements in your ARO code to provide visibility into execution. The logged output is your primary window into what the native binary does during execution.

* * *

19.8 Output Formatting

Native binaries produce cleaner output than interpreted execution. This difference is intentional and reflects the different contexts in which each mode is used.

When running with the interpreter using `aro run`, log messages include a feature set name prefix:

```
[Application-Start] Starting server...
[Application-Start] Server ready on port 8080
[listUsers] Processing request...
```

When running a compiled binary, the same log messages appear without the prefix:

```
Starting server...
Server ready on port 8080
Processing request...
```

The interpreter's prefix identifies which feature set produced each message. This visibility aids debugging during development—when something goes wrong, you can see exactly where messages originated. The prefix becomes unnecessary noise in production, where the focus shifts from debugging to clean operation.

Response formatting remains unchanged between modes. The `[OK]` status prefix and response data appear identically in both cases, providing consistent machine-parseable output for scripts and monitoring tools.

* * *

19.9 Development Workflow

Development typically uses interpreted execution for rapid iteration. The interpreted mode has faster turnaround—you change code and immediately run the updated version without a compile step. Verbose output shows what the runtime does, aiding debugging and understanding.

Native compilation enters the workflow for testing deployment configurations and for final release builds. Testing with native binaries before deployment catches problems that might only appear in the native build, such as missing files or incorrect paths.

Continuous integration should build and test native binaries to ensure they work correctly. The CI pipeline builds the binary, runs tests against it, and produces artifacts for deployment. Catching problems in CI prevents deployment failures.

Release processes should produce native binaries with release optimizations. Tag releases in version control, build the release binary, and archive it alongside release notes and deployment documentation.

* * *

19.10 Limitations

Native compilation has limitations compared to interpreted execution.

Dynamic plugin loading is not supported. Plugins must be compile-time dependencies, statically linked into the binary. This is a significant limitation for applications that depend on runtime-discoverable plugins.

Some runtime reflection capabilities may not be available. Features that depend on examining the structure of running code may behave differently or not work at all in native builds.

Cross-compilation is not currently supported. You build binaries for the platform where you run the compiler. Building for different target platforms requires building on those platforms or using platform emulation.

The compilation step adds time to the development cycle. For rapid iteration, this overhead makes interpreted execution preferable. Native compilation is best reserved for testing and release.

* * *

19.11 Best Practices

Use interpreted mode during development for fast iteration and detailed diagnostics. Switch to native compilation for deployment testing and release.

Test native binaries before deployment. Some problems only appear in native builds—missing files, path issues, platform differences. Running your test suite against the native binary catches these problems early.

Include native binary builds in continuous integration. Automated builds ensure that native compilation continues to work as the codebase evolves.

Use release optimizations for production deployments. The strip, optimize, and size options (or the combined release option) produce the smallest and fastest binaries.

Deploy the OpenAPI specification and other required files alongside the binary. The binary alone is not sufficient for applications that serve HTTP requests.

* * *

Next: Chapter 20 — Multi-file Applications

§ Chapter 20: Multi-file Applications

> "Organize by domain, not by technical layer."

* * *

20.1 Application Structure

An ARO application is a directory containing source files, configuration, and optional auxiliary files. The runtime automatically discovers all ARO source files in this directory and its subdirectories, compiling them together into a unified application.

This directory-based approach differs from many languages where you explicitly import dependencies between files. In ARO, there are no imports. Every feature set in every source file is globally visible within the application. An event emitted in one file can trigger a handler defined in another file without any explicit connection between them.

The automatic discovery means you can organize your source files however makes sense for your project. Group by domain, by feature, by technical concern, or by any other scheme. The runtime does not care about your directory structure—it simply finds all the source files and processes them together.

Certain files have special significance. The openapi.yaml file, if present, defines the HTTP API contract. Plugin directories can contain dynamically loaded extensions. Configuration files might be loaded during startup. But the core source files—the .aro files containing your feature sets—are all treated equally regardless of where they appear in the directory tree.

* * *

20.2 Auto-Discovery

When you run an ARO application, the runtime scans the application directory recursively for files with the .aro extension. Each file is parsed, and its feature sets are registered. This happens automatically without any configuration.

Verbose output shows what the discovery process finds. You can see which files were discovered, how many feature sets each contains, and whether an OpenAPI specification was found. This visibility helps you verify that your application structure matches your expectations.

Subdirectories are fully supported. You can create deep hierarchies if that helps organize your code. A large application might have domain directories, each containing subdirectories for different aspects of that domain. The runtime flattens this hierarchy during discovery—all feature sets end up in the same namespace regardless of which subdirectory they came from.

The discovery process validates structural requirements. It checks that exactly one Application-Start feature set exists across all discovered files. It checks that at most one Application-End: Success and one Application-End: Error exist. Violations of these constraints produce clear error messages indicating which files contain the conflicting definitions.

* * *

20.3 Global Visibility

Feature sets are globally visible without imports. This is a fundamental design choice that enables loose coupling through events.

When you emit an event in one file, handlers in any file can respond to it. The emitting code does not need to know that handlers exist or where they are defined. The handlers do not need to import the emitting code. The connection happens at runtime through the event bus based on naming conventions.

This global visibility means you must ensure feature set names are unique across your entire application. If two files define feature sets with the same name, the runtime reports an error. Choose distinctive names that avoid conflicts.

The same visibility applies to published variables within a business activity. When you use the Publish action to make a value available, it becomes accessible from any feature set with the same business activity that executes afterward. This scoping to business activity enforces modularity—feature sets in different domains cannot accidentally depend on each other's published variables. Use publishing sparingly because shared state complicates reasoning about program behavior.

Repositories are implicitly shared. When one feature set stores data to a repository and another retrieves from the same repository, they are working with shared storage. This is how data flows between feature sets beyond the scope of a single event handling chain.

* * *

20.4 The Application-Start Requirement

Every ARO application must have exactly one Application-Start feature set. This requirement ensures there is an unambiguous entry point for execution.

Having zero Application-Start feature sets is an error. The runtime would have nothing to execute, no way to begin the application's work. The error message explains this clearly and reminds you of the requirement.

Having multiple Application-Start feature sets is also an error. The runtime would not know which one to execute first or whether to execute all of them. The error message lists the locations of all conflicting definitions so you can resolve the conflict.

This constraint applies across all source files in the application. It does not matter which file contains the Application-Start—what matters is that exactly one exists somewhere in the application.

The Application-End feature sets (Success and Error variants) have similar constraints. You can have at most one of each. Having multiple success handlers or multiple error handlers creates ambiguity about shutdown behavior.

* * *

20.5 Organization Strategies

Several strategies for organizing multi-file applications have proven effective in practice.

Organization by domain groups related functionality together. A user management domain might have files for CRUD operations, authentication, and profile management. An order domain might have files for checkout, payment, and fulfillment. Each domain directory contains everything related to that business concept.

Organization by feature groups all code for a specific feature. A product search feature might have its HTTP handlers, event handlers, and utilities in one directory. This organization works well when features are relatively independent and you want to see everything related to a feature in one place.

Organization by technical concern groups code by what it does technically. HTTP handlers go in one directory, event handlers in another, background jobs in a third. This organization works well for teams familiar with layered architectures and can make it easy to find all handlers of a particular type.

Many applications combine these strategies. Domain directories at the top level, with technical concerns separated within each domain. Or feature directories with shared utilities factored out. The right organization depends on your team and your application.

Complete Application Layout Example

Here is a complete e-commerce API organized by domain:

```
ecommerce-api/
├── openapi.yaml              # API contract (required for HTTP server)
├── main.aro                  # Application lifecycle only
├── products/
│   └── products.aro          # Product CRUD operations
├── orders/
│   ├── orders.aro            # Order CRUD operations
│   └── order-events.aro      # Order event handlers
├── inventory/
│   └── inventory.aro         # Stock management
└── notifications/
    └── notifications.aro     # Email/notification handlers
```

main.aro — Lifecycle only, no business logic:

```
(Application-Start: E-commerce API) {
    <Log> the <message> for the <console> with "Starting e-commerce API...".
    <Start> the <http-server> on port 8080.
    <Keepalive> the <application> for the <events>.
    <Return> an <OK: status> for the <startup>.
}

(Application-End: Success) {
    <Stop> the <http-server>.
    <Log> the <message> for the <console> with "E-commerce API stopped.".
    <Return> an <OK: status> for the <shutdown>.
}
```

products/products.aro — Product domain:

```
(listProducts: Product API) {
    <Retrieve> the <products> from the <product-repository>.
    <Return> an <OK: status> with <products>.
}

(getProduct: Product API) {
    <Extract> the <id> from the <pathParameters: id>.
    <Retrieve> the <product> from the <product-repository> where id = <id>.
    <Return> an <OK: status> with <product>.
}
```

orders/orders.aro — Order domain:

```
(createOrder: Order API) {
    <Extract> the <order-data> from the <request: body>.
    <Create> the <order> with <order-data>.
    <Store> the <order> in the <order-repository>.
    <Emit> an <OrderPlaced: event> with <order>.
    <Return> a <Created: status> with <order>.
}
```

orders/order-events.aro — Separated event handlers:

```
(Reserve Stock: OrderPlaced Handler) {
    <Extract> the <order> from the <event: order>.
    <Extract> the <items> from the <order: items>.
    <Update> the <inventory> for <items> with { reserved: true }.
    <Emit> an <StockReserved: event> with <order>.
}
```

notifications/notifications.aro — Cross-domain event handlers:

```
(Send Order Confirmation: OrderPlaced Handler) {
    <Extract> the <order> from the <event: order>.
    <Extract> the <email> from the <order: customerEmail>.
    <Send> the <confirmation-email> to the <email-service> with {
        to: <email>,
        template: "order-confirmation",
        order: <order>
    }.
    <Return> an <OK: status> for the <notification>.
}
```

Key points in this layout: - **main.aro** contains only lifecycle handlers - **Each domain** has its own directory - **Event handlers** are separated from emitters - **Cross-domain handlers** (notifications) have their own location - **No imports needed** — all feature sets are globally visible

> **See also:** `Examples/UserService` for a working multi-file application.

* * *

20.6 Recommended Patterns

Experience suggests some patterns that work well across different types of applications.

Keep the main file minimal. It should contain only lifecycle feature sets—Application-Start and Application-End. Business logic belongs in other files organized by domain or feature. This keeps the entry point focused and easy to understand.

Group related feature sets in the same file. All CRUD handlers for a resource belong together. All handlers for a particular event type might belong together. When you need to understand or modify how something works, you should find all the relevant code in one or a few related files.

Separate event handlers from the feature sets that emit events. The code that creates a user and emits UserCreated belongs in the user domain. The handlers that react to UserCreated might belong in separate files—one for notifications, one for analytics, one for search indexing. This separation allows adding handlers without modifying the emitting code.

Use descriptive file names that indicate content. A file named users.aro clearly contains user-related code. A file named user-events.aro clearly contains event handlers for user events. Readers should be able to navigate your codebase by file names alone.

* * *

20.7 Sharing Data Between Files

Feature sets in different files can share data through several mechanisms, each appropriate for different scenarios.

Events carry data from producers to consumers. When you emit an event with a payload, handlers extract the data they need from that payload. This is the primary mechanism for loosely coupled communication. The producer does not know who consumes the event; consumers do not need to be in the same file as the producer.

Published variables make data available within a business activity. When you publish a value during startup, any feature set with the same business activity that executes afterward can access it. This is appropriate for configuration, constants, and shared state within a domain. Use it sparingly because shared state complicates understanding of program behavior.

Repositories provide persistent shared storage. One feature set stores data; another retrieves it. The repository provides the shared namespace. This is how data persists beyond the lifetime of individual event handling and how different parts of the application work with the same underlying data.

The choice between these mechanisms depends on the relationship between the sharing feature sets. Closely related feature sets reacting to the same event should use event data. Widely separated feature sets needing configuration should use published variables. Feature sets that work with persistent domain entities should use repositories.

* * *

20.8 Best Practices

Establish naming conventions early and follow them consistently. File names, feature set names, event names, and repository names should follow predictable patterns that team members can learn and apply.

Document the intended organization. New team members should understand how files are organized and where new code should go. A brief README or documented convention saves confusion and prevents inconsistent organization.

Keep files focused. A file with fifty feature sets is probably too large. If a file grows unwieldy, split it along natural boundaries—separate CRUD from search, separate handlers from emitters, separate domains from each other.

Consider how changes propagate. When you modify a feature set, what other feature sets might be affected? The event-driven model means effects can be non-obvious. Understanding the event flow through your application helps you understand change impact.

Test files can follow the same organization as production code. Test files for user functionality go alongside or parallel to user functionality source files. This keeps tests easy to find and maintain.

* * *

Next: Chapter 21 — Patterns & Practices

§ Chapter 21: Patterns & Practices

> “Good patterns emerge from solving real problems.”

* * *

21.1 CRUD Patterns

Most business applications need to create, read, update, and delete resources. These CRUD operations follow predictable patterns in ARO that you can apply consistently across your domains.

A complete CRUD service for a resource typically has five feature sets. The list operation retrieves collections with pagination support. The get operation retrieves a single resource by identifier. The create operation makes new resources, stores them, and emits events. The update operation modifies existing resources while preserving consistency. The delete operation removes resources and notifies interested parties.

The list operation extracts pagination parameters from the query string, retrieves matching records with those constraints, computes the total count for pagination metadata, and returns a structured response with both data and pagination information. Clients need the total to know how many pages exist.

The get operation extracts the identifier from the path, retrieves the matching record, and returns it. If no record matches, the runtime generates a not-found error automatically. You do not write explicit handling for the missing case.

The create operation extracts data from the request body, validates it against the resource schema, creates the entity, stores it to the repository, emits a created event for downstream processing, and returns the new entity with a Created status.

The update operation must handle the merge between existing and new data. It extracts the identifier and update data, retrieves the existing record, merges the updates into the existing record, stores the result, emits an updated event, and returns the modified entity.

The delete operation extracts the identifier, deletes the matching record, emits a deleted event, and returns NoContent to indicate successful completion without a response body.

* * *

21.2 Event Sourcing

Event sourcing stores the history of changes rather than current state. Instead of updating a record in place, you append an event describing what happened. Current state is computed by replaying events from the beginning.

This pattern provides a complete audit trail of every change. You can reconstruct state at any point in time by replaying events up to that moment. You can analyze patterns of changes. You can correct errors by appending compensating events rather than modifying history.

In ARO, event sourcing stores events to an event store repository rather than storing entities to a regular repository. Each event includes its type, the aggregate identifier it belongs to, the data describing what happened, and a timestamp.

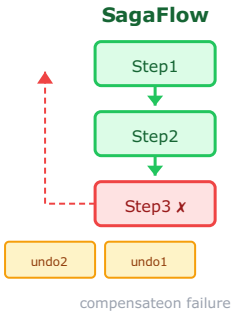
Reading current state requires retrieving all events for an aggregate and reducing them to compute the current state. A custom action can encapsulate this reduction logic, taking a list of events and producing the current state object.

Projections build optimized read models from the event stream. Event handlers listen for events and update read-optimized data structures. This separates the write path (append events) from the read path (query projections), enabling optimization of each independently.

Event sourcing adds complexity compared to simple CRUD. It is most valuable when the audit trail is important, when you need to analyze change patterns, or when you need to support time travel queries. For simpler applications, traditional state-based storage is often sufficient.

* * *

21.3 The Saga Pattern



Sagas coordinate long-running business processes that span multiple steps, each of which might succeed or fail independently. Rather than executing all steps in a single transaction, sagas chain steps through events and provide compensation when steps fail.

A typical saga begins with an initiating request that starts the process. This first step stores initial state, emits an event to trigger the next step, and returns immediately with an accepted status. The caller knows the process has started but not that it has completed.

Each subsequent step is an event handler that performs one piece of the overall process. It receives the event from the previous step, does its work, and emits an event to trigger the next step. Success propagates forward through the event chain.

When a step fails, compensation handlers clean up the effects of previous steps. A PaymentFailed event might trigger a handler that releases reserved inventory. An InventoryUnavailable event might trigger a handler that cancels the pending order. Each compensation reverses one step of the saga.

The saga pattern trades atomicity for availability. Unlike a transaction that succeeds or fails completely, a saga can be partially complete while some steps succeed and others fail. The compensation handlers bring the system to a consistent state, but intermediate states are visible.

Use sagas when you need to coordinate actions across multiple services or when steps take significant time. For quick operations that can complete atomically, simpler patterns are appropriate.

* * *

21.4 The Gateway Pattern

API gateways aggregate data from multiple backend services into unified responses. Rather than having clients make multiple calls and combine results, the gateway handles this coordination.

A gateway handler extracts request parameters, makes parallel or sequential calls to backend services, combines the results into a unified response, and returns it. The client sees a single endpoint that provides rich, aggregated data.

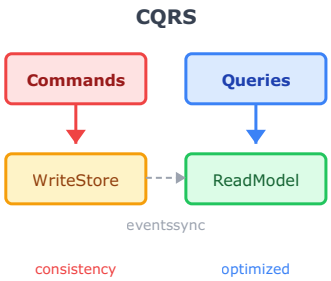
The pattern is valuable when clients need data from multiple sources. A product details page might need product information from the catalog service, stock levels from the inventory service, reviews from the review service, and pricing from the pricing service. A gateway endpoint fetches all of this and returns a complete product details object.

Gateway handlers should be designed for resilience. Backend services might be slow or unavailable. Consider timeout handling, fallback data for unavailable services, and partial responses when some data cannot be retrieved.

The gateway pattern can also handle cross-cutting concerns like authentication, rate limiting, and logging that apply across all backend calls. The gateway becomes a single enforcement point for these concerns.

* * *

21.5 Command Query Responsibility Segregation



CQRS separates read operations from write operations, using different models optimized for each. Commands (writes) update the authoritative data store. Queries (reads) retrieve data from optimized read models.

The write side receives commands, validates them, updates the authoritative store, and emits events describing what changed. The focus is on maintaining consistency and enforcing business rules.

The read side maintains projections optimized for query patterns. When events occur, handlers update these projections. The projections might denormalize data, pre-compute aggregates, or organize data for specific query patterns.

This separation allows independent optimization. The write side can use a normalized relational database that enforces consistency. The read side can use denormalized document stores, search indices, or caching layers that optimize for specific query patterns.

CQRS adds complexity because you maintain multiple representations of data that must stay synchronized. It is most valuable when read and write patterns differ significantly—when you

need rich queries that do not match your write model, or when read load vastly exceeds write load and you need to scale them independently.

* * *

21.6 Error Handling Patterns

ARO's happy path philosophy means you do not write explicit error handling, but you can still respond to errors through event-driven patterns.

Error events can be emitted by custom actions when failures occur. A PaymentFailed event carries information about what went wrong. Handlers for error events can log details, notify administrators, trigger compensating actions, or update status to reflect the failure.

Retry patterns can be implemented in custom actions that wrap unreliable operations. The action attempts the operation, retries on transient failures with backoff, and eventually either succeeds or emits a failure event. The ARO code sees only success or a well-defined failure event.

Dead letter handling captures messages that fail repeatedly. After a configured number of retries, the message goes to a dead letter queue where it can be examined, corrected, and replayed. This prevents poison messages from blocking processing while preserving them for investigation.

Circuit breaker patterns can protect against cascading failures when backend services are unavailable. A custom action tracks failure rates and stops making calls when failures exceed a threshold, returning a fallback response instead. This prevents overwhelming already struggling services.

* * *

21.7 Security Patterns

Authentication verifies caller identity. Security-sensitive endpoints extract authentication tokens, validate them against an authentication service, and extract identity claims. Subsequent operations use the validated identity.

Authorization verifies caller permissions. After authentication establishes who the caller is, authorization checks what they can do. This might involve checking role membership, querying a permissions service, or evaluating policy rules.

Rate limiting prevents abuse by limiting request rates per client. A rate limiting action checks whether the current request exceeds limits and fails if so. This protects against denial of service and ensures fair resource allocation.

Input validation prevents injection and other attacks. Validating request data against schemas catches malformed input before it can cause harm. Custom validation actions can implement domain-specific security rules.

These patterns compose together. A secured endpoint might extract and validate a token, check rate limits, validate input, and only then proceed with business logic. Each layer provides defense in depth.

* * *

21.8 Performance Patterns

Caching reduces load on backend services by storing frequently accessed data. A cache-aware handler first checks the cache. On cache hit, it returns immediately. On cache miss, it fetches from the source, stores in the cache, and returns. Time-to-live settings control cache freshness.

Batch processing handles multiple items efficiently. Rather than processing items one at a time, a batch handler receives a collection and processes them together. This can reduce round trips to backend services and enable bulk operations that are more efficient than individual operations.

Parallel processing handles independent operations concurrently. When multiple pieces of data are needed and they do not depend on each other, fetching them in parallel reduces total latency compared to sequential fetching.

Connection pooling maintains reusable connections to backend services. Rather than establishing a new connection for each request, handlers borrow connections from a pool and return them when done. This amortizes connection setup cost across many requests.

Pagination prevents unbounded result sets. List operations return limited pages of results with metadata indicating total count and how to fetch additional pages. This prevents memory exhaustion from large result sets and provides consistent response times.

* * *

21.9 Best Practices Summary

These practices have emerged from experience building ARO applications and reflect lessons learned about what works well.

Keep feature sets focused on single responsibilities. A feature set that does one thing well is easier to understand, test, and maintain than one that does many things.

Use events for side effects and communication. Rather than calling between feature sets, emit events and let handlers react. This decoupling makes the system more flexible and easier to evolve.

Organize code by domain rather than technical layer. Put user-related code together, not all HTTP handlers together. This makes it easier to understand and modify domain functionality.

Leverage the happy path philosophy. Trust the runtime to handle errors. Focus your code on what should happen when things work correctly.

Use meaningful names because they become part of error messages. Good names make errors understandable without consulting source code.

Keep Application-Start minimal. It should start services and set up the environment, not implement business logic.

Publish variables sparingly. Shared state complicates reasoning about program behavior. Prefer events and repositories for sharing data.

Design for idempotency. Events might be delivered more than once. Handlers that can safely process duplicates are more resilient than those that cannot.

Test at multiple levels. Unit test individual feature sets. Integration test event flows. End-to-end test complete scenarios.

Document the non-obvious. Code should be self-documenting for basic behavior. Comments should explain why, not what—the reasons behind non-obvious choices.

* * *

Next: Chapter 22 — State Machines

§ Chapter 22: State Machines

> “A business object is only as valid as its current state.”

* * *

22.1 Why State Machines Matter

Business entities rarely exist in a single state. An order is drafted, placed, paid, shipped, and delivered. A support ticket is opened, assigned, escalated, and resolved. A document is drafted, reviewed, approved, and published. These lifecycles define what operations are valid at each point in time.

Without explicit state management, code becomes defensive. Developers scatter validation checks throughout the codebase: “Is this order paid?”, “Can we ship this?”, “Has this been approved?” These checks duplicate business rules, create inconsistencies, and make the intended lifecycle invisible.

State machines make lifecycles explicit. They define what states exist, what transitions between states are valid, and what happens when an invalid transition is attempted. The state machine becomes a source of truth for the entity’s lifecycle.

* * *

22.2 ARO’s Approach to State

ARO takes a deliberately simple approach to state machines. There is no state machine library, no visual editor, no hierarchical states. Instead, ARO provides a single action—`Accept`—that validates and applies state transitions within your existing feature sets.

This simplicity is intentional. Most business logic needs straightforward linear or branching state flows, not the full complexity of hierarchical state machines with parallel regions and history states. By using a simple primitive, ARO keeps the language small while enabling the patterns that matter most.

States in ARO are defined as OpenAPI enum types. This means your API contract documents the valid states for each entity. Clients can see what states exist. Tooling can validate state values. The contract becomes the single source of truth.

* * *

22.3 Defining States in OpenAPI

States are defined as string enums in your OpenAPI specification. Here is how an order lifecycle might be defined:

```
components:
  schemas:
    OrderStatus:
      type: string
      description: Valid order states in the lifecycle
      enum:
        - draft
        - placed
        - paid
        - shipped
        - delivered
        - cancelled
```

The enum values define all possible states. The order in the enum often reflects the expected progression, though this is convention rather than enforcement. An order typically flows from draft through placed, paid, shipped, to delivered—but the enum itself does not encode these transitions.

The entity schema references this status type:

```
    Order:
      type: object
      properties:
        id:
          type: string
        status:
          $ref: '#/components/schemas/OrderStatus'
        customerId:
          type: string
        items:
          type: array
          items:
            $ref: '#/components/schemas/OrderItem'
```

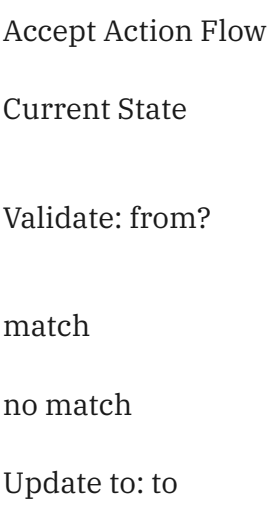
This creates a clear contract: orders have a status field that must be one of the defined enum values. The API documentation shows clients exactly what states exist.

* * *

22.4 The Accept Action

The `Accept` action validates and applies state transitions. It checks that an object's current state matches an expected "from" state, and if so, updates it to a "to" state. If the current state does not match, execution stops with a descriptive error.

The syntax uses the `_to_` separator to specify the transition:



> Throw Error
>
> continue

```
<Accept> the <transition: from_to_target> on <object: field>.
```

The transition specifier contains both states separated by `_to_`. The object specifier identifies which entity and which field to examine and update.

Here is a concrete example that transitions an order from draft to placed:

```
<Accept> the <transition: draft_to_placed> on <order: status>.
```

This statement does three things:

1. Retrieves the current value of `order.status`
2. Validates that the current value equals `"draft"`
3. Updates `order.status` to `"placed"`

If the order's status is not `"draft"`, execution stops with an error. The caller receives a message explaining what went wrong.

22.5 State Transition Validation

The Accept action provides clear error messages when transitions fail. If you attempt to place an order that has already been paid:

```
Cannot accept state draft->placed on order: status. Current state is "paid".
```

This message tells you exactly what happened: the transition expected the order to be in draft state, but it was actually in paid state. No stack traces, no cryptic error codes—just a clear statement of the business rule violation.

This follows ARO's happy path philosophy. You write code for the expected flow. The runtime handles the error cases automatically, generating messages that describe the problem in business terms.

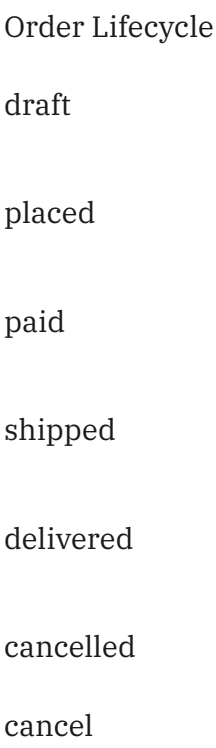
Consider what this means for debugging. When a user reports “I can’t place my order,” you know immediately that the order is not in draft state. The error message tells you the actual state. You can investigate why the order reached that state without the transition being attempted first.

* * *

22.6 Complete Example: Order Lifecycle

Let us walk through a complete order management service that uses state transitions. Orders flow through a lifecycle: they start as drafts, get placed, are paid for, ship, and finally arrive. At any point before placement, an order can be cancelled.

First, the OpenAPI contract defines the states and the operations that trigger transitions:



```
paths:
  /orders/{id}/place:
    post:
      operationId: placeOrder
      summary: Place a draft order
      description: Transitions from draft to placed

  /orders/{id}/pay:
    post:
      operationId: payOrder
      summary: Record payment
      description: Transitions from placed to paid

  /orders/{id}/ship:
    post:
      operationId: shipOrder
      summary: Ship the order
      description: Transitions from paid to shipped

  /orders/{id}/deliver:
    post:
      operationId: deliverOrder
      summary: Mark as delivered
      description: Transitions from shipped to delivered

  /orders/{id}/cancel:
    post:
      operationId: cancelOrder
      summary: Cancel an order
      description: Only valid from draft state
```

Each endpoint documents which state transition it performs. This makes the API self-documenting—clients can see the lifecycle from the contract.

Now the ARO feature sets that implement these transitions:

```
(* Create a new order in draft state *)
(createOrder: Order Management) {
    <Extract> the <data> from the <request: body>.
    <Create> the <order: Order> with <data>.
    <Update> the <order: status> with "draft".
    <Store> the <order> into the <order-repository>.
    <Return> a <Created: status> with <order>.
}

(* Place an order - transitions from draft to placed *)
(placeOrder: Order Management) {
    <Extract> the <order-id> from the <pathParameters: id>.
    <Retrieve> the <order> from the <order-repository>
        where <id> is <order-id>.

    <Accept> the <transition: draft_to_placed> on <order: status>.

    <Store> the <order> into the <order-repository>.
    <Return> an <OK: status> with <order>.
}
```

The `createOrder` feature set explicitly sets the initial state to “draft”. This makes the starting point clear in the code. The `placeOrder` feature set uses `Accept` to validate the transition before updating storage.

The remaining transitions follow the same pattern:

```
(* Pay for an order - transitions from placed to paid *)
(payOrder: Order Management) {
    <Extract> the <order-id> from the <pathParameters: id>.
    <Extract> the <payment> from the <request: body>.
    <Retrieve> the <order> from the <order-repository>
        where <id> is <order-id>.

    <Accept> the <transition: placed_to_paid> on <order: status>.

    <Update> the <order: paymentMethod> from the <payment: paymentMethod>.
    <Store> the <order> into the <order-repository>.
    <Return> an <OK: status> with <order>.
}

(* Ship an order - transitions from paid to shipped *)
(shipOrder: Order Management) {
    <Extract> the <order-id> from the <pathParameters: id>.
    <Extract> the <shipping> from the <request: body>.
    <Retrieve> the <order> from the <order-repository>
        where <id> is <order-id>.

    <Accept> the <transition: paid_to_shipped> on <order: status>.

    <Update> the <order: trackingNumber>
        from the <shipping: trackingNumber>.
    <Store> the <order> into the <order-repository>.
    <Return> an <OK: status> with <order>.
}

(* Deliver an order - transitions from shipped to delivered *)
(deliverOrder: Order Management) {
    <Extract> the <order-id> from the <pathParameters: id>.
    <Retrieve> the <order> from the <order-repository>
        where <id> is <order-id>.

    <Accept> the <transition: shipped_to_delivered> on <order: status>.

    <Store> the <order> into the <order-repository>.
    <Return> an <OK: status> with <order>.
}
```

Each feature set retrieves the order, validates and applies its transition, stores the result, and returns. The business logic is clear: you can pay a placed order, ship a paid order, and deliver a shipped order. Any attempt to skip steps fails with a descriptive error.

* * *

22.7 Cancellation Paths

Real business processes have more than one terminal state. Orders can be cancelled as well as delivered. The state machine must define which transitions lead to cancellation.

```
(* Cancel an order - only valid from draft state *)
(cancelOrder: Order Management) {
    <Extract> the <order-id> from the <pathParameters: id>.
    <Retrieve> the <order> from the <order-repository>
        where <id> is <order-id>.

    <Accept> the <transition: draft_to_cancelled> on <order: status>.

    <Store> the <order> into the <order-repository>.
    <Return> an <OK: status> with <order>.
}
```

This implementation only allows cancellation from the draft state. If you need to allow cancellation from multiple states, you have options.

One approach is multiple feature sets with different transitions:

```
(* Cancel from draft *)
(cancelDraftOrder: Order Management) {
    (* ... *)
    <Accept> the <transition: draft_to_cancelled> on <order: status>.
    (* ... *)
}

(* Cancel from placed - might require refund logic *)
(cancelPlacedOrder: Order Management) {
    (* ... *)
    <Accept> the <transition: placed_to_cancelled> on <order: status>.
    <Emit> a <RefundRequired: event> with <order>.
    (* ... *)
}
```

Another approach is conditional logic using standard control flow:

```
(cancelOrder: Order Management) {
    <Extract> the <order-id> from the <pathParameters: id>.
    <Retrieve> the <order> from the <order-repository>
        where <id> is <order-id>.

    (* Check current state and apply appropriate transition *)
    match <order: status> {
        case "draft" {
            <Accept> the <transition: draft_to_cancelled> on <order: status>.
        }
        case "placed" {
            <Accept> the <transition: placed_to_cancelled> on <order: status>.
            <Emit> a <RefundRequired: event> with <order>.
        }
    }

    <Store> the <order> into the <order-repository>.
    <Return> an <OK: status> with <order>.
}
```

The choice depends on whether the different cancellation paths have different side effects. If cancelling a placed order requires a refund but cancelling a draft does not, separate feature sets make the distinction clear.

* * *

22.8 What ARO Does Not Do

ARO's state machine approach is deliberately minimal. Understanding what it does not do helps you decide when simpler approaches suffice and when you need external tooling.

No hierarchical states. States are flat strings. If you need a state like "processing.validating" or "processing.charging", you would model these as separate states or use a separate field for the sub-state.

No parallel states. An entity has one state at a time. If you need to track multiple concurrent aspects—order fulfillment state and payment state separately—use multiple state fields, each with its own transitions.

No entry or exit actions. The Accept action does not automatically trigger code when entering or leaving a state. If you need side effects on state change, add explicit statements after the Accept action or emit events that handlers process.

No state machine visualization. ARO does not generate state diagrams from your code. The OpenAPI contract with its enum and operation descriptions serves as documentation. For visual diagrams, use external tools with your OpenAPI spec as input.

No automatic validation against the enum. The Accept action validates that the current state matches the expected "from" state. It does not validate that both "from" and "to" are members of the OpenAPI enum. That validation happens when you store the entity or when an HTTP response is serialized.

These limitations are features, not bugs. They keep the language simple. For most business applications, flat states with explicit transitions are sufficient. When you need the full power of statecharts, integrate a dedicated state machine library through custom actions.

* * *

22.9 Combining States with Events

State transitions naturally pair with events. When an order moves to a new state, other parts of the system often need to react. Shipping needs to know when orders are paid. Notifications need to know when orders are delivered.

```
(shipOrder: Order Management) {
    <Extract> the <order-id> from the <pathParameters: id>.
    <Retrieve> the <order> from the <order-repository>
        where <id> is <order-id>.

    <Accept> the <transition: paid_to_shipped> on <order: status>.

    <Store> the <order> into the <order-repository>.

    (* Notify interested parties *)
    <Emit> an <OrderShipped: event> with <order>.

    <Return> an <OK: status> with <order>.
}

(* Handler reacts to the state change *)
(Notify Customer: OrderShipped Handler) {
    <Extract> the <order> from the <event: order>.
    <Extract> the <customer-email> from the <order: customerEmail>.
    <Extract> the <tracking> from the <order: trackingNumber>.

    <Send> the <shipping-notification: email> to <customer-email>
        with <tracking>.

    <Return> an <OK: status> for the <notification>.
}
```

The feature set that changes state is responsible for emitting events. Handlers subscribe to those events and perform side effects. This separation keeps the state transition code focused on the transition itself while allowing arbitrary reactions through the event system.

This pattern also supports saga-style workflows where a state change in one entity triggers state changes in others, each with its own validation.

22.9.1 State-Guarded Handlers

Sometimes you want handlers to only execute when an entity is in a specific state. Rather than checking the state inside the handler, you can filter events at the handler definition using state guards:

```
(* Only process orders that are in "paid" state *)
(Process Paid Order: OrderUpdated Handler<status:paid>) {
    <Extract> the <order> from the <event: order>.
    (* This handler only runs when order.status = "paid" *)
    <Process> the <fulfillment> for the <order>.
    <Return> an <OK: status> for the <processing>.
}
```

State guards use angle bracket syntax after "Handler":

- `<field:value>` - Match when field equals value
- `<field:value1,value2>` - Match when field equals any value (OR logic)
- `<field1:value;field2:value>` - Match when both conditions are true (AND logic)

```
(* Handle orders that are paid OR shipped *)
(Track Fulfillment: OrderUpdated Handler<status:paid,shipped>) {
    <Extract> the <order> from the <event: order>.
    <Log> the <message> for the <console> with "Tracking update".
    <Return> an <OK: status> for the <tracking>.
}

(* Only premium customers with delivered orders *)
(VIP Reward: OrderUpdated Handler<status:delivered;tier:premium>) {
    <Extract> the <order> from the <event: order>.
    <Send> the <reward> to the <order: email>.
    <Return> an <OK: status> for the <reward>.
}
```

State guards enable guaranteed ordering through state-based filtering. Instead of relying on event order, you can ensure handlers only execute when the entity has reached a specific state.

* * *

22.10 Best Practices

Define all states in OpenAPI. The contract should be the source of truth for what states exist. Clients and tooling can use this information.

Use explicit initial states. When creating entities, set their initial state explicitly rather than relying on defaults. This makes the starting point visible in the code.

Document transitions in OpenAPI descriptions. Each endpoint that changes state should document which transition it performs. The `description` field is a good place for this.

Keep states coarse-grained. Prefer fewer states with clear business meaning over many fine-grained states. "Processing" is often better than "validating", "charging", "confirming" as separate states unless those distinctions matter to callers.

Emit events on state changes. Other parts of the system often need to know when state changes. Emit events after successful transitions to enable loose coupling.

Handle the cancelled state carefully. Cancellation often needs cleanup—refunds, inventory release, notification. Use events to trigger this cleanup from handlers rather than cramming it into the cancellation feature set.

Test transition failures. Write tests that attempt invalid transitions and verify the correct error is returned. This documents the state machine constraints and catches regressions.

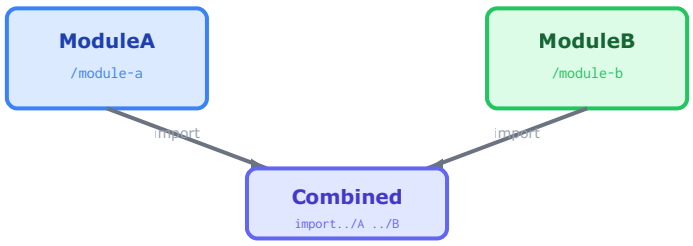
Next: Chapter 23 — Modules and Imports

§ Chapter 23: Modules and Imports

> “Compose applications from small, understandable pieces.”

* * *

23.1 Application Composition



ARO applications are directories containing `.aro` files. When an application grows or when you want to share functionality between projects, the import system lets you compose applications from smaller pieces.

The import mechanism is radically simple. You import another application directory, and all its feature sets become accessible. No visibility modifiers, no selective imports, no namespacing. If you import an application, you trust it and want all of it.

This design reflects how project managers think about systems. Applications are black boxes that do things. If you need what another application provides, you import it. There are no access control decisions to make, no visibility declarations to configure.

* * *

23.2 Import Syntax

The import statement appears at the top of an ARO file, before any feature sets:

```
import ../auth-service
import ../payment-gateway
import ../../shared/utilities
```

Paths are relative to the current file's directory. The `..` notation moves up one directory level. The path points to a directory containing `.aro` files.

When the compiler encounters an import:

1. It resolves the path relative to the current file
2. It finds all `.aro` files in that directory
3. It makes all feature sets from those files accessible
4. Published variables become available
5. Types become available

There is no need to specify what you want from the imported application. Everything becomes accessible.

* * *

23.3 No Visibility Modifiers

ARO explicitly rejects visibility modifiers. There is no `public`, `private`, or `internal`.

Traditional	ARO Approach
`public`	Everything is accessible after import
`private`	Feature set scope handles encapsulation
`internal`	Not needed
`protected`	No inheritance hierarchy

This might seem dangerous. What about encapsulation? What about hiding implementation details?

ARO takes a different position. Within a feature set, variables are scoped naturally. They exist only within that feature set unless explicitly published. If you want to share data between

feature sets, you use the Publish action or emit events. These are explicit sharing mechanisms.

When you import an application, you are saying: I want this application's capabilities. You trust the imported code. If you need to restrict what is accessible, the answer is not visibility modifiers. The answer is to factor the code into appropriate applications.

23.4 The ModulesExample

The `Examples/ModulesExample` directory demonstrates application composition with three directories:

```
ModulesExample/
├── ModuleA/
│   ├── main.aro
│   └── openapi.yaml
├── ModuleB/
│   ├── main.aro
│   └── openapi.yaml
└── Combined/
    ├── main.aro
    └── openapi.yaml
```

Each module can run standalone or be imported into a larger application.

Module A

Module A provides a single endpoint at `/module-a`:

```
(* Module A - Standalone Application *)

(Application-Start: ModuleA) {
    <Log> the <startup: message> for the <console> with "Module A starting...".
    <Start> the <http-server> for the <contract>.
    <Keepalive> the <application> for the <events>.
    <Return> an <OK: status> for the <startup>.
}

(getModuleA: ModuleA API) {
    <Create> the <response> with { message: "Hello from Module A" }.
    <Return> an <OK: status> with <response>.
}
```

Source: Examples/ModulesExample/ModuleA/main.aro

Run it standalone:

```
aro build ./Examples/ModulesExample/ModuleA
./Examples/ModulesExample/ModuleA/ModuleA
```

Module B

Module B provides a single endpoint at `/module-b`:

```
(* Module B - Standalone Application *)

(Application-Start: ModuleB) {
    <Log> the <startup: message> for the <console> with "Module B starting...".
    <Start> the <http-server> for the <contract>.
    <Keepalive> the <application> for the <events>.
    <Return> an <OK: status> for the <startup>.
}

(getModuleB: ModuleB API) {
    <Create> the <response> with { message: "Hello from Module B" }.
    <Return> an <OK: status> with <response>.
}
```

Source: Examples/ModulesExample/ModuleB/main.aro

Combined Application

The Combined application imports both modules and provides both endpoints:

```
(* Combined Application *)

import ../ModuleA
import ../ModuleB

(Application-Start: Combined) {
    <Log> the <startup: message> for the <console> with "Combined application starting...".
    <Start> the <http-server> for the <contract>.
    <Keepalive> the <application> for the <events>.
    <Return> an <OK: status> for the <startup>.
}
```

Source: Examples/ModulesExample/Combined/main.aro

The `getModuleA` and `getModuleB` feature sets come from the imported applications. They do not need to be redefined. The Combined application's OpenAPI contract defines both routes, and the imported feature sets handle them.

* * *

23.5 Building Standalone Binaries

Each module produces its own standalone binary:

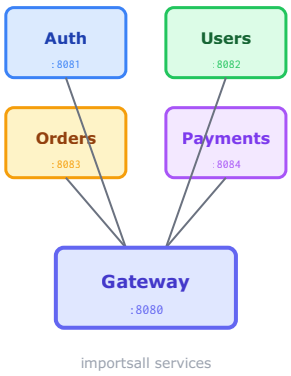
```
# Build Module A
aro build ./Examples/ModulesExample/ModuleA
# Creates: ModuleA/ModuleA

# Build Module B
aro build ./Examples/ModulesExample/ModuleB
# Creates: ModuleB/ModuleB

# Build Combined
aro build ./Examples/ModulesExample/Combined
# Creates: Combined/Combined
```

The Combined binary includes all code from both imported modules. The resulting binary is self-contained and requires no runtime dependencies.

* * *

23.6 Distributed Services Pattern

A common pattern is building microservices that can run independently or be composed into a monolith for simpler deployments:




```
services/
├── auth/
│   ├── main.aro
│   └── openapi.yaml
├── users/
│   ├── main.aro
│   └── openapi.yaml
├── orders/
│   ├── main.aro
│   └── openapi.yaml
└── gateway/
    ├── main.aro
    └── openapi.yaml
```

Each service has its own Application-Start and can run on its own port. The gateway imports all services and provides a unified API.

For development, you might run the gateway monolith. For production, you might run each service independently and use a real API gateway.

* * *

23.7 Sharing Data Between Applications

When you import an application, you get access to its published variables within the same business activity. The Publish action (see ARO-0003) makes values available to feature sets sharing that business activity:

```
(* In auth-service/auth.aro *)
(Authenticate User: Security) {
    <Extract> the <credentials> from the <request: body>.
    <Retrieve> the <user> from the <user-repository> where credentials = <credentials>.
    <Publish> as <authenticated-user> <user>.
    <Return> an <OK: status> with <user>.
}
```

After importing auth-service, other feature sets can access the published variable:

```
(* In gateway/main.aro *)
import ../auth-service

(Process Request: Gateway) {
    (* Access published variable from imported application *)
    <Use> the <authenticated-user> in the <authorization-check>.
    <Return> an <OK: status> for the <request>.
}
```

* * *

23.8 What Is Not Imported

When you import an application, its Application-Start feature set is not executed. Only the importing application's Application-Start runs. The imported feature sets become available, but lifecycle management remains with the importing application.

This prevents conflicts when composing applications. Each composed application might have its own startup logic, but only the top-level application controls the actual startup sequence.

Similarly, Application-End handlers from imported applications are not triggered during shutdown. The importing application manages its own lifecycle.

* * *

23.9 Circular Imports

Circular imports are technically allowed:

```
(* service-a/main.aro *)
import ../service-b

(* service-b/main.aro *)
import ../service-a
```

The compiler handles this by loading all files from all imported applications, building a unified symbol table, and resolving references across all loaded feature sets.

However, circular dependencies usually indicate poor architecture. If two applications need each other, consider:

1. Extracting shared code to a third application that both import
2. Using events instead of direct access
3. Reorganizing the application boundaries

23.10 What Is Not Provided

ARO deliberately omits many features found in other module systems:

- **Module declarations** - No `module com.example.foo`
- **Namespace qualifiers** - No `com.example.foo.MyType`
- **Selective imports** - No `import { User, Order } from ./users`
- **Import aliases** - No `import ./users as u`
- **Package manifests** - No `Package.yaml` or `aro.config`
- **Version constraints** - No `^1.0.0` or `~2.1.0`
- **Remote package repositories** - No central registry

These are implementation concerns that add complexity without matching how ARO applications are designed to work. If you need versioning, use git. If you need remote packages, use git submodules or symbolic links.

23.11 Summary

The import system embodies ARO's philosophy of simplicity:

1. `import ../path` imports another application
2. Everything becomes accessible after import
3. No visibility modifiers complicate decisions
4. Each application can run standalone or be composed
5. Native compilation produces self-contained binaries

This is not enterprise-grade module management. It is application composition for developers who want to build systems from small, understandable pieces.

* * *

Next: Chapter 24 — Control Flow

§ Chapter 24: Control Flow

ARO provides control flow constructs for conditional execution. This chapter covers how to make decisions in your feature sets using guarded statements and match expressions.

When Guards

The `when` clause conditionally executes a single statement. If the condition is false, the statement is skipped and execution continues to the next statement.

Syntax

```
<Action> the <result> preposition the <object> when <condition>.
```

Basic Guards

```
(getUser: User API) {
    <Extract> the <user-id> from the <pathParameters: id>.
    <Retrieve> the <user> from the <user-repository> where id = <user-id>.

    (* Return NotFound only when user is empty *)
    <Return> a <NotFound: status> for the <missing: user> when <user> is empty.

    <Return> an <OK: status> with <user>.
}
```

Guard Examples

```
(* Only return OK when count is not zero *)
<Return> an <OK: status> with <items> when <count> is not 0.

(* Send notification only when user has email *)
<Send> the <notification> to the <user: email> when <user: email> exists.

(* Log admin access only for admins *)
<Log> the <admin-access> for the <audit> when <user: role> = "admin".

(* Return error when validation fails *)
<Return> a <BadRequest: status> for the <invalid: input> when <validation> is failed.
```

Comparison Operators

Operator	Meaning
`is`	Equality
`is not`	Inequality
`is empty`	Null/empty check
`is not empty`	Has value
`exists`	Value exists
`is defined`	Value is defined
`is null`	Value is null
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal
`<=`	Less than or equal
`=`	Equality
`!=`	Strict inequality

Boolean Operators

Combine conditions with `and`, `or`, `not`:

```
(* Multiple conditions with and *)
<Return> an <OK: status> with <user> when <user: active> is true and <user: verified> is true.

(* Either condition with or *)
<Return> a <BadRequest: status> for the <unavailable: product> when <stock> is empty or <stock> <

(* Negation *)
<Allow> the <access> for the <user> when not <user: banned>.

(* Complex condition *)
<Grant> the <admin-features> for the <user> when (<user: role> is "admin" or <user: is-owner> is
```

Match Expressions

Pattern matching for multiple cases:

```
match <value> {
    case <pattern> {
        (* handle this case *)
    }
    case <pattern> where <condition> {
        (* handle case with guard *)
    }
    otherwise {
        (* handle all other cases *)
    }
}
```

Simple Value Matching

```
(PUT /orders/{id}/status: Order API) {
    <Extract> the <order-id> from the <pathParameters: id>.
    <Extract> the <new-status> from the <request: body.status>.
    <Retrieve> the <order> from the <order-repository> where id = <order-id>.

    match <new-status> {
        case "confirmed" {
            <Validate> the <order> for the <confirmation-rules>.
            <Emit> an <OrderConfirmed: event> with <order>.
        }
        case "shipped" {
            <Validate> the <order> for the <shipping-rules>.
            <Emit> an <OrderShipped: event> with <order>.
        }
        case "delivered" {
            <Emit> an <OrderDelivered: event> with <order>.
        }
        case "cancelled" {
            <Emit> an <OrderCancelled: event> with <order>.
        }
        otherwise {
            <Return> a <BadRequest: status> for the <invalid: status>.
        }
    }

    <Transform> the <updated-order> from the <order> with { status: <new-status> }.
    <Store> the <updated-order> into the <order-repository>.
    <Return> an <OK: status> with <updated-order>.
}
```

HTTP Method Routing

```
match <http: method> {
    case "GET" {
        <Retrieve> the <resource> from the <database>.
    }
    case "POST" {
        <Create> the <resource> in the <database>.
    }
    case "PUT" {
        <Update> the <resource> in the <database>.
    }
    case "DELETE" {
        <Remove> the <resource> from the <database>.
    }
    otherwise {
        <Return> a <MethodNotAllowed: error> for the <request>.
    }
}
```

Pattern Matching with Guards

```
match <user: subscription> {
    case <premium> where <user: credits> > 0 {
        <Grant> the <premium-features> for the <user>.
        <Deduct> the <credit> from the <user: account>.
    }
    case <premium> {
        <Notify> the <user> about the <low-credits>.
        <Grant> the <basic-features> for the <user>.
    }
    case <basic> {
        <Grant> the <basic-features> for the <user>.
    }
    otherwise {
        <Redirect> the <user> to the <subscription-page>.
    }
}
```

Status Code Handling

```
match <status-code> {
    case 200 {
        <Parse> the <response: body> from the <http-response>.
        <Return> the <data> for the <request>.
    }
    case 404 {
        <Return> a <NotFound: error> for the <request>.
    }
    case 500 {
        <Log> the <server-error> for the <monitoring>.
        <Return> a <ServerError> for the <request>.
    }
    otherwise {
        <Return> an <UnknownError> for the <request>.
    }
}
```

Regular Expression Patterns

Match statements support regex patterns for flexible string matching. Use forward slashes to delimit a regex pattern:

```
match <message.text> {
    case /^ERROR:/i {
        <Log> the <error: alert> for the <console> with <message.text>.
        <Emit> an <AlertTriggered: event> with <message>.
    }
    case /^WARN:/i {
        <Log> the <warning: message> for the <console> with <message.text>.
    }
    case /^[A-Z]{3}-\d{4}$/ {
        (* Matches ticket IDs like "ABC-1234" *)
        <Process> the <ticket-reference> from the <message>.
    }
    otherwise {
        <Log> the <info: message> for the <console> with <message.text>.
    }
}
```

Regex Flags

Flag	Description
`i`	Case insensitive
`s`	Dot matches newlines
`m`	Multiline (^ and $ match line boundaries)

Common Use Cases

```
(* Email validation *)
match <email> {
    case /^[\w.+-]+@[\w.-]+\.[a-zA-Z]{2,}$/i {
        <Return> an <OK: status> with { valid: true }.
    }
    otherwise {
        <Return> a <BadRequest: status> with { error: "Invalid email format" }.
    }
}

(* Command routing *)
match <input> {
    case /^\/help/i {
        <Emit> a <HelpRequested: event> with <message>.
    }
    case /^\/status\s+\w+$/i {
        <Emit> a <StatusQuery: event> with <message>.
    }
    otherwise {
        <Emit> a <MessageReceived: event> with <message>.
    }
}
```

Use `^` and `$` anchors when you need full-string matching rather than substring matching.

Common Patterns

Validate-or-Fail

```
(POST /users: User API) {
    <Extract> the <user-data> from the <request: body>.
    <Validate> the <user-data> for the <user-schema>.

    <Return> a <BadRequest: status> with <validation: errors> when <validation> is failed.

    <Create> the <user> with <user-data>.
    <Store> the <user> into the <user-repository>.
    <Return> a <Created: status> with <user>.
}
```

Find-or-404

```
(GET /products/{id}: Product API) {
    <Extract> the <product-id> from the <pathParameters: id>.
    <Retrieve> the <product> from the <product-repository> where id = <product-id>.

    <Return> a <NotFound: status> for the <missing: product> when <product> is empty.

    <Return> an <OK: status> with <product>.
}
```

Check-Permission

```
(DELETE /posts/{id}: Post API) {
    <Extract> the <post-id> from the <pathParameters: id>.
    <Retrieve> the <post> from the <post-repository> where id = <post-id>.

    <Return> a <NotFound: status> for the <missing: post> when <post> is empty.

    <Return> a <Forbidden: status> for the <unauthorized: deletion>
        when <post: authorId> is not <current-user: id> and <current-user: role> is not "admin".

    <Delete> the <post> from the <post-repository> where id = <post-id>.
    <Return> a <NoContent: status> for the <deletion>.
}
```

Fail Fast with Guards

Check error conditions early with guarded returns:

```
(POST /transfer: Banking) {
    <Extract> the <amount> from the <request: body.amount>.
    <Extract> the <from-account> from the <request: body.from>.
    <Extract> the <to-account> from the <request: body.to>.

    (* Early exits for invalid input *)
    <Return> a <BadRequest: status> for the <invalid: amount> when <amount> <= 0.
    <Return> a <BadRequest: status> for the <same: accounts> when <from-account> is <to-account>.

    <Retrieve> the <source> from the <account-repository> where id = <from-account>.

    <Return> a <NotFound: status> for the <missing: source-account> when <source> is empty.
    <Return> a <BadRequest: status> for the <insufficient: funds> when <source: balance> < <amount

    (* Now proceed with transfer *)
    <Retrieve> the <destination> from the <account-repository> where id = <to-account>.
    <Transfer> the <amount> from the <source> to the <destination>.
    <Return> an <OK: status> for the <transfer>.
}
```

Conditional Processing

```
(POST /orders: Order API) {
    <Extract> the <order-data> from the <request: body>.
    <Create> the <order> with <order-data>.

    (* Conditional discount *)
    <Compute> the <discount> with <order: total> * 0.1 when <order: total> >= 100.
    <Transform> the <order> from the <order> with { discount: <discount> } when <discount> exists

    (* Conditional express shipping *)
    <Compute> the <express-fee> for the <order> when <order: express> is true.
    <Transform> the <order> from the <order> with { shippingFee: <express-fee> } when <express-fee

    <Store> the <order> into the <order-repository>.
    <Return> a <Created: status> with <order>.
}
```

Complete Example

```
(User Authentication: Security) {
    <Require> the <request> from the <framework>.
    <Require> the <user-repository> from the <framework>.

    (* Extract credentials *)
    <Extract> the <username> from the <request: body>.
    <Extract> the <password> from the <request: body>.

    (* Validate input - guarded return *)
    <Return> a <BadRequest: error> for the <request>
        when <username> is empty or <password> is empty.

    (* Look up user *)
    <Retrieve> the <user> from the <user-repository>.

    (* Handle user not found - guarded statements *)
    <Log> the <failed-login: attempt> for the <username> when <user> is null.
    <Return> an <Unauthorized: error> for the <request> when <user> is null.

    (* Check account status with match *)
    match <user: status> {
        case "locked" {
            <Return> an <AccountLocked: error> for the <request>.
        }
        case "pending" {
            <Send> the <verification-email> to the <user: email>.
            <Return> a <PendingVerification: status> for the <request>.
        }
        case "active" {
            (* Verify password *)
            <Compute> the <password-hash> for the <password>.

            match <password-hash> {
                case <user: password-hash> {
                    <Create> the <session-token> for the <user>.
                    <Log> the <successful-login> for the <user>.
                    <Return> an <OK: status> with the <session-token>.
                }
                otherwise {
                    <Increment> the <failed-attempts> for the <user>.
                    <Lock> the <user: account> for the <security-policy>
                        when <failed-attempts> >= 5.
                    <Return> an <Unauthorized: error> for the <request>.
                }
            }
        }
        otherwise {
            <Return> an <InvalidAccountStatus: error> for the <request>.
        }
    }
}
```

Best Practices

Use Guards for Early Exits

Guards with `when` are ideal for: - Input validation - Preconditions - Error returns

```
(* Good - guards for early exit *)
<Return> a <BadRequest: status> for the <missing: id> when <user-id> is empty.
<Return> a <NotFound: status> for the <missing: user> when <user> is empty.
<Return> a <Forbidden: status> for the <private: profile> when <user: private> is true.

(* Continue with main logic *)
<Return> an <OK: status> with <user>.
```

Use Match for Multiple Outcomes

Match expressions are ideal for: - Status handling - Role-based logic - State machines - Multiple distinct cases

```
(* Good - match for multiple cases *)
match <order: status> {
    case "pending" { ... }
    case "processing" { ... }
    case "shipped" { ... }
    case "delivered" { ... }
    otherwise { ... }
}
```

Be Explicit in Conditions

```
(* Good - explicit conditions *)
<Grant> the <access> for the <user> when <user: active> is true and <user: verified> is true.

(* Avoid - implicit truthiness *)
<Grant> the <access> for the <user> when <user: active> and <user: verified>.
```

* * *

Next: Chapter 25 — Data Pipelines

§ Chapter 25: Data Pipelines

ARO provides a map/reduce style data pipeline for filtering, transforming, and aggregating collections. All operations are type-safe, with results typed via OpenAPI schemas.

Pipeline Operations

ARO supports four core data operations:

Operation	Purpose	Example
Fetch	Retrieve and filter data	`<Fetch> the <users: List<User>> from the <repository>...`
Filter	Filter existing collection	`<Filter> the <active: List<User>> from the <users>...`
Map	Transform to different type	`<Map> the <summaries> as List<UserSummary> from the <users>.`
Reduce	Aggregate to single value	`<Reduce> the <total> as Float from the <orders> with sum(<amount>).`

Type Annotation Syntax

ARO supports two equivalent syntaxes for type annotations:

```
(* Colon syntax: type inside angle brackets *)
<Filter> the <active-users: List<User>> from the <users> where <active> is true.

(* As syntax: type follows the result descriptor *)
<Filter> the <active-users> as List<User> from the <users> where <active> is true.
```

Both produce identical results. The `as Type` syntax (ARO-0038) can be more readable when the variable name is long, while the colon syntax keeps everything compact. Type annotations are optional since ARO infers types from the source collection.

Fetch

Retrieves data with optional filtering, sorting, and pagination.

```
(* Basic fetch *)
<Fetch> the <users: List<User>> from the <user-repository>.

(* With filter *)
<Fetch> the <active-users: List<User>> from the <users>
    where <status> is "active".

(* With sorting *)
<Fetch> the <recent-users: List<User>> from the <users>
    order by <created-at> desc.

(* With pagination *)
<Fetch> the <page: List<User>> from the <users>
    order by <name> asc
    limit 20
    offset 40.

(* Combined *)
<Fetch> the <top-customers: List<User>> from the <users>
    where <tier> is "premium"
    order by <total-purchases> desc
    limit 10.
```

Filter

Filters an existing collection with a predicate.

```
(* Filter by equality *)
<Filter> the <admins: List<User>> from the <users>
    where <role> is "admin".

(* Filter by comparison *)
<Filter> the <high-value: List<Order>> from the <orders>
    where <amount> > 1000.

(* Filter with multiple conditions *)
<Filter> the <active-premium: List<User>> from the <users>
    where <status> is "active" and <tier> is "premium".
```

Comparison Operators

Operator	Description	Example
`is` , `=`	Equality	`<status> is "active"`
`is not` , `!=`	Inequality	`<role> is not "guest"`
`>` , `>=` , `<` , `<=`	Comparison	`<age> >= 18`
`in`	Set membership	`<status> in ["a", "b"]`
`between`	Range	`<price> between 10 and 100`
`contains`	Substring	`<name> contains "test"`
`starts with`	Prefix match	`<email> starts with "admin"`
`ends with`	Suffix match	`<file> ends with ".pdf"`
`matches`	Regex pattern	`<email> matches /^admin@/i`

The `matches` operator supports regex literals with flags:

```
(* Filter users with admin emails *)
<Filter> the <admins: List<User>> from the <users>
    where <email> matches /^admin@|@admin\./i.

(* Filter valid email addresses *)
<Filter> the <valid-emails: List<User>> from the <users>
    where <email> matches /^[\w.+-]+@[\w.-]+\.[a-zA-Z]{2,}$/i.
```

* * *

Map

Transforms a collection to a different OpenAPI-defined type. The runtime automatically maps fields with matching names.

```
(* Map User to UserSummary *)
<Map> the <summaries: List<UserSummary>> from the <users>.
```

Requirements: - Target type must be defined in `openapi.yaml` components/schemas - Fields with matching names are automatically copied - Missing optional fields are omitted - Missing required fields cause an error

Example Types

```
# openapi.yaml
components:
  schemas:
    User:
      type: object
      properties:
        id: { type: string }
        name: { type: string }
        email: { type: string }
        password-hash: { type: string }
        created-at: { type: string }

    UserSummary:
      type: object
      properties:
        id: { type: string }
        name: { type: string }
        email: { type: string }
```

When mapping `List<User>` to `List<UserSummary>`, only `id`, `name`, and `email` are copied. Sensitive fields like `password-hash` are excluded.

* * *

Reduce

Aggregates a collection to a single value using aggregation functions.

```
(* Count items *)
<Reduce> the <user-count: Integer> from the <users>
    with count().

(* Sum numeric field *)
<Reduce> the <total-revenue: Float> from the <orders>
    with sum(<amount>).

(* Average *)
<Reduce> the <avg-price: Float> from the <products>
    with avg(<price>).

(* Min/Max *)
<Reduce> the <highest-score: Float> from the <scores>
    with max(<value>).

(* With filter *)
<Reduce> the <pending-count: Integer> from the <orders>
    where <status> is "pending"
    with count().
```

Aggregation Functions

Function	Description	Example
`count()`	Number of items	`with count()`
`sum(field)`	Sum of numeric field	`with sum(<amount>)`
`avg(field)`	Average of numeric field	`with avg(<price>)`
`min(field)`	Minimum value	`with min(<date>)`
`max(field)`	Maximum value	`with max(<score>)`
`first()`	First element	`with first()`
`last()`	Last element	`with last()`

* * *

Pipeline Composition

Chain operations to build complex data transformations:

```
(Generate Report: Analytics) {
    (* Step 1: Fetch recent orders *)
    <Fetch> the <recent-orders: List<Order>> from the <orders>
        where <created-at> > now().minus(30.days)
        order by <created-at> desc.

    (* Step 2: Filter high-value orders *)
    <Filter> the <high-value: List<Order>> from the <recent-orders>
        where <amount> > 1000.

    (* Step 3: Map to summaries *)
    <Map> the <summaries: List<OrderSummary>> from the <high-value>.

    (* Step 4: Calculate total *)
    <Reduce> the <total: Float> from the <high-value>
        with sum(<amount>).

    <Return> an <OK: status> with {
        orders: <summaries>,
        total: <total>,
        count: <high-value>.count()
    }.
}
```

* * *

Sorting

Sort results by one or more fields:

```
(* Single field, ascending *)
<Fetch> the <users: List<User>> from the <repository>
    order by <name> asc.

(* Single field, descending *)
<Fetch> the <recent: List<Order>> from the <orders>
    order by <created-at> desc.

(* Multiple fields *)
<Fetch> the <products: List<Product>> from the <catalog>
    order by <category> asc, <price> desc.
```

* * *

Pagination

Limit results with offset for pagination:

```
(* First page: items 1-20 *)
<Fetch> the <page1: List<User>> from the <users>
    order by <name> asc
    limit 20.

(* Second page: items 21-40 *)
<Fetch> the <page2: List<User>> from the <users>
    order by <name> asc
    limit 20
    offset 20.

(* Third page: items 41-60 *)
<Fetch> the <page3: List<User>> from the <users>
    order by <name> asc
    limit 20
    offset 40.
```

* * *

Complete Example

openapi.yaml

```
openapi: 3.0.3
info:
  title: Order Analytics
  version: 1.0.0

components:
  schemas:
    Order:
      type: object
      properties:
        id: { type: string }
        customer-id: { type: string }
        customer-name: { type: string }
        amount: { type: number }
        status: { type: string }
        region: { type: string }
        created-at: { type: string, format: date-time }
      required: [id, customer-id, amount, status]

    OrderSummary:
      type: object
      properties:
        id: { type: string }
        customer-name: { type: string }
        amount: { type: number }
      required: [id, customer-name, amount]
```

analytics.aro

```
(* Application entry point *)
(Application-Start: Order Analytics) {
    <Log> the <message> for the <console> with "Order Analytics ready".
    <Return> an <OK: status> for the <startup>.
}

(* Analytics report generation *)
(Generate Report: Order Analytics) {
    (* Fetch recent orders *)
    <Fetch> the <recent: List<Order>> from the <orders>
        where <created-at> > now().minus(30.days)
        order by <created-at> desc.

    (* Calculate metrics *)
    <Reduce> the <total-revenue: Float> from the <recent>
        with sum(<amount>).

    <Reduce> the <order-count: Integer> from the <recent>
        with count().

    <Reduce> the <avg-order: Float> from the <recent>
        with avg(<amount>).

    (* Filter pending orders *)
    <Filter> the <pending: List<Order>> from the <recent>
        where <status> is "pending".

    <Reduce> the <pending-count: Integer> from the <pending>
        with count().

    (* Map to summaries for response *)
    <Map> the <summaries: List<OrderSummary>> from the <recent>.

    <Return> an <OK: status> with {
        orders: <summaries>,
        metrics: {
            total-revenue: <total-revenue>,
            order-count: <order-count>,
            avg-order-value: <avg-order>,
            pending-count: <pending-count>
        }
    }.
}
```

* * *

Design Philosophy

ARO's data pipelines follow these principles:

1. **Type-First**: All results are typed via OpenAPI schemas
2. **No SQL Complexity**: No JOINs, subqueries, or CTEs
3. **Pipeline Style**: Chain simple operations for complex transformations
4. **Predictable Performance**: Simple operations with clear cost

For complex data needs, use multiple feature sets and compose results in your business logic.

Next: Chapter 26 — Repositories

§ Chapter 26: Repositories

Repositories provide persistent in-memory storage that survives across HTTP requests and event handlers. Unlike regular variables which are scoped to a single feature set execution, repositories maintain state for the lifetime of the application.

Overview

In ARO, each HTTP request creates a fresh execution context. Variables defined in one request aren't available in another:

```
(* This won't work - count resets on each request *)
(GET /count: Counter API) {
    <Create> the <count> with 0.
    <Compute> the <new-count> from <count> + 1.
    <Return> an <OK: status> with <new-count>.
}
```

Repositories solve this by providing shared storage:

```
(POST /increment: Counter API) {
    <Retrieve> the <counts> from the <counter-repository>.
    <Compute> the <current> from <counts: length>.
    <Store> the <current> into the <counter-repository>.
    <Return> an <OK: status> with <current>.
}

(GET /count: Counter API) {
    <Retrieve> the <counts> from the <counter-repository>.
    <Compute> the <total> from <counts: length>.
    <Return> an <OK: status> with { count: <total> }.
}
```

Repository Naming Convention

Repository names **must** end with `-repository`. This is how ARO distinguishes repositories from regular variables:

```
(* These are repositories *)
<user-repository>
<message-repository>
<order-repository>
<session-repository>

(* These are NOT repositories - just regular variables *)
<users>
<messages>
<user-data>
```

The naming convention: - Makes repositories visually distinct in code - Enables automatic persistence by the runtime - Follows ARO's self-documenting code philosophy

Storing Data

Use the `<Store>` action to save data to a repository:

```
<Store> the <data> into the <name-repository>.
```

Preposition Variants

All of these are equivalent:

```
<Store> the <user> into the <user-repository>.
<Store> the <user> in the <user-repository>.
<Store> the <user> to the <user-repository>.
```

Storage Semantics

Repositories use **list-based storage**. Each store operation appends to the list:

```
(* First request *)
<Store> the <user1> into the <user-repository>.
(* Repository: [user1] *)

(* Second request *)
<Store> the <user2> into the <user-repository>.
(* Repository: [user1, user2] *)

(* Third request *)
<Store> the <user3> into the <user-repository>.
(* Repository: [user1, user2, user3] *)
```

Example: Storing Messages

```
(postMessage: Chat API) {
    <Extract> the <data> from the <request: body>.
    <Extract> the <text> from the <data: message>.
    <Extract> the <author> from the <data: author>.

    <Create> the <message> with {
        text: <text>,
        author: <author>,
        timestamp: now
    }.

    <Store> the <message> into the <message-repository>.

    <Return> a <Created: status> with <message>.
}
```

Retrieving Data

Use the `<Retrieve>` action to fetch data from a repository:

```
<Retrieve> the <items> from the <name-repository>.
```

Return Value

- Returns a **list** of all stored items
- Returns an **empty list** `[]` if the repository is empty or doesn't exist
- Never throws an error for missing repositories

Example: Retrieving All Messages

```
(getMessages: Chat API) {
    <Retrieve> the <messages> from the <message-repository>.
    <Return> an <OK: status> with { messages: <messages> }.
}
```

Filtered Retrieval

Use `where` to filter results:

```
(getUserById: User API) {
    <Extract> the <id> from the <pathParameters: id>.
    <Retrieve> the <user> from the <user-repository> where id = <id>.
    <Return> an <OK: status> with <user>.
}
```

Single Item Retrieval

Use specifiers to retrieve a single item from a repository:

```
(* Get the most recently stored item *)
<Retrieve> the <message> from the <message-repository: last>.

(* Get the first stored item *)
<Retrieve> the <message> from the <message-repository: first>.

(* Get by numeric index - 0 = most recent *)
<Retrieve> the <latest> from the <message-repository: 0>.
<Retrieve> the <second-latest> from the <message-repository: 1>.
```

Numeric indices count from most recently added (0 = newest, 1 = second newest, etc.).

This is useful when you only need one item, like the latest message in a chat:

```
(getLatestMessage: Chat API) {
    <Retrieve> the <message> from the <message-repository: last>.
    <Return> an <OK: status> with { message: <message> }.
}
```

If the repository is empty, an empty string is returned.

Business Activity Scoping

Repositories are scoped to their **business activity**. Feature sets with the same business activity share repositories:

```
(* Same business activity: "Chat API" *)
(* These share the same <message-repository> *)

(postMessage: Chat API) {
    <Store> the <message> into the <message-repository>.
    <Return> a <Created: status>.
}

(getMessages: Chat API) {
    <Retrieve> the <messages> from the <message-repository>.
    <Return> an <OK: status> with <messages>.
}

(deleteMessage: Chat API) {
    (* Same repository as above *)
    <Retrieve> the <messages> from the <message-repository>.
    (* ... *)
}
```

Different Business Activities = Different Repositories

```
(* Business activity: "Chat API" *)
(postMessage: Chat API) {
    <Store> the <msg> into the <message-repository>.
}

(* Business activity: "Admin API" - DIFFERENT repository! *)
(postAuditLog: Admin API) {
    (* This <message-repository> is separate from Chat API's *)
    <Store> the <log> into the <message-repository>.
}
```

This scoping: - Prevents accidental data leakage between domains - Allows reuse of generic repository names - Enforces domain boundaries

Complete Example: Simple Chat Application

main.aro

```
(Application-Start: Simple Chat) {
    <Log> the <startup: message> for the <console> with "Starting Simple Chat...".
    <Start> the <http-server> for the <contract>.
    <Keepalive> the <application> for the <events>.
    <Return> an <OK: status> for the <startup>.
}
```

api.aro

```
(* GET /status - Return the last message *)
(getStatus: Simple Chat API) {
    <Retrieve> the <message> from the <message-repository: last>.
    <Return> an <OK: status> with { message: <message> }.
}

(* POST /status - Store a new message *)
(postStatus: Simple Chat API) {
    <Extract> the <message> from the <body: message>.

    <Store> the <message> into the <message-repository>.

    <Return> a <Created: status> with { message: <message> }.
}
```

Testing

```
# Post a message
curl -X POST http://localhost:8080/status \
  -H 'Content-Type: application/json' \
  -d '{"message":"Hello!"}'
# Response: {"message":"Hello!"}

# Post another message
curl -X POST http://localhost:8080/status \
  -H 'Content-Type: application/json' \
  -d '{"message":"World!"}'
# Response: {"message":"World!"}

# Get the last message
curl http://localhost:8080/status
# Response: {"message":"World!"}
```

Deleting from Repositories

Use the `<Delete>` action with a `where` clause to remove items from a repository:

```
<Delete> the <user> from the <user-repository> where id = <userId>.
```

Example: Deleting a User

```
(deleteUser: User API) {
    <Extract> the <userId> from the <pathParameters: id>.
    <Delete> the <user> from the <user-repository> where id = <userId>.
    <Return> an <OK: status> with { deleted: <userId> }.
}
```

The deleted item(s) are bound to the result variable (`user` in this example).

Repository Observers

Repository observers are feature sets that automatically react to repository changes. They receive access to both old and new values, enabling audit logging, synchronization, and reactive patterns.

Observer Syntax

Create an observer by naming your feature set's business activity as `{repository-name} Observer`:

```
(Audit Changes: user-repository Observer) {
    <Extract> the <changeType> from the <event: changeType>.
    <Extract> the <newValue> from the <event: newValue>.
    <Extract> the <oldValue> from the <event: oldValue>.

    <Log> the <audit: message> for the <console> with <changeType>.
    <Return> an <OK: status> for the <audit>.
}
```

Event Payload

Observers receive an event with the following fields:

Field	Type	Description
`repositoryName`	String	The repository name (e.g., "user-repository")
`changeType`	String	"created", "updated", or "deleted"
`entityId`	String?	ID of the changed entity (if available)
`newValue`	Any?	The new value (nil for deletes)
`oldValue`	Any?	The previous value (nil for creates)
`timestamp`	Date	When the change occurred

Change Types

Observers are triggered for three types of changes:

- **created**: New item stored (no previous value existed with matching ID)
- **updated**: Existing item modified (matched by ID)
- **deleted**: Item removed using `<Delete>` action

Example: Tracking User Changes

```
(Track User Changes: user-repository Observer) {
    <Extract> the <changeType> from the <event: changeType>.
    <Extract> the <entityId> from the <event: entityId>.

    <Compare> the <changeType> equals "updated".

    <Extract> the <oldName> from the <event: oldValue: name>.
    <Extract> the <newName> from the <event: newValue: name>.

    <Log> the <change: message> for the <console>
        with "User " + <entityId> + " renamed from " + <oldName> + " to " + <newName>.

    <Return> an <OK: status> for the <tracking>.
}
```

Multiple Observers

You can have multiple observers for the same repository:

```
(* Audit logging observer *)
(Log All Changes: user-repository Observer) {
    <Extract> the <changeType> from the <event: changeType>.
    <Log> the <audit: message> for the <console> with <changeType>.
    <Return> an <OK: status>.
}

(* Email notification observer *)
(Notify Admin: user-repository Observer) {
    <Extract> the <changeType> from the <event: changeType>.
    <Compare> the <changeType> equals "deleted".
    <Send> the <notification> to the <admin-email>.
    <Return> an <OK: status>.
}
```

Lifetime and Persistence

Application Lifetime

Repositories persist for the **lifetime of the application**:

- Created when first accessed
- Survive across all HTTP requests

- Cleared when application restarts

No Disk Persistence

Repositories are **in-memory only**:

- Data is lost when the application stops
- No external database required
- Fast and simple for prototyping

For persistent storage, use a database integration (future ARO feature).

Best Practices

Use Descriptive Repository Names

```
(* Good - clear what's stored *)
<user-repository>
<pending-order-repository>
<session-token-repository>

(* Avoid - too generic *)
<data-repository>
<stuff-repository>
```

One Repository Per Domain Concept

```
(* Good - separate repositories for different concepts *)
<user-repository>
<order-repository>
<product-repository>

(* Avoid - mixing concepts *)
<everything-repository>
```

Keep Repository Data Simple

Store simple, serializable data:

```
(* Good - simple object *)
<Create> the <user> with {
    id: <id>,
    name: <name>,
    email: <email>
}.
<Store> the <user> into the <user-repository>.

(* Avoid - complex nested structures *)
<Store> the <entire-request-context> into the <debug-repository>.
```

Next: Chapter 27 — System Commands

§ Chapter 27: System Commands

ARO provides the `<Exec>` action for executing shell commands on the host system. This chapter covers command execution, result handling, and security considerations.

Basic Execution

Simple Commands

Execute shell commands using the `<Exec>` action:

```
<Exec> the <result> for the <command> with "ls -la".
<Exec> the <listing> for the <files> with "find . -name '*.txt'".
<Exec> the <status> for the <check> with "git status".
```

Using Variables

Build commands dynamically with variables:

```
<Create> the <directory> with "/var/log".
<Exec> the <result> for the <listing> with "ls -la ${directory}".

<Create> the <pattern> with "*.aro".
<Exec> the <files> for the <search> with "find . -name '${pattern}'".
```

Result Object

Every `<Exec>` action returns a structured result object:

Field	Type	Description
`error`	Boolean	`true` if command failed (non-zero exit code)
`message`	String	Human-readable status message
`output`	String	Command stdout (or stderr if error)
`exitCode`	Int	Process exit code (0 = success, -1 = timeout)
`command`	String	The executed command string

Accessing Result Fields

```
<Exec> the <result> for the <command> with "whoami".

(* Access individual fields *)
<Log> the <user: message> for the <console> with <result.output>.
<Log> the <exit: code> for the <console> with <result.exitCode>.

(* Check for errors *)
<Log> the <error: message> for the <console> with <result.message> when <result.error> = true.
```

Error Handling

Checking for Errors

```
(Check Disk Space: System Monitor) {
    <Exec> the <result> for the <disk-check> with "df -h".

    <Log> the <error> for the <console> with <result.message> when <result.error> = true.
    <Return> an <Error: status> with <result> when <result.error> = true.

    <Return> an <OK: status> with <result>.
}
```

Handling Non-Zero Exit Codes

```
(Git Status: Version Control) {
    <Exec> the <result> for the <git> with "git status --porcelain".

    <Log> the <warning> for the <console> with "Not a git repository" when <result.exitCode> != 0
    <Return> a <BadRequest: status> with { error: "Not a git repository" } when <result.exitCode>

    <Return> an <OK: status> with { message: "Working tree clean" } when <result.output> is empty

    <Return> an <OK: status> with { changes: <result.output> }.
}
```

Timeout Handling

Commands that exceed the timeout return with `exitCode: -1`:

```
<Exec> the <result> for the <long-task> with {
    command: "sleep 60",
    timeout: 5000
}.

<Log> the <timeout> for the <console> with "Command timed out" when <result.exitCode> = -1.
```

Configuration Options

For advanced control, use object syntax:

```
<Exec> the <result> on the <system> with {
    command: "npm install",
    workingDirectory: "/app",
    timeout: 60000,
    shell: "/bin/bash",
    environment: { NODE_ENV: "production" }
}.
```

Available Options

Option	Type	Default	Description
`command`	String	(required)	The shell command to execute
`workingDirectory`	String	current	Working directory for the command
`timeout`	Int	30000	Timeout in milliseconds
`shell`	String	/bin/sh	Shell to use for execution
`environment`	Object	(inherited)	Additional environment variables
`captureStderr`	Boolean	true	Include stderr in output

Context-Aware Output

Following ARO's context-aware response formatting (ARO-0031), `<Exec>` results display differently based on execution context.

Console Output (Human Context)

```
command: ls -la
exitCode: 0
error: false
message: Command executed successfully
output:
  total 48
  drwxr-xr-x  12 user  staff   384 Dec 23 10:00 .
  -rw-r--r--   1 user  staff  1234 Dec 23 10:00 main.aro
```

HTTP Response (Machine Context)

```
{
    "status": "OK",
    "reason": "success",
    "data": {
        "result": {
            "error": false,
            "message": "Command executed successfully",
            "output": "total 48\ndrwxr-xr-x  12 user  staff...",
            "exitCode": 0,
            "command": "ls -la"
        }
    }
}
```

Common Patterns

Directory Listing

```
(Application-Start: Directory Lister) {
    <Log> the <startup: message> for the <console> with "Directory Lister".
    <Exec> the <listing> for the <command> with "ls -la".
    <Return> an <OK: status> for the <listing>.
}
```

System Information

```
(System Info: Status API) {
    <Exec> the <hostname> for the <check> with "hostname".
    <Exec> the <uptime> for the <check> with "uptime".
    <Exec> the <memory> for the <check> with "free -h".

    <Create> the <info> with {
        hostname: <hostname.output>,
        uptime: <uptime.output>,
        memory: <memory.output>
    }.

    <Return> an <OK: status> with <info>.
}
```

Build Pipeline

```
(Run Build: CI Pipeline) {
    <Log> the <step> for the <console> with "Installing dependencies...".
    <Exec> the <install> for the <npm> with {
        command: "npm install",
        workingDirectory: "./app",
        timeout: 120000
    }.

    <Return> an <Error: status> with <install> when <install.error> = true.

    <Log> the <step> for the <console> with "Running tests...".
    <Exec> the <test> for the <npm> with {
        command: "npm test",
        workingDirectory: "./app"
    }.

    <Return> an <Error: status> with <test> when <test.error> = true.

    <Log> the <step> for the <console> with "Building...".
    <Exec> the <build> for the <npm> with {
        command: "npm run build",
        workingDirectory: "./app"
    }.

    <Return> an <OK: status> with <build>.
}
```

Health Checks

```
(Health Check: Monitoring) {
    <Exec> the <curl> for the <health> with {
        command: "curl -s http://localhost:8080/health",
        timeout: 5000
    }.

    <Return> a <ServiceUnavailable: status> with {
        service: "api",
        error: <curl.message>
    } when <curl.error> = true.

    <Return> an <OK: status> with { healthy: true }.
}
```

Process Management

```
(List Processes: Admin API) {
    <Exec> the <processes> for the <list> with "ps aux | head -20".
    <Return> an <OK: status> with <processes>.
}

(Check Process: Admin API) {
    <Extract> the <name> from the <queryParameters: name>.
    <Exec> the <result> for the <check> with "pgrep -l ${name}".

    <Return> an <OK: status> with { running: false, process: <name> } when <result.error> = true.

    <Return> an <OK: status> with { running: true, process: <name>, pids: <result.output> }.
}
```

Security Considerations

Command Injection Prevention

Be cautious when constructing commands from user input:

```
(* DANGEROUS - user input directly in command *)
<Exec> the <result> for the <command> with "ls ${userInput}".

(* SAFER - validate input first *)
<Validate> the <path> for the <userInput> against "^[a-zA-Z0-9_/.-]+$".
<Return> a <BadRequest: status> with "Invalid path characters" when <path> is not <valid>.
<Exec> the <result> for the <command> with "ls ${path}".
```

Best Practices

1. **Never trust user input** - Always validate and sanitize before using in commands
2. **Use allowlists** - Define allowed commands or patterns rather than blocking bad ones
3. **Limit permissions** - Run the ARO application with minimal required privileges
4. **Set timeouts** - Always specify reasonable timeouts to prevent hanging
5. **Log commands** - Keep audit logs of executed commands for security review

Sandboxing (Future)

Future versions may support sandboxing options:

```
<Exec> the <result> for the <command> with {
    command: "npm install",
    sandbox: {
        network: false,
        filesystem: ["/app"],
        maxMemory: "512MB",
        maxTime: 60000
    }
}.
```

* * *

Next: Chapter 28 — HTTP Client

§ Chapter 28: HTTP Client

ARO provides HTTP client capabilities for making requests to external services and APIs.

Making HTTP Requests

Use the `<Request>` action to make HTTP requests. The preposition determines the HTTP method:

- `from` - GET request
- `to` - POST request
- `via METHOD` - Explicit method (PUT, DELETE, PATCH)

GET Requests

Use `from` preposition to make GET requests:

```
(* Simple GET request *)
<Create> the <url> with "https://api.example.com/users".
<Request> the <users> from the <url>.
```

POST Requests

Use `to` preposition to make POST requests:

```
(* POST with data from context *)
<Create> the <user-data> with { name: "Alice", email: "alice@example.com" }.
<Create> the <api-url> with "https://api.example.com/users".
<Request> the <result> to the <api-url> with <user-data>.
```

Other HTTP Methods

Use `via` preposition with method specifier for PUT, DELETE, PATCH:

```
(* PUT request *)
<Request> the <result> via PUT the <url> with <update-data>.

(* DELETE request *)
<Request> the <result> via DELETE the <url>.

(* PATCH request *)
<Request> the <result> via PATCH the <url> with <partial-data>.
```

Config Object Syntax

For full control over requests, use a config object with the `with { ... }` clause:

```
(* POST with custom headers and timeout *)
<Request> the <response> from the <api-url> with {
    method: "POST",
    headers: { "Content-Type": "application/json", "Authorization": "Bearer token" },
    body: <data>,
    timeout: 60
}.

(* GET with authorization header *)
<Request> the <protected-data> from the <api-url> with {
    headers: { "Authorization": "Bearer my-token" }
}.
```

Config Options:

Option	Type	Description
`method`	String	HTTP method: GET, POST, PUT, DELETE, PATCH
`headers`	Map	Custom HTTP headers
`body`	String/Map	Request body (auto-serialized to JSON if map)
`timeout`	Number	Request timeout in seconds (default: 30)

The config `method` overrides the preposition-based method detection. This allows you to use `from` (which defaults to GET) while specifying POST in the config.

Response Handling

The `<Request>` action automatically parses JSON responses. After a request, these variables are available:

Variable	Description
`result`	Parsed response body (JSON as map/list, or string)
`result.statusCode`	HTTP status code (e.g., 200, 404)
`result.headers`	Response headers as map
`result.isSuccess`	Boolean: true if status 200-299

```
<Create> the <api-url> with "https://api.example.com/users".
<Request> the <response> from the <api-url>.

(* Access response metadata *)
<Extract> the <status> from the <response: statusCode>.
<Extract> the <is-ok> from the <response: isSuccess>.

(* Access response body - response IS the parsed body *)
<Extract> the <first-user> from the <response: 0>.
```

Complete Example

```
(* Fetch weather data from Open-Meteo API *)

(Application-Start: Weather Client) {
    <Log> the <message> for the <console> with "Fetching weather...".

    <Create> the <api-url> with "https://api.open-meteo.com/v1/forecast?latitude=52.52&longitude=1
    <Request> the <weather> from the <api-url>.

    <Log> the <data: message> for the <console> with <weather>.

    <Return> an <OK: status> for the <startup>.
}
```

Implementation Notes

- Uses Foundation's URLSession for HTTP requests

- Compatible with both interpreter (`aro run`) and compiled binaries (`aro build`)
- Default timeout: 30 seconds
- Automatically parses JSON responses
- Thread-safe for concurrent requests
- Available on macOS and Linux (not Windows)

Next: Chapter 29 — Concurrency

§ Chapter 29: Concurrency

ARO's concurrency model is radically simple: **feature sets are async, statements are sync**. This chapter explains how ARO handles concurrent operations without requiring you to think about threads, locks, or async/await.

The Philosophy

ARO's concurrency model matches how project managers think:

- **"When X happens, do Y"** - Feature sets are triggered by events
- **"Do this, then this, then this"** - Steps happen in order

You don't think about threads, locks, race conditions, or async/await. You think about things happening and responding to them in sequence.

Feature Sets Are Async

Every feature set runs asynchronously when triggered by an event:

```
+-----------------------------------------------------+
|                    Event Bus                        |
|                                                     |
|  HTTP Request --+---> (listUsers: User API)         |
|                 |                                   |
|  Socket Data ---+---> (Handle Data: Socket Handler) |
|                 |                                   |
|  File Changed --+---> (Process File: File Handler)  |
|                 |                                   |
|  UserCreated ---+---> (Send Email: Notification)    |
|                                                     |
|  (Multiple events trigger multiple feature sets     |
|   running concurrently)                             |
+-----------------------------------------------------+
```

When multiple events arrive, multiple feature sets execute simultaneously. 100 HTTP requests = 100 concurrent feature set executions.

Statements Are Sync

Inside a feature set, statements execute **synchronously** and **serially**:

```
(Process Order: Order API) {
    <Extract> the <data> from the <request: body>.      (* 1. First *)
    <Validate> the <data> for the <order-schema>.       (* 2. Second *)
    <Create> the <order> with <data>.                   (* 3. Third *)
    <Store> the <order> in the <order-repository>.      (* 4. Fourth *)
    <Emit> to <Send Confirmation> with <order>.         (* 5. Fifth *)
    <Return> a <Created: status> with <order>.          (* 6. Last *)
}
```

Each statement completes before the next one starts. No callbacks. No promises. No async/await syntax. Just sequential execution.

Why This Model?

Simplicity

Traditional async code in JavaScript:

```
async function processOrder(req) {
    const data = await extractData(req);
    const validated = await validate(data);
    const order = await createOrder(validated);
    await storeOrder(order);
    await emitEvent('OrderCreated', order);
    return { status: 201, body: order };
}
```

ARO code:

```
(Process Order: Order API) {
    <Extract> the <data> from the <request: body>.
    <Validate> the <data> for the <order-schema>.
    <Create> the <order> with <data>.
    <Store> the <order> in the <order-repository>.
    <Emit> to <Send Confirmation> with <order>.
    <Return> a <Created: status> with <order>.
}
```

No `async`. No `await`. Just statements in order.

No Race Conditions

Within a feature set, there's no shared mutable state problem:

- Variables are scoped to the feature set
- Statements execute serially
- No concurrent access to the same data

Natural Event Flow

Events naturally express concurrency:

- User A requests an order while User B requests their profile
- Both feature sets run concurrently
- Each processes their own data independently

Runtime Optimization

While you write synchronous-looking code, the ARO runtime executes operations **asynchronously** based on data dependencies. This is transparent to you.

How It Works

The runtime performs **data-flow driven execution**:

1. **Eager Start**: I/O operations begin immediately (non-blocking)
2. **Dependency Tracking**: The runtime tracks which variables each statement needs
3. **Lazy Synchronization**: Only wait for data when it's actually used
4. **Preserved Semantics**: Results appear in statement order

Example

```
(Process Config: File Handler) {
    <Open> the <config-file> from the <path>.        (* 1. Starts file load *)
    <Compute> the <hash> for the <request>.          (* 2. Runs immediately *)
    <Log> the <status> for the <request>.            (* 3. Runs immediately *)
    <Parse> the <config> from the <config-file>.     (* 4. Waits for file *)
    <Return> an <OK: status> with <config>.
}
```

What happens:

- Statement 1 kicks off file loading (async, returns immediately)
- Statements 2 and 3 execute while the file loads in background
- Statement 4 waits only if the file isn't ready yet
- You see: synchronous execution
- Runtime does: parallel I/O with sequential semantics

Write synchronous code. Get async performance.

Event Emission

Feature sets can trigger other feature sets:

```
(Create User: User API) {
    <Extract> the <data> from the <request: body>.
    <Create> the <user> with <data>.
    <Store> the <user> in the <user-repository>.

    (* Triggers other feature sets asynchronously *)
    <Emit> to <Send Welcome Email> with <user>.

    (* Continues immediately, doesn't wait for handler *)
    <Return> a <Created: status> with <user>.
}

(Send Welcome Email: Notifications) {
    <Extract> the <email> from the <event: email>.
    <Send> the <welcome-email> to the <email>.
    <Return> an <OK: status>.
}
```

When `<Emit>` executes:

1. The event is dispatched to the target feature set
2. Execution continues in the current feature set
3. The target handler starts executing independently

No Concurrency Primitives

ARO explicitly does **not** provide:

- `async` / `await` keywords
- Promises / Futures
- Threads / Task spawning
- Locks / Mutexes / Semaphores
- Channels
- Actors
- Parallel for loops

These are implementation concerns. The runtime handles them. You write sequential code that responds to events.

Examples

HTTP Server

```
(Application-Start: My API) {
    <Start> the <http-server> on port 8080.
    <Keepalive> the <application> for the <events>.
    <Return> an <OK: status>.
}

(* Each request triggers this independently *)
(getUser: User API) {
    <Extract> the <id> from the <pathParameters: id>.
    <Retrieve> the <user> from the <user-repository> where id = <id>.
    <Return> an <OK: status> with <user>.
}
```

Socket Echo Server

```
(Application-Start: Echo Server) {
    <Start> the <socket-server> on port 9000.
    <Keepalive> the <application> for the <events>.
    <Return> an <OK: status>.
}

(* Each client message triggers this independently *)
(Handle Data: Socket Event Handler) {
    <Extract> the <data> from the <event: data>.
    <Extract> the <connection> from the <event: connection>.
    <Send> the <data> to the <connection>.
    <Return> an <OK: status>.
}
```

File Watcher

```
(Application-Start: File Watcher) {
    <Watch> the <directory> for the <changes> with "./watched".
    <Keepalive> the <application> for the <events>.
    <Return> an <OK: status>.
}

(* Each file change triggers this independently *)
(Handle File Change: File Event Handler) {
    <Extract> the <path> from the <event: path>.
    <Extract> the <type> from the <event: type>.
    <Log> the <change: message> with <path> and <type>.
    <Return> an <OK: status>.
}
```

Summary

Concept	Behavior
Feature sets	Run async (triggered by events)
Statements	Appear sync (serial execution)
I/O operations	Async under the hood
Events	Non-blocking dispatch
Concurrency primitives	None needed

Write synchronous code. Get async performance. No callbacks, no promises, no await.

* * *

Next: Chapter 30 — Context-Aware Response Formatting

§ Chapter 30: Context-Aware Response Formatting

ARO automatically formats responses and log output based on the execution context. The same code produces different output formats depending on how it's invoked.

Execution Contexts

ARO recognizes three execution contexts:

Context	Trigger	Output Format
Human	`aro run` , CLI execution	Readable text
Machine	HTTP API, events	JSON/structured data
Developer	`aro test` , debug mode	Detailed diagnostics

How It Works

Human Context (Default)

When you run ARO from the command line, output is formatted for readability:

```
(Application-Start: Example) {
    <Log> the <message> for the <console> with "Hello, World!".
    <Return> an <OK: status> for the <startup>.
}
```

Running with `aro run` :

```
[Application-Start] Hello, World!
```

Machine Context

When code runs via HTTP request or event handler, output is structured data:

```
{"level":"info","source":"Application-Start","message":"Hello, World!"}
```

Developer Context

During test execution, output is displayed as a formatted table with type annotations:

```
+-----------------------------------------+
| LOG [console] Application-Start         |
+-----------------------------------------+
| message: String("Hello, World!")        |
+-----------------------------------------+
```

Automatic Detection

The runtime automatically detects context:

Entry Point	Context
`aro run` command	Human
`aro test` command	Developer
HTTP route handler	Machine
Event dispatch	Machine
`--debug` flag	Developer

Example: Same Code, Different Contexts

```
(getUser: User API) {
    <Retrieve> the <user> from the <user-repository> where id = <id>.
    <Log> the <status> for the <console> with "User retrieved".
    <Return> an <OK: status> with <user>.
}
```

CLI Output (Human)

```
[getUser] User retrieved
[OK] success
  user.id: 123
  user.name: Alice
```

HTTP Response (Machine)

```
{
  "status": "OK",
  "reason": "success",
  "data": {
    "user": {"id": 123, "name": "Alice"}
  }
}
```

Test Output (Developer)

```
+-----------------------------------------+
| LOG [console] getUser                   |
+-----------------------------------------+
| message: String("User retrieved")       |
+-----------------------------------------+

+-----------------------------------------+
| Response<OK>                            |
+----------------+------------------------+
| reason         | String("success")      |
| user.id        | Int(123)               |
| user.name      | String("Alice")        |
+----------------+------------------------+
```

Benefits

1. **Write Once**: No need for separate formatting code
2. **Automatic Adaptation**: Output suits the consumer automatically
3. **Consistent Behavior**: Same logic, appropriate presentation
4. **Debug Friendly**: Rich diagnostics during development

Integration with Actions

Log Action

The `<Log>` action respects output context:

```
<Log> the <message> for the <console> with <value>.
```

- **Human**: `[FeatureSetName] value`
- **Machine**: `{"level":"info","source":"FeatureSetName","message":"value"}`
- **Developer**: Formatted table with type annotations

Return Action

The `<Return>` action sets the response, which is formatted based on context:

```
<Return> an <OK: status> with <result>.
```

Next: Chapter 31 — Type System

§ Chapter 31: Type System

ARO has a simple type system: four built-in primitives, two collection types, and complex types defined externally in OpenAPI. This chapter explains how types work in ARO.

Primitive Types

ARO has four built-in primitive types:

Type	Description	Literal Examples
`String`	Text	`"hello"` , `'world'`
`Integer`	Whole numbers	`42` , `-17` , `0xFF`
`Float`	Decimal numbers	`3.14` , `2.5e10`
`Boolean`	True/False	`true` , `false`

Collection Types

ARO has two built-in collection types:

Type	Description	Literal Examples
`List<T>`	Ordered collection	`[1, 2, 3]`
`Map<K, V>`	Key-value pairs	`{ name: "Alice", age: 30 }`

List Examples

```
<Create> the <numbers: List<Integer>> with [1, 2, 3].
<Create> the <names: List<String>> with ["Alice", "Bob", "Charlie"].

for each <number> in <numbers> {
    <Log> the <value> for the <console> with <number>.
}
```

Map Examples

```
<Create> the <config: Map<String, Integer>> with {
    port: 8080,
    timeout: 30
}.

<Extract> the <port> from the <config: port>.
```

Complex Types from OpenAPI

All complex types (records, enums) are defined in `openapi.yaml`. There are no `type` or `enum` keywords in ARO.

Why OpenAPI?

1. **Single Source of Truth**: Types are defined once, used everywhere
2. **Contract-First**: Design your data before implementing
3. **Documentation**: OpenAPI schemas are self-documenting
4. **Validation**: Runtime can validate against schemas

Defining Types in OpenAPI

```
# openapi.yaml
openapi: 3.0.3
info:
  title: My Application
  version: 1.0.0

components:
  schemas:
    User:
      type: object
      properties:
        id:
          type: string
        name:
          type: string
        email:
          type: string
        status:
          $ref: '#/components/schemas/UserStatus'
      required:
        - id
        - name
        - email

    UserStatus:
      type: string
      enum:
        - active
        - inactive
        - suspended
```

Using OpenAPI Types in ARO

```
(Create User: User Management) {
    <Extract> the <data> from the <request: body>.

    (* User type comes from openapi.yaml *)
    <Create> the <user: User> with <data>.

    (* Access fields defined in the schema *)
    <Log> the <message> for the <console> with <user: name>.

    <Return> a <Created: status> with <user>.
}
```

Type Annotations

Type annotations specify the type of a variable.

Syntax

```
<name: Type>
```

Examples

```
<name: String>                    (* Primitive *)
<count: Integer>                  (* Primitive *)
<items: List<String>>             (* Collection of primitives *)
<user: User>                      (* OpenAPI schema reference *)
<users: List<User>>               (* Collection of OpenAPI types *)
<config: Map<String, Integer>>    (* Map with primitives *)
```

When to Use Type Annotations

Type annotations are optional but recommended when:

- Extracting data from external sources
- Working with OpenAPI schema types
- Clarifying intent in complex operations

```
(* Recommended: explicit types for external data *)
<Extract> the <userId: String> from the <request: body>.
<Extract> the <items: List<OrderItem>> from the <request: body>.
<Retrieve> the <user: User> from the <user-repository> where id = <userId>.
```

Type Inference

Types are inferred from literals and expressions:

```
<Create> the <count> with 42.              (* count: Integer *)
<Create> the <name> with "John".           (* name: String *)
<Create> the <active> with true.           (* active: Boolean *)
<Create> the <price> with 19.99.           (* price: Float *)
<Create> the <items> with [1, 2, 3].       (* items: List<Integer> *)
```

No Optionals

ARO has no optional types (`?` , `null` , `undefined` , `Option<T>`). Every variable has a value.

What Happens When Data Doesn't Exist?

The runtime throws a descriptive error:

```
(Get User: API) {
    <Extract> the <id> from the <pathParameters: id>.
    <Retrieve> the <user: User> from the <user-repository> where id = <id>.
    (* If user doesn't exist, runtime throws: *)
    (* "Cannot retrieve the user from the user-repository where id = 123" *)

    <Return> an <OK: status> with <user>.
}
```

No Null Checks Needed

Traditional code:

```
const user = await repository.find(id);
if (user === null) {
    throw new Error("User not found");
}
console.log(user.name);
```

ARO code:

```
<Retrieve> the <user> from the <user-repository> where id = <id>.
<Log> the <message> for the <console> with <user: name>.
```

The runtime error message IS the error handling. See the Error Handling chapter for more details.

OpenAPI Without HTTP

You can use OpenAPI just for type definitions, without any HTTP routes:

```
# openapi.yaml - No paths, just types
openapi: 3.0.3
info:
  title: My Application Types
  version: 1.0.0

# No paths section = No HTTP server
# But types are still available!

components:
  schemas:
    Config:
      type: object
      properties:
        port:
          type: integer
        host:
          type: string
```

openapi.yaml	paths	components	HTTP Server	Types Available
Missing	-	-	No	Primitives only
Present	Empty	Has schemas	No	Primitives + Schemas
Present	Has routes	Has schemas	Yes	Primitives + Schemas

Type Checking

Assignment Compatibility

From	To	Allowed
`T`	`T`	Yes
`Integer`	`Float`	Yes (widening)
`Float`	`Integer`	Warning (narrowing)
`List<T>`	`List<T>`	Yes
Schema	Same Schema	Yes

Type Errors

Error	Message
Type mismatch	`Expected 'String', got 'Integer'`
Unknown schema	`Schema 'Foo' not found in openapi.yaml`
Missing field	`Schema 'User' has no field 'age'`

Summary

Concept	Details
Primitives	`String` , `Integer` , `Float` , `Boolean`
Collections	`List<T>` , `Map<K, V>`
Complex types	Defined in `openapi.yaml` components/schemas
Optionals	None - values exist or operations fail
Type annotations	`<name: Type>`
Type inference	From literals and expressions

* * *

Next: Appendix A — Action Reference

§ Appendix A: Action Reference

Complete reference for all built-in actions in ARO.

Quick Reference Table

Action	Role	Description	Example
Extract	REQUEST	Pull data from structured source	`<Extract> the <id> from the <request: params>.`
Retrieve	REQUEST	Fetch from repository	`<Retrieve> the <user> from the <users> where id = <id>.`
Request	REQUEST	Make HTTP request	`<Request> the <data> from the <api-url>.`
Fetch	REQUEST	Fetch from external API	`<Fetch> the <weather> from <WeatherAPI: GET /forecast>.`
Read	REQUEST	Read from file	`<Read> the <config> from the <file: "./config.json">.`
List	REQUEST	List directory contents	`<List> the <files> from the <directory: src-path>.`
Stat	REQUEST	Get file metadata	`<Stat> the <info> for the <file: "./doc.pdf">.`
Exists	REQUEST	Check file existence	`<Exists> the <found> for the <file: "./config.json">.`
Receive	REQUEST	Receive event data	`<Receive> the <message> from the <event>.`
Exec	REQUEST	Execute shell command	`<Exec> the <result> for the`

Action	Role	Description	Example
			`<command> with "ls -la".`
Create	OWN	Create new data	`<Create> the <user> with { name: "Alice" }.`
Compute	OWN	Perform calculations	`<Compute> the <total> for the <items>.`
Transform	OWN	Convert/map data	`<Transform> the <dto> from the <entity>.`
Validate	OWN	Check against rules	`<Validate> the <data> for the <schema>.`
Compare	OWN	Compare values	`<Compare> the <hash> against the <stored>.`
Update	OWN	Modify existing data	`<Update> the <user> with <changes>.`
CreateDirectory	OWN	Create directory	`<CreateDirectory> the <dir> to the <path: "./out">.`
Copy	OWN	Copy file/directory	`<Copy> the <file: "./a.txt"> to the <destination: "./b.txt">.`
Move	OWN	Move/rename file	`<Move> the <file: "./old.txt"> to the <destination: "./new.txt">.`
Map	OWN	Transform collection elements	`<Map> the <names> from the <users: name>.`

Action	Role	Description	Example
Filter	OWN	Select matching elements	`<Filter> the <active> from the <users> where status = "active".`
Reduce	OWN	Aggregate collection	`<Reduce> the <total> from the <items> with sum(<amount>).`
Sort	OWN	Order collection	`<Sort> the <users> by <name>.`
Split	OWN	Split string by regex	`<Split> the <parts> from the <string> by /,/.`
Merge	OWN	Combine data	`<Merge> the <existing-user> with <update-data>.`
Return	RESPONSE	Return result	`<Return> an <OK: status> with <data>.`
Throw	RESPONSE	Throw error	`<Throw> a <NotFound: error> for the <user>.`
Log	EXPORT	Write to logs	`<Log> the <msg> for the <console> with "Done".`
Store	EXPORT	Save to repository	`<Store> the <user> into the <users>.`
Write	EXPORT	Write to file	`<Write> the <data> to the <file: "./out.txt">.`
Append	EXPORT	Append to file	`<Append> the <line> to the`

Action	Role	Description	Example
			`<file: "./log.txt">.`
Send	EXPORT	Send to destination	`<Send> the <email> to the <recipient>.`
Emit	EXPORT	Emit domain event	`<Emit> a <UserCreated: event> with <user>.`
Publish	EXPORT	Make globally available	`<Publish> as <config> <settings>.`
Notify	EXPORT	Send notification	`<Notify> the <alert> to the <admin>.`
Delete	EXPORT	Remove data	`<Delete> the <user> from the <users> where id = <id>.`
Start	SERVICE	Start a service	`<Start> the <http-server> with <contract>.`
Stop	SERVICE	Stop a service	`<Stop> the <http-server> with <application>.`
Listen	SERVICE	Listen for connections	`<Listen> on port 9000 as <socket-server>.`
Connect	SERVICE	Connect to service	`<Connect> to <host: "db"> on port 5432.`
Close	SERVICE	Close connection	`<Close> the <connection>.`

Action	Role	Description	Example
Broadcast	SERVICE	Send to all connections	`<Broadcast> the <msg> to the <server>.`
Route	SERVICE	Define HTTP route	`<Route> the <handler> for "/api/users".`
Keepalive	SERVICE	Keep app running	`<Keepalive> the <app> for the <events>.`
Call	SERVICE	Call external API	`<Call> the <result> via <API: POST /users>.`
Accept	STATE	Accept state transition	`<Accept> the <order: placed>.`
Given	TEST	Test precondition	`<Given> the <user> with { name: "Test" }.`
When	TEST	Test action	`<When> the <action> is performed.`
Then	TEST	Test expectation	`<Then> the <result> should be <expected>.`
Assert	TEST	Assert condition	`<Assert> the <value> equals <expected>.`

Action Categories

Category	Role	Data Flow
REQUEST	Bring data in	External -> Internal
OWN	Transform data	Internal -> Internal
RESPONSE	Send results	Internal -> External
EXPORT	Publish/persist	Internal -> External
SERVICE	Control services	System operations
STATE	State transitions	Internal state changes
TEST	Testing	Verification actions

* * *

REQUEST Actions

Extract

Pulls data from a structured source.

Syntax:

```
<Extract> the <result> from the <source: property>.
<Extract> the <result: specifier> from the <list>.
```

Examples:

```
<Extract> the <user-id> from the <request: parameters>.
<Extract> the <body> from the <request: body>.
<Extract> the <token> from the <request: headers.Authorization>.
<Extract> the <email> from the <user: email>.
<Extract> the <order> from the <event: order>.
```

List Element Access (ARO-0038):

Extract specific elements from arrays using result specifiers:

```
(* Keywords *)
<Extract> the <item: first> from the <list>.
<Extract> the <item: last> from the <list>.

(* Numeric index: 0 = last, 1 = second-to-last *)
<Extract> the <item: 0> from the <list>.
<Extract> the <item: 1> from the <list>.

(* Range: elements 2, 3, 4, 5 *)
<Extract> the <subset: 2-5> from the <list>.

(* Pick: elements at indices 0, 3, 7 *)
<Extract> the <selection: 0,3,7> from the <list>.
```

Specifier	Returns
`first`	First element
`last`	Last element
`0`	Last element (reverse indexing)
`n`	Element at (count - 1 - n)
`2-5`	Array of elements
`0,3,7`	Array of elements

Valid Prepositions: `from`

* * *

Retrieve

Fetches data from a repository.

Syntax:

```
<Retrieve> the <result> from the <repository> [where <condition>].
```

Examples:

```
<Retrieve> the <user> from the <user-repository>.
<Retrieve> the <user> from the <user-repository> where id = <user-id>.
<Retrieve> the <orders> from the <order-repository> where status = "pending".
<Retrieve> the <products> from the <repository> where category = <cat> and active = true.
```

Valid Prepositions: `from`

Fetch

Makes HTTP requests to external APIs.

Syntax:

```
<Fetch> the <result> from <url-or-api-reference>.
```

Examples:

```
<Fetch> the <data> from "https://api.example.com/resource".
<Fetch> the <users> from <UserAPI: GET /users>.
<Fetch> the <weather> from <WeatherAPI: GET /forecast?city=${city}>.
```

Valid Prepositions: `from`

Request

Makes HTTP requests to external URLs or APIs.

Syntax:

```
<Request> the <result> from <url>.              (* GET request *)
<Request> the <result> to <url> with <data>.    (* POST request *)
<Request> the <result> via METHOD <url>.        (* Explicit method *)
```

Examples:

```
(* GET request *)
<Create> the <api-url> with "https://api.open-meteo.com/v1/forecast".
<Request> the <weather> from the <api-url>.

(* POST request *)
<Create> the <user-data> with { name: "Alice", email: "alice@example.com" }.
<Request> the <result> to the <api-url> with <user-data>.

(* PUT/DELETE/PATCH via explicit method *)
<Request> the <result> via PUT the <url> with <update-data>.
<Request> the <result> via DELETE the <url>.
```

Response Metadata: After a request, these variables are available: - `result` - Parsed response body (JSON as map/list, or string) - `result.statusCode` - HTTP status code (e.g., 200, 404) - `result.headers` - Response headers as map - `result.isSuccess` - Boolean: true if status 200-299

Valid Prepositions: `from` , `to` , `via`

* * *

Read

Reads from files.

Syntax:

```
<Read> the <result> from the <file: path>.
<Read> the <result: type> from the <file: path>.
```

Examples:

```
<Read> the <content> from the <file: "./data.txt">.
<Read> the <config: JSON> from the <file: "./config.json">.
<Read> the <image: bytes> from the <file: "./logo.png">.
```

Valid Prepositions: `from`

* * *

Exec

Executes shell commands on the host system and returns structured results.

Syntax:

```
<Exec> the <result> for the <command> with "command-string".
<Exec> the <result> for the <command> with <variable>.
<Exec> the <result> on the <system> with {
    command: "command-string",
    workingDirectory: "/path",
    timeout: 30000
}.
```

Result Object: The Exec action returns a structured result with the following fields: - `result.error` - Boolean: true if command failed (non-zero exit code) - `result.message` - Human-readable status message - `result.output` - Command stdout (or stderr if error) - `result.exitCode` - Process exit code (0 = success, -1 = timeout) - `result.command` - The executed command string

Examples:

```
(* Basic command execution *)
<Exec> the <listing> for the <command> with "ls -la".
<Return> an <OK: status> for the <listing>.

(* With error handling *)
<Exec> the <result> for the <disk-check> with "df -h".
<Log> the <error> for the <console> with <result.message> when <result.error> = true.
<Return> an <Error: status> for the <result> when <result.error> = true.
<Return> an <OK: status> for the <result>.

(* Using a variable for the command *)
<Create> the <cmd> with "ps aux | head -20".
<Exec> the <processes> for the <listing> with <cmd>.

(* With configuration options *)
<Exec> the <result> on the <system> with {
    command: "npm install",
    workingDirectory: "/app",
    timeout: 60000
}.
```

Configuration Options: When using object syntax, these options are available: - `command` (required) - The shell command to execute - `workingDirectory` - Working directory (default:

current) - `timeout` - Timeout in milliseconds (default: 30000) - `shell` - Shell to use (default: /bin/sh) - `environment` - Additional environment variables as object

Security Note: Be cautious when constructing commands from user input. Always validate and sanitize input to prevent command injection.

Valid Prepositions: `for` , `on` , `with`

OWN Actions

Create

Creates new data structures.

Syntax:

```
<Create> the <result> with <data>.
```

Examples:

```
<Create> the <user> with <user-data>.
<Create> the <response> with { message: "Success" }.
<Create> the <order> with {
    items: <items>,
    total: <total>,
    customer: <customer-id>
}.
```

Valid Prepositions: `with`

Compute

Performs calculations.

Syntax:

```
<Compute> the <result> for the <input>.
<Compute> the <result> from <expression>.
```

Examples:

```
<Compute> the <total> for the <items>.
<Compute> the <hash> for the <password>.
<Compute> the <tax> for the <subtotal>.
<Compute> the <sum> from <a> + <b>.
```

Valid Prepositions: `for` , `from`

* * *

Transform

Converts or maps data.

Syntax:

```
<Transform> the <result> from the <source>.
<Transform> the <result> from the <source> with <modifications>.
```

Examples:

```
<Transform> the <dto> from the <entity>.
<Transform> the <updated-user> from the <user> with <updates>.
<Transform> the <response> from the <data>.
```

Valid Prepositions: `from`

* * *

Validate

Checks data against rules.

Syntax:

```
<Validate> the <data> for the <schema>.
```

Examples:

```
<Validate> the <user-data> for the <user-schema>.
<Validate> the <email> for the <email-pattern>.
<Validate> the <order> for the <order-rules>.
```

Valid Prepositions: `for`

Compare

Compares two values.

Syntax:

```
<Compare> the <value1> against the <value2>.
```

Examples:

```
<Compare> the <password-hash> against the <stored-hash>.
<Compare> the <signature> against the <expected>.
```

Valid Prepositions: `against`

Split

Splits a string into an array of parts using a regex delimiter.

Syntax:

```
<Split> the <parts> from the <string> by /delimiter/.
```

Examples:

```
(* Split CSV line by comma *)
<Create> the <csv-line> with "apple,banana,cherry".
<Split> the <fruits> from the <csv-line> by /,/.
(* fruits = ["apple", "banana", "cherry"] *)

(* Split by whitespace *)
<Split> the <words> from the <sentence> by /\s+/.

(* Split by multiple delimiters *)
<Split> the <tokens> from the <code> by /[;,\s]+/.

(* Case-insensitive split *)
<Split> the <sections> from the <text> by /SECTION/i.
```

Behavior: - Returns an array of strings between delimiter matches - If no match is found, returns original string as single-element array - Empty strings included when delimiters are adjacent - Supports regex flags: `i` (case-insensitive), `s` (dotall), `m` (multiline)

Valid Prepositions: `from` (with `by` clause)

Merge

Combines two data structures together. The source values are merged into the target, with source values overwriting target values for matching keys.

Syntax:

```
<Merge> the <target> with <source>.
<Merge> the <target> from <source>.
```

Examples:

```
(* Merge update data into existing entity *)
<Retrieve> the <existing-user> from the <user-repository> where id = <id>.
<Extract> the <update-data> from the <request: body>.
<Merge> the <existing-user> with <update-data>.
<Store> the <existing-user> into the <user-repository>.

(* Combine configuration objects *)
<Merge> the <defaults> with <overrides>.

(* Concatenate arrays *)
<Merge> the <all-items> with <new-items>.

(* Concatenate strings *)
<Merge> the <greeting> with <name>.
```

Supported Types: - **Dictionaries**: Source keys overwrite target keys; other target keys preserved - **Arrays**: Source elements appended to target array - **Strings**: Source string concatenated to target string

Valid Prepositions: `with` , `into` , `from`

RESPONSE Actions

Return

Returns a result with status.

Syntax:

```
<Return> [article] <status> [with <data>] [for <context>].
```

Examples:

```
<Return> an <OK: status> with <data>.
<Return> a <Created: status> with <resource>.
<Return> a <NoContent: status> for the <deletion>.
<Return> a <BadRequest: status> with <errors>.
<Return> a <NotFound: status> for the <missing: user>.
```

Valid Prepositions: `with` , `for`

Throw

Throws an error.

Syntax:

```
<Throw> [article] <error-type> for the <context>.
```

Examples:

```
<Throw> a <ValidationError> for the <invalid: input>.
<Throw> a <NotFoundError> for the <missing: user>.
<Throw> an <AuthenticationError> for the <invalid: token>.
```

Valid Prepositions: `for`

EXPORT Actions

Store

Saves to a repository.

Syntax:

```
<Store> the <data> into the <repository>.
```

Examples:

```
<Store> the <user> into the <user-repository>.
<Store> the <order> into the <order-repository>.
<Store> the <log-entry> into the <file: "./app.log">.
```

Valid Prepositions: `into`

Publish

Makes variables globally available.

Syntax:

```
<Publish> as <alias> <variable>.
```

Examples:

```
<Publish> as <app-config> <config>.
<Publish> as <current-user> <user>.
```

Valid Prepositions: `as`

Log

Writes to logs.

Syntax:

```
<Log> the <message-type> for the <destination> with <content>.
```

Examples:

```
<Log> the <message> for the <console> with "User logged in".
<Log> the <error: message> for the <console> with <error>.
<Log> the <audit: entry> for the <audit-log> with <details>.
```

Valid Prepositions: `for` , `with`

Send

Sends data to external destinations.

Syntax:

```
<Send> the <data> to the <destination>.
<Send> the <data> to the <destination> with <content>.
```

Examples:

```
<Send> the <email> to the <user: email>.
<Send> the <notification> to the <push-service>.
<Send> the <data> to the <connection>.
<Send> the <message> to the <connection> with "Hello".
```

Valid Prepositions: `to` , `with`

Emit

Emits domain events.

Syntax:

```
<Emit> [article] <event-type: event> with <data>.
```

Examples:

```
<Emit> a <UserCreated: event> with <user>.
<Emit> an <OrderPlaced: event> with <order>.
<Emit> a <PaymentProcessed: event> with <payment>.
```

Valid Prepositions: `with`

Write

Writes to files.

Syntax:

```
<Write> the <data> to the <file: path>.
```

Examples:

```
<Write> the <content> to the <file: "./output.txt">.
<Write> the <data: JSON> to the <file: "./data.json">.
```

Valid Prepositions: `to`

Append

Appends content to a file.

Syntax:

```
<Append> the <data> to the <file: path>.
```

Examples:

```
<Append> the <log-line> to the <file: "./logs/app.log">.
<Append> the <entry> to the <file: "./data.txt">.
```

Valid Prepositions: `to` , `into`

List

Lists directory contents.

Syntax:

```
<Create> the <dir-path> with "./path".
<List> the <result> from the <directory: dir-path>.
<List> the <result> from the <directory: dir-path> matching "pattern".
<List> the <result> from the <directory: dir-path> recursively.
```

Examples:

```
<Create> the <uploads-path> with "./uploads".
<List> the <entries> from the <directory: uploads-path>.
<List> the <aro-files> from the <directory: src-path> matching "*.aro".
<List> the <all-files> from the <directory: project-path> recursively.
```

Valid Prepositions: `from`

Stat

Gets file or directory metadata.

Syntax:

```
<Stat> the <result> for the <file: path>.
<Stat> the <result> for the <directory: path>.
```

Examples:

```
<Stat> the <info> for the <file: "./document.pdf">.
<Stat> the <dir-info> for the <directory: "./src">.
```

Result Properties: - `name` - file or directory name - `path` - full path - `size` - size in bytes - `isFile` - true if file - `isDirectory` - true if directory - `created` - creation date (ISO 8601) - `modified` - modification date (ISO 8601) - `permissions` - Unix-style permissions

Valid Prepositions: `for`

Exists

Checks if a file or directory exists.

Syntax:

```
<Exists> the <result> for the <file: path>.
<Exists> the <result> for the <directory: path>.
```

Examples:

```
<Exists> the <found> for the <file: "./config.json">.
<Exists> the <dir-exists> for the <directory: "./output">.
```

Valid Prepositions: `for`

CreateDirectory

Creates a directory with all intermediate directories.

Syntax:

```
<CreateDirectory> the <result> to the <path: path>.
```

Examples:

```
<CreateDirectory> the <output-dir> to the <path: "./output/reports/2024">.
```

Valid Prepositions: `to` , `for`

Copy

Copies files or directories.

Syntax:

```
<Copy> the <file: source> to the <destination: dest>.
<Copy> the <directory: source> to the <destination: dest>.
```

Examples:

```
<Copy> the <file: "./template.txt"> to the <destination: "./copy.txt">.
<Copy> the <directory: "./src"> to the <destination: "./backup/src">.
```

Valid Prepositions: `to`

Move

Moves or renames files and directories.

Syntax:

```
<Move> the <file: source> to the <destination: dest>.
<Move> the <directory: source> to the <destination: dest>.
```

Examples:

```
<Move> the <file: "./draft.txt"> to the <destination: "./final.txt">.
<Move> the <directory: "./temp"> to the <destination: "./processed">.
```

Valid Prepositions: `to`

Delete

Removes data.

Syntax:

```
<Delete> the <target> from the <source> [where <condition>].
<Delete> the <file: path>.
```

Examples:

```
<Delete> the <user> from the <user-repository> where id = <user-id>.
<Delete> the <file: "./temp.txt">.
<Delete> the <sessions> from the <repository> where expired = true.
```

Valid Prepositions: `from`

SERVICE Actions

Start

Starts a service. All services use the standardized `with` preposition.

Syntax:

```
(* HTTP server from OpenAPI contract *)
<Start> the <http-server> with <contract>.

(* Socket server with port configuration *)
<Start> the <socket-server> with { port: 9000 }.

(* File monitor with directory path *)
<Start> the <file-monitor> with ".".
<Start> the <file-monitor> with { directory: "./data" }.
```

Examples:

```
<Start> the <http-server> with <contract>.
<Start> the <socket-server> with { port: 9000 }.
<Start> the <file-monitor> with "./uploads".
```

Valid Prepositions: `with`

* * *

Stop

Stops a service gracefully.

Syntax:

```
<Stop> the <service> with <application>.
```

Examples:

```
<Stop> the <http-server> with <application>.
<Stop> the <socket-server> with <application>.
<Stop> the <file-monitor> with <application>.
```

Valid Prepositions: `with`

* * *

Listen

Listens for connections.

Syntax:

```
<Listen> on port <number> as <name>.
```

Examples:

```
<Listen> on port 9000 as <socket-server>.
```

Valid Prepositions: `on` , `as`

* * *

Connect

Connects to a service.

Syntax:

```
<Connect> to <host: address> on port <number> as <name>.
```

Examples:

```
<Connect> to <host: "localhost"> on port 5432 as <database>.
<Connect> to <host: "redis.local"> on port 6379 as <cache>.
```

Valid Prepositions: `to` , `on` , `as`

* * *

Close

Closes connections.

Syntax:

```
<Close> the <connection>.
```

Examples:

```
<Close> the <database-connections>.
<Close> the <socket-server>.
<Close> the <connection>.
```

Valid Prepositions: None

Call

Makes API calls.

Syntax:

```
<Call> the <result> via <api-reference> [with <data>].
```

Examples:

```
<Call> the <result> via <UserAPI: POST /users> with <user-data>.
<Call> the <response> via <PaymentAPI: POST /charge> with <payment>.
```

Valid Prepositions: `via` , `with`

Broadcast

Sends to all connections.

Syntax:

```
<Broadcast> the <message> to the <server>.
```

Examples:

```
<Broadcast> the <message> to the <socket-server>.
<Broadcast> the <notification> to the <chat-server> with "User joined".
```

Valid Prepositions: `to` , `with`

Keepalive

Keeps a long-running application alive to process events.

Syntax:

```
<Keepalive> the <application> for the <events>.
```

Description: The `Keepalive` action blocks execution until a shutdown signal is received (SIGINT/SIGTERM). This is essential for applications that need to stay alive and process events, such as HTTP servers, file watchers, and socket servers.

Examples:

```
(* HTTP server auto-starts when openapi.yaml is present *)
(Application-Start: My API) {
    <Log> the <startup: message> for the <console> with "API starting...".
    <Keepalive> the <application> for the <events>.
    <Return> an <OK: status> for the <startup>.
}

(Application-Start: File Watcher) {
    <Watch> the <directory> for the <changes> with "./watched".
    <Keepalive> the <application> for the <events>.
    <Return> an <OK: status> for the <startup>.
}
```

Valid Prepositions: `for`

* * *

Action Summary Table

Action	Role	Prepositions
Extract	REQUEST	from
Retrieve	REQUEST	from
Request	REQUEST	from, to, via
Fetch	REQUEST	from
Read	REQUEST	from
Receive	REQUEST	from
Exec	REQUEST	for, on, with
Create	OWN	with
Compute	OWN	for, from
Transform	OWN	from
Validate	OWN	for
Compare	OWN	against
Update	OWN	with
Map	OWN	from
Filter	OWN	from, where
Reduce	OWN	from, with
Sort	OWN	by
Split	OWN	from (by clause)
Merge	OWN	with, into, from
Return	RESPONSE	with, for
Throw	RESPONSE	for
Store	EXPORT	into
Publish	EXPORT	as
Log	EXPORT	for, with

Action	Role	Prepositions
Send	EXPORT	to, with
Emit	EXPORT	with
Write	EXPORT	to
Delete	EXPORT	from
Notify	EXPORT	to
List	FILE	from
Stat	FILE	for
Exists	FILE	for
CreateDirectory	FILE	to
Copy	FILE	to
Move	FILE	to
Append	FILE	to
Start	SERVICE	with
Stop	SERVICE	with
Listen	SERVICE	on, as
Connect	SERVICE	to, on, as
Close	SERVICE	-
Route	SERVICE	for
Call	SERVICE	via, with
Broadcast	SERVICE	to, with
Keepalive	SERVICE	for
Accept	STATE	-
Given	TEST	with
When	TEST	-

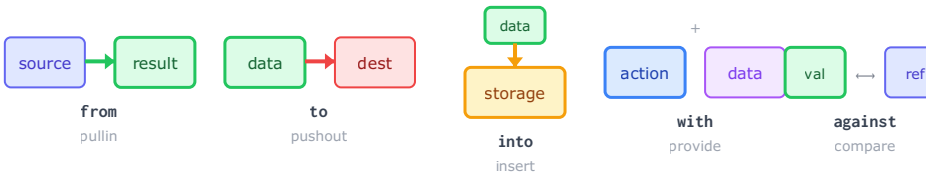
Action	Role	Prepositions
Then	TEST	-
Assert	TEST	equals, contains

§ Appendix B: Preposition Semantics

> How prepositions shape meaning in ARO.

* * *

Overview



ARO uses eight prepositions, each with specific semantic meaning:

Preposition	Primary Meaning	Data Flow
`from`	Source/extraction	External → Internal
`with`	Accompaniment/using	Provides data
`for`	Purpose/target	Indicates beneficiary
`to`	Destination	Internal → External
`into`	Insertion/transformation	Internal → Storage
`against`	Comparison/validation	Reference point
`via`	Through/medium	Intermediate channel
`on`	Location/surface	Attachment point

* * *

from

Meaning: Source extraction — data flows from an external source inward.

Indicates: The origin of data being pulled into the current context.

Common with: `Extract` , `Retrieve` , `Fetch` , `Read` , `Receive`

Examples

```
(* Extract from request context *)
<Extract> the <user-id> from the <pathParameters: id>.
<Extract> the <body> from the <request: body>.
<Extract> the <token> from the <headers: authorization>.

(* Retrieve from repository *)
<Retrieve> the <user> from the <user-repository>.
<Retrieve> the <orders> from the <order-repository> where <status> is "active".

(* Fetch from external URL *)
<Fetch> the <data> from "https://api.example.com/users".

(* Read from file *)
<Read> the <config> from the <file> with "config.json".

(* Filter from collection *)
<Filter> the <active> from the <users> where <status> is "active".
```

Semantic Notes

- `from` typically indicates an external or persistent source
- Used when data crosses a boundary into the current scope
- The preposition signals that the action is “pulling” data

* * *

with

Meaning: Accompaniment — data provided alongside or used by the action.

Indicates: Additional data, parameters, or configuration.

Common with: `Create` , `Return` , `Emit` , `Merge` , `Log`

Examples

```
(* Create with data *)
<Create> the <user> with <user-data>.
<Create> the <greeting> with "Hello, World!".
<Create> the <total> with <subtotal> + <tax>.

(* Return with payload *)
<Return> an <OK: status> with <users>.
<Return> a <Created: status> with <user>.

(* Emit with event data *)
<Emit> a <UserCreated: event> with <user>.
<Emit> an <OrderPlaced: event> with { orderId: <id>, total: <total> }.

(* Merge with updates *)
<Merge> the <updated> from <existing> with <changes>.

(* Log with message *)
<Log> the <message> for the <console> with "Application started".

(* Read with path *)
<Read> the <content> from the <file> with "data.json".
```

Semantic Notes

- `with` provides the data or value to use
- Often specifies literal values, expressions, or object references
- Indicates “using this” rather than “from this”

---

for

Meaning: Purpose/target — indicates the beneficiary or purpose.

Indicates: What the action is intended for or aimed at.

Common with: `Return` , `Log` , `Compute` , `Validate`

Examples

```
(* Return for a target *)
<Return> an <OK: status> for the <request>.
<Return> a <NoContent: status> for the <deletion>.

(* Log for a destination *)
<Log> the <message> for the <console>.
<Log> the <error> for the <error-log>.

(* Compute for an input *)
<Compute> the <total> for the <items>.
<Compute> the <hash> for the <password>.

(* Validate for a type *)
<Validate> the <input> for the <user-type>.
```

Semantic Notes

- `for` indicates purpose or beneficiary
- Often used with logging and return statements
- Specifies “on behalf of” or “intended for”

---

to

Meaning: Destination — data flows outward to a target.

Indicates: The endpoint or recipient of data.

Common with: `Send` , `Write` , `Connect`

Examples

```
(* Send to destination *)
<Send> the <email> to the <user: email>.
<Send> the <notification> to the <admin>.
<Send> the <request> to "https://api.example.com/webhook".

(* Write to file *)
<Write> the <data> to the <file> with "output.json".

(* Connect to service *)
<Connect> the <database> to "postgres://localhost/mydb".
<Connect> the <socket> to "localhost:9000".
```

Semantic Notes

- `to` indicates outward data flow
- Used when sending or directing data to an external destination
- Opposite direction from `from`

---

into

Meaning: Insertion/transformation — data enters or transforms.

Indicates: A container or new form for the data.

Common with: `Store` , `Transform`

Examples

```
(* Store into repository *)
<Store> the <user> into the <user-repository>.
<Store> the <order> into the <order-repository>.
<Store> the <cache-entry> into the <cache>.

(* Transform into format *)
<Transform> the <dto> into the <json>.
<Transform> the <entity> into the <response-model>.
```

Semantic Notes

- `into` suggests insertion or transformation
- Used for persistence and format conversion
- Implies the data “enters” something

---

against

Meaning: Comparison/validation — data is checked against a reference.

Indicates: The standard or rule for comparison.

Common with: `Validate` , `Compare`

Examples

```
(* Validate against schema *)
<Validate> the <input> against the <user: schema>.
<Validate> the <password> against the <password-rules>.
<Validate> the <token> against the <auth-service>.

(* Compare against reference *)
<Compare> the <old-value> against the <new-value>.
<Compare> the <actual> against the <expected>.
```

Semantic Notes

- `against` implies testing or comparison
- Used for validation, verification, and comparison
- The object is the reference standard

---

via

Meaning: Through/medium — indicates an intermediate channel.

Indicates: The pathway or method used.

Common with: `Fetch`, `Send`

Examples

```
(* Fetch via proxy *)
<Fetch> the <data> from "https://api.example.com" via the <proxy>.

(* Send via channel *)
<Send> the <message> to the <user> via the <email-service>.
<Send> the <notification> to the <subscriber> via the <websocket>.
```

Semantic Notes

- `via` indicates an intermediate hop or method
- Less common than other prepositions
- Used when specifying how data travels

---

on

Meaning: Location/surface — indicates attachment or location.

Indicates: The point of attachment or surface.

Common with: `Start`, `Serve`

Examples

```
(* Start on port *)
<Start> the <http-server> on port 8080.
<Start> the <socket-server> on port 9000.

(* Serve on host *)
<Start> the <http-server> on "0.0.0.0:8080".
```

Semantic Notes

- `on` specifies a location or attachment point
- Primarily used for network configuration
- Indicates “located at” or “attached to”

* * *

Preposition Selection Guide

Intent	Preposition	Example
Pull data in	`from`	`<Extract> the <x> from the <y>`
Provide data	`with`	`<Create> the <x> with <y>`
Indicate purpose	`for`	`<Return> the <x> for the <y>`
Push data out	`to`	`<Send> the <x> to the <y>`
Store/transform	`into`	`<Store> the <x> into the <y>`
Compare/validate	`against`	`<Validate> the <x> against the <y>`
Specify channel	`via`	`<Fetch> the <x> via the <y>`
Specify location	`on`	`<Start> the <x> on <y>`

* * *

External Source Indicators

Some prepositions indicate external sources:

```
// From Token.swift
public var indicatesExternalSource: Bool {
    switch self {
    case .from, .via: return true
    default: return false
    }
}
```

The `from` and `via` prepositions signal that data is coming from outside the current context, which affects semantic analysis and data flow tracking.

§ Appendix C: Grammar Specification

This appendix provides the complete formal grammar specification for ARO using Extended Backus-Naur Form (EBNF).

Notation

Symbol	Meaning
`=`	Definition
`,`	Concatenation
`|`	Alternative
`[ ]`	Optional (0 or 1)
`{ }`	Repetition (0 or more)
`( )`	Grouping
`" "`	Terminal string
`' '`	Terminal character
`(* *)`	Comment

Program Structure

```
(* Top-level program *)
program = { feature_set } ;

(* Feature set definition *)
feature_set = "(" , feature_name , ":" , business_activity , ")" , block ;

feature_name = identifier | route_pattern ;
business_activity = identifier , { identifier } ;

(* Route patterns for HTTP handlers *)
route_pattern = http_method , route_path ;
http_method = "GET" | "POST" | "PUT" | "DELETE" | "PATCH" ;
route_path = "/" , { path_segment , "/" } , [ path_segment ] ;
path_segment = identifier | path_parameter ;
path_parameter = "{" , identifier , "}" ;

(* Block of statements *)
block = "{" , { statement } , "}" ;
```

Statements

```
(* Statement types *)
statement = aro_statement
          | guarded_statement
          | publish_statement
          | match_statement ;

(* Core ARO statement: Action-Result-Object *)
aro_statement = action , [ article ] , result , preposition , [ article ] , object , [ modifiers ]

(* Publish statement *)
publish_statement = "<Publish>" , "as" , alias , variable , "." ;

(* Guarded statement - ARO statement with conditional suffix *)
guarded_statement = aro_statement_base , "when" , condition , "." ;
aro_statement_base = action , [ article ] , result , preposition , [ article ] , object , [ modif

(* Match statements *)
match_statement = "match" , variable , "{" , { match_case } , [ default_case ] , "}" ;
match_case = "case" , pattern , block ;
pattern = literal | regex_literal | variable ;
default_case = "default" , block | "otherwise" , block ;
```

Actions and Objects

```
(* Action - the verb *)
action = "<" , action_verb , ">" ;
action_verb = identifier ;

(* Result - what is produced *)
result = variable | typed_variable ;

(* Object - the source or target *)
object = variable
       | typed_variable
       | literal
       | file_reference
       | api_reference
       | repository_reference ;

(* Modifiers *)
modifiers = where_clause | with_clause | on_clause ;
where_clause = "where" , condition , { "and" , condition } ;
with_clause = "with" , ( variable | object_literal | literal ) ;
on_clause = "on" , "port" , number ;
```

Variables and Types

```
(* Variable forms *)
variable = "<" , identifier , ">" ;
typed_variable = "<" , identifier , ":" , type_hint , ">" ;
qualified_variable = "<" , identifier , ":" , qualifier , ">" ;

(* Type hints *)
type_hint = "JSON" | "bytes" | "List" | "String" | "Number" | "Boolean" | "Date" | identifier ;

(* Qualifier for accessing properties *)
qualifier = identifier , { identifier } ;

(* Alias for publishing *)
alias = "<" , identifier , ">" ;
```

References

```
(* File reference *)
file_reference = "<" , "file:" , ( string_literal | variable ) , ">" ;

(* Directory reference *)
directory_reference = "<" , "directory:" , string_literal , ">" ;

(* API reference *)
api_reference = "<" , api_name , ":" , [ http_method ] , route_path , ">" ;
api_name = identifier ;

(* Repository reference *)
repository_reference = "<" , identifier , "-repository" , ">" ;

(* Host reference *)
host_reference = "<" , "host:" , string_literal , ">" ;
```

Conditions

```
(* Condition expressions *)
condition = comparison | existence_check | boolean_condition ;

(* Comparisons *)
comparison = variable , comparison_op , ( variable | literal | regex_literal ) ;
comparison_op = "is" | "is not" | ">" | "<" | ">=" | "<=" | "matches" | "contains" ;

(* Existence checks *)
existence_check = variable , ( "is empty" | "is not empty" ) ;

(* Boolean combinations *)
boolean_condition = condition , boolean_op , condition ;
boolean_op = "and" | "or" ;

(* Negation *)
negation = "not" , condition ;
```

Literals

```
(* Literal values *)
literal = string_literal | number | boolean | object_literal ;

(* String literal *)
string_literal = '"' , { string_char | interpolation } , '"' ;
string_char = (* any character except " and $ *) | escape_sequence ;
escape_sequence = "\\" , ( '"' | "\\" | "n" | "t" | "r" ) ;
interpolation = "${" , ( identifier | qualified_variable ) , "}" ;

(* Number literal *)
number = [ "-" ] , digit , { digit } , [ "." , digit , { digit } ] ;

(* Boolean literal *)
boolean = "true" | "false" ;

(* Object literal *)
object_literal = "{" , [ property , { "," , property } ] , "}" ;
property = property_name , ":" , ( literal | variable ) ;
property_name = identifier | string_literal ;

(* Regex literal *)
regex_literal = "/" , regex_body , "/" , [ regex_flags ] ;
regex_body = { regex_char | regex_escape } ;
regex_char = (* any character except "/" and newline *) ;
regex_escape = "\\" , (* any character *) ;
regex_flags = { "i" | "s" | "m" | "g" } ;
```

Regex Flags

Flag	Description
`i`	Case insensitive matching
`s`	Dot matches newlines (dotall)
`m`	Multiline mode (^ and $ match line boundaries)
`g`	Global (reserved for future replace operations)

Lexical Elements

```
(* Identifier *)
identifier = letter , { letter | digit | "-" } ;

(* Article *)
article = "a" | "an" | "the" ;

(* Preposition *)
preposition = "from" | "to" | "for" | "with" | "into" | "against" | "via" | "on" | "as" ;

(* Basic character classes *)
letter = "a" | "b" | ... | "z" | "A" | "B" | ... | "Z" ;
digit = "0" | "1" | ... | "9" ;

(* Whitespace (ignored) *)
whitespace = " " | "\t" | "\n" | "\r" ;

(* Comment *)
comment = "(*" , { any_char } , "*)" ;
```

Special Feature Sets

```
(* Application lifecycle *)
application_start = "(" , "Application-Start" , ":" , business_activity , ")" , block ;
application_end_success = "(" , "Application-End" , ":" , "Success" , ")" , block ;
application_end_error = "(" , "Application-End" , ":" , "Error" , ")" , block ;

(* Event handlers *)
event_handler = "(" , handler_name , ":" , event_type , "Handler" , ")" , block ;
handler_name = identifier , { identifier } ;
event_type = identifier ;
```

API Definitions

```
(* API definition block *)
api_definition = "api" , identifier , "{" , { api_property } , "}" ;
api_property = property_name , ":" , ( string_literal | object_literal ) , ";" ;
```

Complete Examples

Minimal Program

```
program = feature_set
        = "(" , "Application-Start" , ":" , "Test" , ")" , block
        = "(" , "Application-Start" , ":" , "Test" , ")" , "{" , statement , "}"
        = "(" , "Application-Start" , ":" , "Test" , ")" , "{" , aro_statement , "}"
        = "(" , "Application-Start" , ":" , "Test" , ")" , "{" ,
            "<Return>" , "an" , "<OK: status>" , "for" , "the" , "<startup>" , "." ,
          "}"
```

ARO Statement Parse

```
"<Extract> the <user-id> from the <request: parameters>."

= aro_statement
= action , article , result , preposition , article , object , "."
= "<Extract>" , "the" , "<user-id>" , "from" , "the" , "<request: parameters>" , "."
```

Guarded Statement Parse

```
"<Return> a <NotFound: status> for the <user> when <user> is empty."

= guarded_statement
= aro_statement_base , "when" , condition , "."
= action , result , preposition , object , "when" , existence_check , "."
= "<Return>" , "a" , "<NotFound: status>" , "for" , "the" , "<user>" , "when" , "<user>" , "is em
```

Precedence

Operator precedence (highest to lowest):

1. Parentheses `( )`
2. `not`
3. Comparisons (`is` , `is not` , `>` , `<` , `>=` , `<=`)
4. `and`
5. `or`

Reserved Words

The following identifiers are reserved:

Articles: `a` , `an` , `the`

Prepositions: `from` , `to` , `for` , `with` , `into` , `against` , `via` , `on` , `as`

Control Flow: `when` , `match` , `case` , `default` , `otherwise` , `where` , `and` , `or` , `not` , `is`

Literals: `true` , `false` , `empty`

Status Codes: `OK` , `Created` , `Accepted` , `NoContent` , `BadRequest` , `Unauthorized` , `Forbidden` , `NotFound` , `Conflict` , `InternalError` , `ServiceUnavailable`

Special: `Application-Start` , `Application-End` , `Success` , `Error` , `Handler`

File Encoding

ARO source files must be encoded in UTF-8. The `.aro` file extension is required.

§ Appendix D: Statements Reference

This appendix provides a complete reference for all statement types in ARO.

ARO Statement

The fundamental statement type following the Action-Result-Object pattern.

Syntax

```
<Action> [article] <result> preposition [article] <object> [modifiers].
```

Components

Component	Required	Description
Action	Yes	Verb in angle brackets
Article	No	a, an, or the
Result	Yes	Output variable
Preposition	Yes	Relationship word
Object	Yes	Input/target
Modifiers	No	where, with, on, when clauses

Examples

```
<Extract> the <user-id> from the <request: parameters>.
<Create> a <user> with <user-data>.
<Return> an <OK: status> for the <request>.
<Store> the <order> into the <order-repository>.
<Retrieve> the <user> from the <repository> where id = <user-id>.
<Start> the <http-server> on port 8080.
```

Publish Statement

Makes a variable globally accessible across feature sets.

Syntax

```
<Publish> as <alias> <variable>.
```

Components

Component	Description
alias	Name to publish under
variable	Variable to publish

Example

```
<Read> the <config> from the <file: "./config.json">.
<Publish> as <app-config> <config>.
```

Guarded Statement (when)

Conditionally executes a statement based on a condition. If the condition is false, the statement is skipped.

Syntax

```
<Action> the <result> preposition the <object> when <condition>.
```

Conditions

Condition	Description
`<var> is <value>`	Equality
`<var> is not <value>`	Inequality
`<var> is empty`	Null/empty check
`<var> is not empty`	Has value
`<var> exists`	Value exists
`<var> is null`	Null check
`<var> > <value>`	Greater than
`<var> < <value>`	Less than
`<var> >= <value>`	Greater or equal
`<var> <= <value>`	Less or equal
`<cond1> and <cond2>`	Both true
`<cond1> or <cond2>`	Either true
`not <cond>`	Negation

Examples

```
(* Return not found only when user is empty *)
<Return> a <NotFound: status> for the <missing: user> when <user> is empty.

(* Send notification only when user has email *)
<Send> the <notification> to the <user: email> when <user: email> exists.

(* Log admin access only for admins *)
<Log> the <admin-access> for the <audit> when <user: role> = "admin".

(* Early exit on invalid input *)
<Return> a <BadRequest: status> for the <invalid: amount> when <amount> <= 0.

(* Combined conditions *)
<Grant> the <access> for the <user> when <user: active> is true and <user: verified> is true.
```

Usage Pattern

```
(PUT /users/{id}: User API) {
    <Extract> the <user-id> from the <request: parameters>.
    <Extract> the <updates> from the <request: body>.

    (* Early exit guards *)
    <Return> a <BadRequest: status> for the <missing: id> when <user-id> is empty.
    <Return> a <BadRequest: status> for the <missing: data> when <updates> is empty.

    (* Continue with valid input *)
    <Retrieve> the <user> from the <repository> where id = <user-id>.
    <Return> a <NotFound: status> for the <missing: user> when <user> is empty.

    <Transform> the <updated-user> from the <user> with <updates>.
    <Store> the <updated-user> into the <repository>.
    <Return> an <OK: status> with <updated-user>.
}
```

Match Statement

Pattern matching for multiple cases.

Syntax

```
match <variable> {
    case <value1> {
        (* statements *)
    }
    case <value2> {
        (* statements *)
    }
    otherwise {
        (* fallback statements *)
    }
}
```

Case with Guard

```
match <variable> {
    case <value> where <condition> {
        (* statements *)
    }
}
```

Example

```
match <status> {
    case "pending" {
        <Log> the <message> for the <console> with "Order is pending".
    }
    case "shipped" {
        <Log> the <message> for the <console> with "Order has shipped".
        <Emit> an <OrderShipped: event> with <order>.
    }
    case "delivered" {
        <Log> the <message> for the <console> with "Order delivered".
        <Emit> an <OrderDelivered: event> with <order>.
    }
    otherwise {
        <Log> the <warning> for the <console> with "Unknown status".
    }
}
```

Pattern Matching with Guards

```
match <user: subscription> {
    case <premium> where <user: credits> > 0 {
        <Grant> the <premium-features> for the <user>.
        <Deduct> the <credit> from the <user: account>.
    }
    case <premium> {
        <Notify> the <user> about the <low-credits>.
        <Grant> the <basic-features> for the <user>.
    }
    case <basic> {
        <Grant> the <basic-features> for the <user>.
    }
    otherwise {
        <Redirect> the <user> to the <subscription-page>.
    }
}
```

Return Statement

Exits the feature set with a response.

Syntax

```
<Return> [article] <status> [with <data>] [for <context>].
```

Status Codes

Status	HTTP Code	Usage
`OK`	200	Successful request
`Created`	201	Resource created
`Accepted`	202	Async operation started
`NoContent`	204	Success, no body
`BadRequest`	400	Invalid input
`Unauthorized`	401	Auth required
`Forbidden`	403	Access denied
`NotFound`	404	Not found
`Conflict`	409	Resource conflict
`UnprocessableEntity`	422	Validation failed
`TooManyRequests`	429	Rate limited
`InternalError`	500	Server error
`ServiceUnavailable`	503	Service down

Examples

```
<Return> an <OK: status> with <data>.
<Return> a <Created: status> with <resource>.
<Return> a <NoContent: status> for the <deletion>.
<Return> a <BadRequest: status> with <validation: errors>.
<Return> a <NotFound: status> for the <missing: user>.
<Return> a <Forbidden: status> for the <unauthorized: access>.
```

Comment

Adds documentation to code.

Syntax

```
(* comment text *)
```

Examples

```
(* This is a single-line comment *)

(*
   This is a
   multi-line comment
*)

(* Comments can be (* nested *) *)

(Process Order: Order Processing) {
    (* Extract order data from request *)
    <Extract> the <order-data> from the <request: body>.

    (* Validate before processing *)
    <Validate> the <order-data> for the <order-schema>.

    (* Store and return *)
    <Store> the <order> into the <repository>.
    <Return> a <Created: status> with <order>.
}
```

Where Clause

Filters data in retrieval and deletion.

Syntax

```
... where <field> = <value>
... where <field> = <value> and <field2> = <value2>
```

Examples

```
<Retrieve> the <user> from the <repository> where id = <user-id>.
<Retrieve> the <orders> from the <repository> where status = "pending".
<Retrieve> the <users> from the <repository> where role = "admin" and active = true.
<Delete> the <sessions> from the <repository> where userId = <user-id>.
```

With Clause

Provides additional data or parameters.

Syntax

```
... with <variable>
... with <object-literal>
... with <string-literal>
```

Examples

```
<Create> the <user> with <user-data>.
<Create> the <config> with { debug: true, port: 8080 }.
<Transform> the <updated> from the <user> with <updates>.
<Send> the <message> to the <connection> with "Hello, World!".
<Log> the <message> for the <console> with "Application started".
```

On Clause

Specifies ports for network operations.

Syntax

```
... on port <number>
```

Examples

```
<Start> the <http-server> on port 8080.
<Listen> on port 9000 as <socket-server>.
<Connect> to <host: "localhost"> on port 5432 as <database>.
```

When Clause

Conditionally executes a statement.

Syntax

```
<Action> the <result> preposition the <object> when <condition>.
```

Examples

```
<Return> a <NotFound: status> for the <user> when <user> is empty.
<Log> the <warning> for the <console> with "Low stock" when <stock> < 10.
<Send> the <alert> to the <admin: email> when <errors> > <threshold>.
```

Statement Order

Statements execute sequentially from top to bottom:

```
(Process Request: Handler) {
    (* 1. First *)
    <Extract> the <data> from the <request: body>.

    (* 2. Second *)
    <Validate> the <data> for the <schema>.

    (* 3. Third *)
    <Create> the <result> with <data>.

    (* 4. Fourth *)
    <Store> the <result> into the <repository>.

    (* 5. Fifth - ends execution *)
    <Return> a <Created: status> with <result>.

    (* Never executed - after return *)
    <Log> the <message> for the <console> with "This won't run".
}
```

Statement Termination

All statements end with a period (.):

```
(* Correct *)
<Extract> the <data> from the <request>.
<Return> an <OK: status> with <data>.

(* Incorrect - missing period *)
<Extract> the <data> from the <request>
```

Match blocks use braces without periods on closing brace:

```
match <status> {
    case "active" {
        <Return> an <OK: status>.  (* Period on inner statement *)
    }
}  (* No period on closing brace *)
```