Designing with Ports and Adapters

In this lesson, you will learn about the Ports and Adapters design pattern and how to implement it in applications.

WE'LL COVER THE FOLLOWING

- Ports and adapter design pattern
- Splitting the code

Next, you'll need to modify the `ShowFormFunction` Lambda function to validate the requested extension and return an error message in case of unsupported extensions. The function you wrote in the previous chapter was still relatively easy to read and understand. Putting more logic into that function definitely pushes it beyond the threshold of what could be considered simple.

A big part of making robust code is the ability to understand it easily and modify it with confidence. Both those goals are more attainable with some nice unit tests. However, your current function design is not really making that easy. The function directly talks to S3, requiring IAM privileges to execute, and it depends on several environment variables that need to be configured upfront. Automated tests for this function would be slow and error-prone and it would be difficult to set up all the testing dependencies. This would be a good time to redesign the code and prepare it for future evolution.

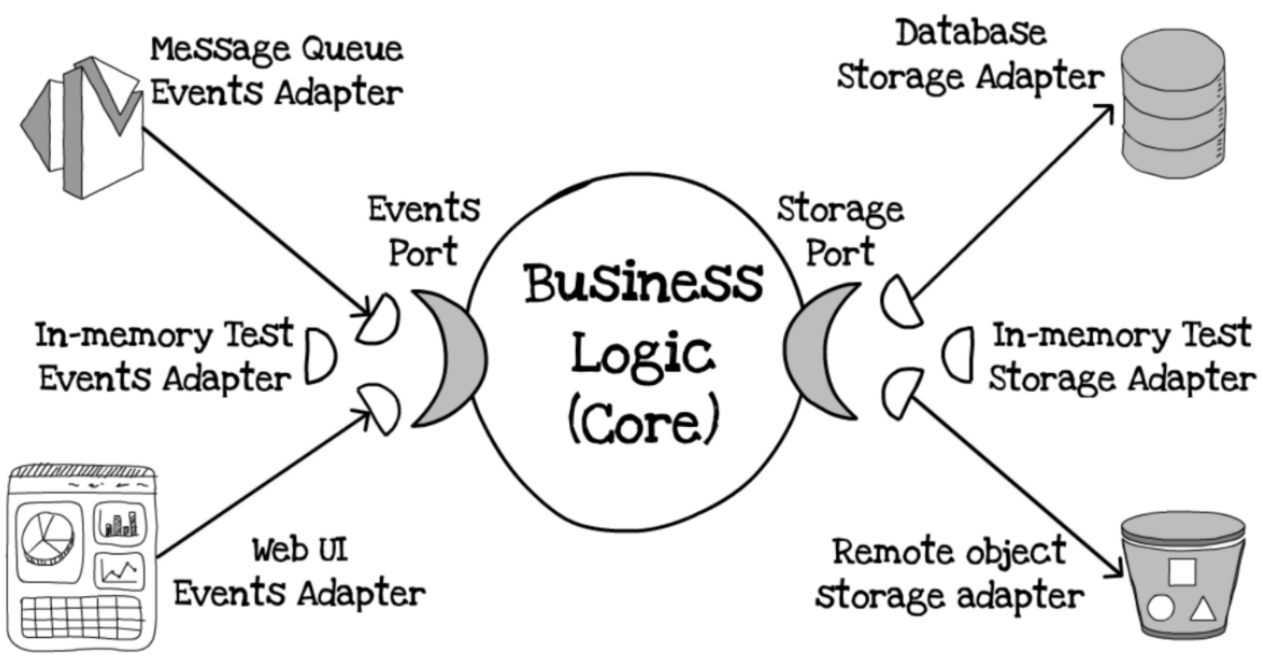
Ports and adapter design pattern

At MindMup, we use the *Ports and Adapters* design pattern (also called *hexagonal architecture*) for any but the simplest Lambda functions. This makes it easy to cover the code with automated tests at various levels effectively and evolve functions easily over time. The pattern was first described by Alistair Cockburn in 2005. It's very closely related to the idea of *Simplicators*, presented by Nat Pryce and Steve Freeman in *Growing Object-Oriented Software*, and the idea of *anti-corruption layers*, described by Eric

Oriented Software, and the idea of *anti-corruption layers*, described by Eric Evans in *Domain-Driven Design*.

The purpose of this design pattern, straight from its canonical description, is to ‘allow an application to equally be driven by users, programs, automated test or batch scripts, and to be developed and tested in isolation from its eventual runtime devices and databases’. Cockburn observed that when business logic gets entangled with external interactions, both become difficult to develop and test. This applies equally to user interface interactions (such as receiving requests from web pages) and to infrastructure interactions (for example saving to a database or acting on queue messages). Lambda code might not have any direct user interactions, but it is always triggered by infrastructure events and often needs to talk to other infrastructure, so this makes Lambda functions prone to the problems that Ports and Adapters addresses.

The design pattern suggests isolating the business logic into a core that has no external interactions but instead provides various interfaces for interactions in the figure given below. These interfaces are the *ports*, and they are described in the domain model of the business logic. For each type of interaction, you can implement the relevant port interface with an *adapter*, which translates from the core model into the specific infrastructure API. Test versions of adapters can help you experiment with the business logic in the core easily, in memory, without any specific setup or access to test infrastructure. This means that you could run a huge number of automated tests for the business logic core quickly and reliably. A much smaller number of integration tests focused around adapters can prove the necessary subset of interactions with real infrastructure. Designing with ports and adapters also allows the easy swapping of one infrastructure for another if they serve the same purpose, for example moving external storage from S3 to DynamoDB.


Message Queue Events Adapter
Events Port
Business Logic (Core)
Storage Port
Database Storage Adapter
In-memory Test Events Adapter
In-memory Test Storage Adapter
Web UI Events Adapter
Remote object storage adapter

Ports and Adapters design pattern separates business logic from infrastructure and interfaces. Individual adapters implement port interfaces to interact with the core, and can be easily replaced with other adapter implementations.

Splitting the code

To apply this pattern to Lambda functions, you will usually split the code into three parts:

1. The request processor, representing the core business logic.
2. Concrete adapters for interaction ports, usually one for each type of infrastructure.
3. The Lambda entry point, responsible for wiring everything together.

The first component, the request processor, is responsible for encapsulating the business logic. It does not talk directly to any AWS resources such as S3. It does not care about Lambda interfaces (such as environment variables or context objects). It does not worry about returning values or reporting errors in a way that Lambda can understand. Ideally, it does not load any third-party libraries. You can configure the request processor by passing specific adapters into the constructor method. Apart from the constructor, the request processor usually has only one other method, which starts the request execution. This method should use the names, relationships, and concepts described in the language of the business domain, including parameters, return values, exceptions, and internal processing. The interactions between this method and abstract ports effectively define the collaborator interfaces. This is usually covered with extensive unit tests, using test doubles or mocks as adapters for ports.

Each infrastructure port gets implemented by a concrete adapter that can load the AWS SDK or any other third-party libraries. Individual adapters are typically separate classes. These classes must not assume they are running in Lambda, so they can't access environment variables or the Lambda context. For any such dependencies, these classes provide constructor arguments. You usually do not write unit tests for those, but cover them with integration tests that talk to a real infrastructure component. For example, if a test creates an S3 bucket, try out the adapter methods, and then remove the bucket at the end.

The Lambda entry-point code is responsible for setting up the application components and translating between the business logic core and the Lambda environment. This is where you consume environment variables and the Lambda context, translate between the Lambda event and the parameters of the request processor, and format results or errors in a way that the Lambda infrastructure expects. If this becomes complicated, you may want to extract bits into utility functions. Utility functions usually get covered with unit tests. The main Lambda function is usually trivially simple, so it does not get covered by unit tests. If it's not possible to make it trivially simple, it will get covered by smoke tests that can run after deployment.

Get ready to look at the request processor component in detail in the next lesson.