Author Picks

FREE

Exploring Microservices

Chapters selected by Christian Horsdal Gammelgaard

manning

Exploring Microservices

Selected by Christian Horsdal Gammelgaard

Manning Author Picks

To pre-order or learn more about these books go to www.manning.com

For online information and ordering of these and other Manning books, please visit www.manning.com. The publisher offers discounts on these books when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Leslie Haimes

ISBN 9781617295072
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 - EBM - 21 20 19 18 17 16

contents

introduction

Over the past few years, microservices architectures have gained tremendous popularity. Considering the promise a well-implemented microservices architecture promises, this is not a surprising development.
Microservices promise to enable:

- Continuous delivery and agility
- An efficient and enjoyable developer workflow
- Highly maintainable services
- Robust systems
- Highly scalable systems

These are all benefits that many organizations would like their server-side systems to have, but these benefits are not always realized. In order to successfully achieve results, microservices have to be used just right. The chapters in this in short ebook give you a taste of what it takes to succeed with microservices and reap the benefits I've listed above. In addition, these chapters give you a peek into some of the technologies you can use to implement microservices on the .NET platform.

Before we dive in, let me set the stage by answering what - in a nutshell - a microservice is: A microservice is an individually deployable, autonomous service with one narrowly focused capability.

Microservice Collaboration

This chapter gives you a thorough introduction to a very important aspect of a microservice system: How the microservices collaborate. In order to gain the benefits of microservices, every part of the system has to collaborate in just the right way—a way that lowers coupling between microservices and increasingly enables agility.

Chapter 4 from *Microservices in .NET Core* by Christian Horsdal Gammelgaard.

Microservice collaboration

This chapter covers

- Understanding how microservices collaborate through commands, queries, and events
- Comparing event-based collaboration with collaboration based on commands and queries
- Implementing an event feed
- Implementing command-, query-, and event-based collaboration

Each microservice implements a single capability; but to deliver end user functionality, microservices need to collaborate. Microservices can use three main communication styles for collaboration: *commands*, *queries*, and *events*. Each style has its strengths and weaknesses, and understanding the trade-offs between them allows you to pick the appropriate one for each microservice collaboration. When you get the collaboration style right, you can implement loosely coupled microservices with clear boundaries. In this chapter, I'll show you how to implement all three collaboration styles in code.

4.1 Types of collaboration: commands, queries, and events

Microservices are fine grained and narrowly scoped. To deliver functionality to an end user, microservices need to collaborate.

As an example, consider the Loyalty Program microservice from the point-of-sale system in chapter 3. The Loyalty Program microservice is responsible for the Loyalty Program business capability. The program is simple: customers can register as users with the loyalty program; once registered, they receive notifications about new special offers and earn loyalty points when they purchase something. Still, the Loyalty Program business capability depends on other business capabilities, and other business capabilities depend on it. As illustrated in figure 4.1, the Loyalty Program microservice needs to collaborate with a number of other microservices.

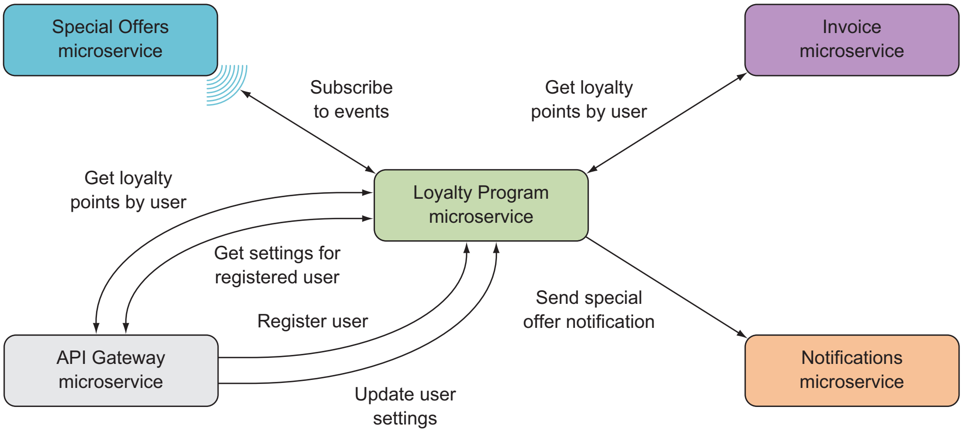

Special Offers microservice
Invoice microservice
Subscribe to events
Get loyalty points by user
Get loyalty points by user
Loyalty Program microservice
Get settings for registered user
Register user
Send special offer notification
API Gateway microservice
Notifications microservice
Update user settings

Figure 4.1 The Loyalty Program microservice collaborates with several other microservices. In some cases, the Loyalty Program microservice receives requests from other microservices; at other times, it sends requests to other microservices.

As stated in the list of microservice characteristics in chapter 1, a microservice is responsible for a single capability; and as discussed in chapter 3, that single capability is typically a business capability. End user functionalities—or use cases—often involve several business capabilities, so the microservices implementing these capabilities must collaborate to deliver functionality to the end user.

When two microservices collaborate, there are three main styles:

- *Commands*—Commands are used when one microservice needs another microservice to perform an action. For example, the Loyalty Program microservice sends a command to the Notifications microservice when it needs a notification to be sent to a registered user.
- *Queries*—Queries are used when one microservice needs information from another microservice. Because customers with many loyalty points receive a

discount, the Invoice microservice queries the Loyalty Program microservice for the number of loyalty points a user has.

- *Events*—Events are used when a microservice needs to react to something that happened in another microservice. The Loyalty Program microservice subscribes to events from the Special Offers microservice so that when a new special offer is made available, it can have notifications sent to registered users.

The collaboration between two microservices can use one, two, or all three of these collaboration styles. Each time two microservices need to collaborate, you must decide which style to use. Figure 4.2 shows the collaborations of Loyalty Program again, but this time identifying the collaboration style I chose for each one.

Collaboration based on commands and queries should use relatively coarse-grained commands and queries. The calls made between microservices are remote calls, meaning they cross at least a process boundary and usually also a network. This means calls between microservices are relatively slow. Even though the microservices are fine grained, you must not fall into the trap of thinking of calls from one microservice to another as being like function calls in a microservice.

Furthermore, you should prefer collaboration based on events over collaboration based on commands or queries. Event-based collaboration is more loosely coupled than the other two forms of collaboration because events are handled asynchronously. That means two microservices collaborating through events aren't temporally coupled: the handling of an event doesn't have to happen immediately after the event is raised. Rather, handling can happen when the subscriber is ready to do so. In contrast, commands and queries are synchronous and therefore need to be handled immediately after they're sent.

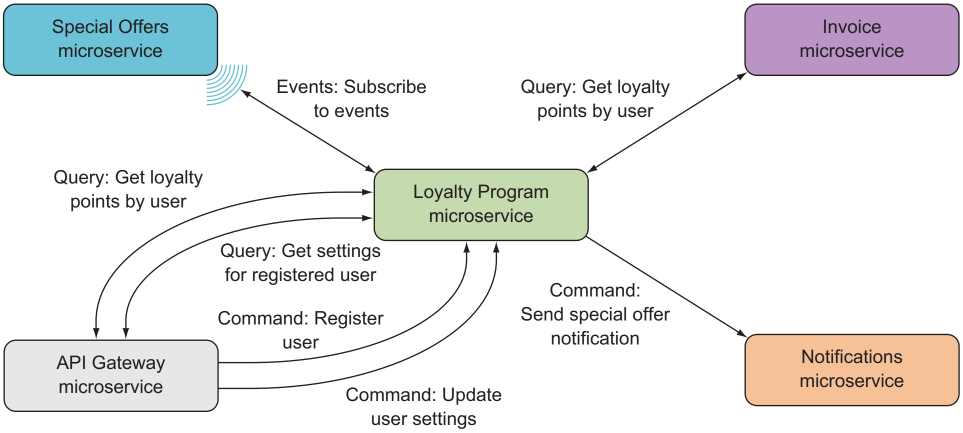

Special Offers microservice
Invoice microservice
Events: Subscribe to events
Query: Get loyalty points by user
Query: Get loyalty points by user
Loyalty Program microservice
Query: Get settings for registered user
Command: Send special offer notification
Command: Register user
API Gateway microservice
Notifications microservice
Command: Update user settings

Figure 4.2 The Loyalty Program microservice uses all three collaboration styles: commands, queries, and events.

4.1.1 Commands and queries: synchronous collaboration

Commands and queries are both synchronous forms of collaboration. Both are implemented as HTTP requests from one microservice to another. Queries are implemented with HTTP `GET` requests, whereas commands are implemented with HTTP `POST` or `PUT` requests.

The Loyalty Program microservice can answer queries about registered users and can handle commands to create or update registered users. Figure 4.3 shows the command- and query-based collaborations that Loyalty Program takes part in.

Figure 4.3 includes two different queries: "Get loyalty points for registered user" and "Get settings for registered user." You'll handle both of these with the same endpoint that returns a representation of the registered user. The representation includes both the number of loyalty points and the settings. You do this for two reasons: it's simpler than having two endpoints, and it's also cleaner because the Loyalty Program microservice gets to expose just one representation of the registered user instead of having to come up with specialized formats for specialized queries.

Two commands are sent to Loyalty Program in figure 4.3: one to register a new user, and one to update an existing registered user. You'll implement the first with an HTTP `POST` and the second with an HTTP `PUT`. This is standard usage of `POST` and `PUT` HTTP methods. `POST` is often used to create a new resource, and `PUT` is defined in the HTTP specification to update a resource.

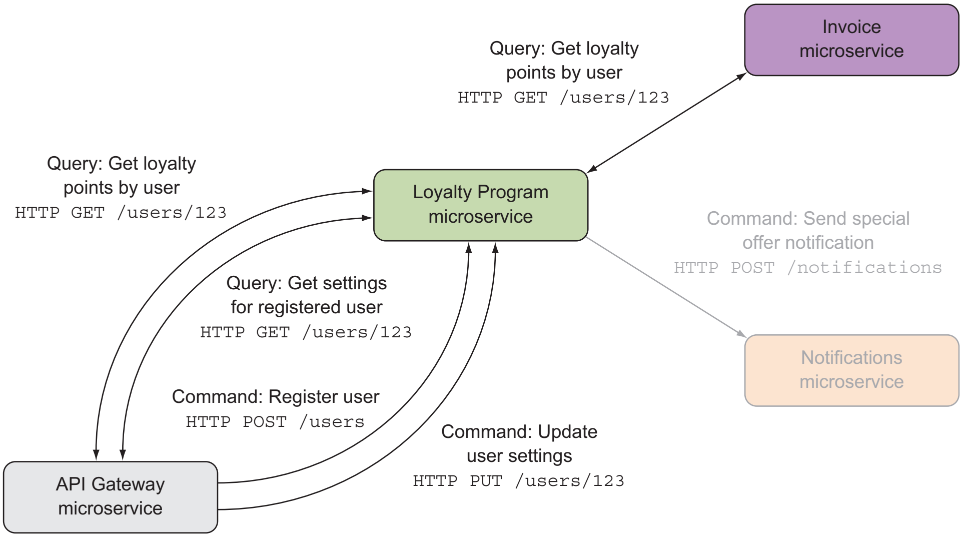

Invoice microservice
Query: Get loyalty points by user
HTTP GET /users/123
Query: Get loyalty points by user
HTTP GET /users/123
Loyalty Program microservice
Command: Send special offer notification
HTTP POST /notifications
Query: Get settings for registered user
HTTP GET /users/123
Notifications microservice
Command: Register user
HTTP POST /users
Command: Update user settings
HTTP PUT /users/123
API Gateway microservice

Figure 4.3 The Loyalty Program microservice collaborates with three other microservices using commands and queries. The queries are implemented as HTTP `GET` requests, and the commands are implemented as HTTP `POST` or `PUT` requests. The command collaboration with the Notifications microservice is grayed out because I'm not going to show its implementation—it's done exactly the same way as the other collaborations.

All in all, the Loyalty Program microservice needs to expose three endpoints:

- An HTTP `GET` endpoint at URLs of the form /users/{userId} that responds with a representation of the user. This endpoint implements both queries in figure 4.3.
- An HTTP `POST` endpoint at /users/ that expects a representation of a user in the body of the request and then registers that user in the loyalty program.
- An HTTP `PUT` endpoint at URLs of the form /users/{userId} that expects a representation of a user in the body of the request and then updates an already-registered user.

The Loyalty Program microservice is made up of the same set of standard components you've seen before, as shown in figure 4.4. The endpoints are implemented in the HTTP API component.

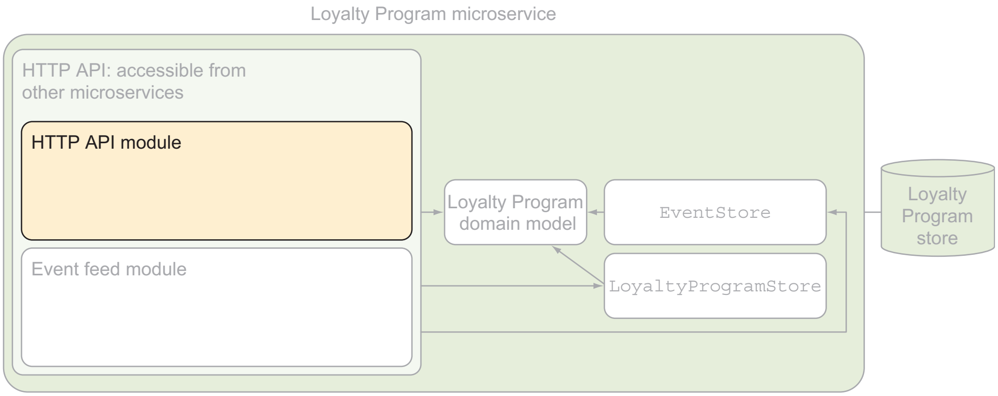

Loyalty Program microservice
HTTP API: accessible from other microservices
HTTP API module
Event feed module
Loyalty Program domain model
EventStore
LoyaltyProgramStore
Loyalty Program store

Figure 4.4 The endpoints exposed by the Loyalty Program microservice are implemented in the HTTP API component.

The other sides of these collaborations are microservices that most likely follow the same standard structure, with the addition of a `LoyaltyProgramClient` component. For instance, the Invoice microservice might be structured as shown in figure 4.5.

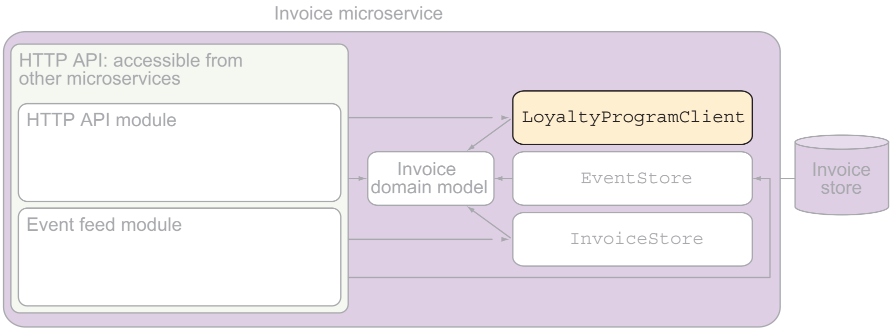


Figure 4.5 The Invoice microservice has a `LoyaltyProgramClient` component responsible for calling the Loyalty Program microservice.

The representation of a registered user that Loyalty Program will expect to receive in the commands and with which it will respond to queries is a serialization of the following `LoyaltyProgramUser` class.

Listing 4.1 The Loyalty Program microservice's user representation

```
public class LoyaltyProgramUser
{
  public int Id { get; set; }
  public string Name { get; set;  }
  public int LoyaltyPoints { get; set;  }
  public LoyaltyProgramSettings Settings { get; set; }
}

public class LoyaltyProgramSettings
{
  public string[] Interests { get; set; }
}
```

The definitions of the endpoints and the two classes in this code effectively form the contract that the Loyalty Program microservice publishes. The `LoyaltyProgramClient` component in the Invoice microservice adheres to this contract when it makes calls to the Loyalty Program microservice, as illustrated in figure 4.6.

Commands and queries are powerful forms of collaboration, but they both suffer from being synchronous by nature. As mentioned earlier, that creates coupling between the microservices that expose the endpoints and the microservices that call the endpoints. Next, we'll turn our attention to asynchronous collaboration through events.

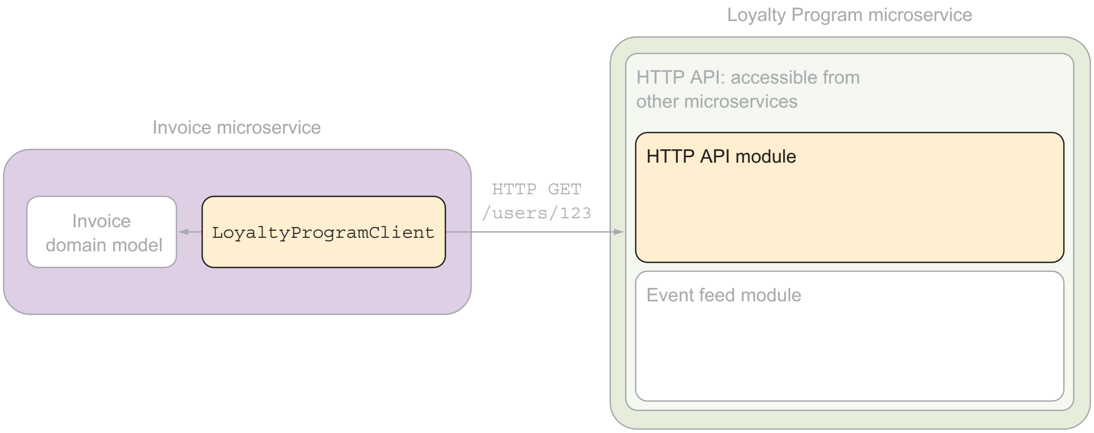


Figure 4.6 The `LoyaltyProgramClient` component in the Invoice microservice is responsible for making calls to the Loyalty Program microservice. It translates between the contract published by Loyalty Program and the domain model of Invoice.

4.1.2 Events: asynchronous collaboration

Collaboration based on events is asynchronous. That is, the microservice that publishes the events doesn't call the microservices that subscribe to the events. Rather, the subscribers poll the microservice that publishes events for new events when they're ready to process them. That polling is what I'll call *subscribing* to an event feed. Although the polling is made out of synchronous requests, the collaboration is asynchronous because publishing events is independent of any subscriber polling for events.

In figure 4.7, you can see the Loyalty Program microservice subscribing to events from the Special Offers microservice. Special Offers can publish events whenever something happens in its domain, such as every time a new special offer becomes active. Publishing an event, in this context, means storing the event in Special Offers. Loyalty Program won't see the event until it makes a call to the event feed on Special Offers. When that happens is entirely up to Loyalty Program. It can happen right after the event is published or at any later point in time.

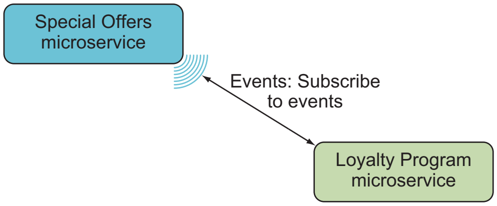

Special Offers microservice
Events: Subscribe to events
Loyalty Program microservice

Figure 4.7 The Loyalty Program microservice processes events from the Special Offers microservice when it's convenient for Loyalty Program.

As with the other types of collaboration, there are two sides to event-based collaboration. One side is the microservice that publishes events through an event feed, and the other is the microservices that subscribe to those events.

EXPOSING AN EVENT FEED

A microservice can publish events to other microservices via an *event feed*, which is just an HTTP endpoint—at /events, for instance—to which that other microservice can make requests and from which it can get event data. Figure 4.8 shows the components in the Special Offers microservice. Once again, the microservice has the same standard set of components that you've seen several times already. In figure 4.8, the components involved in implementing the event feed are highlighted.

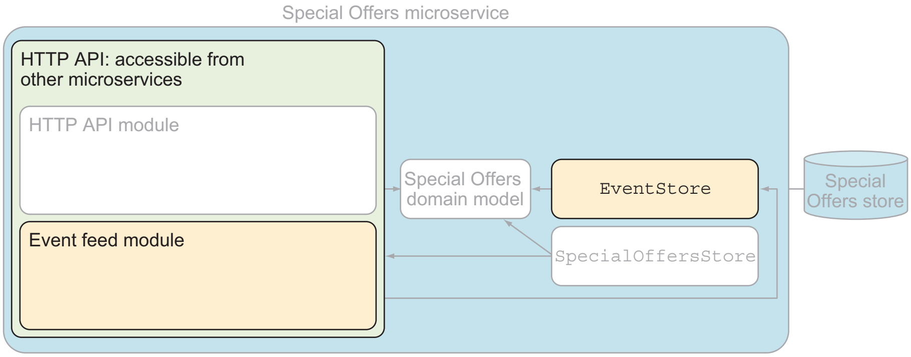

Special Offers microservice
HTTP API: accessible from other microservices
HTTP API module
Event feed module
Special Offers domain model
EventStore
SpecialOffersStore
Special Offers store

Figure 4.8 The event feed in the Special Offers microservice is exposed to other microservices over HTTP and is based on the event store.

The events published by the Special Offers microservice are stored in its database. The `EventStore` component has the code that reads events from and writes them to the database. The domain model code can use `EventStore` to store the events it needs to publish. The `Event Feed` component is the implementation of the HTTP endpoint that exposes the event to other microservices: that is, the /events endpoint.

The `Event Feed` component uses `EventStore` to read events from the database and then returns the events in the body of an HTTP response. Subscribers can use query parameters to control which and how many events are returned.

Subscribing to events

Subscribing to an event feed essentially means you poll the events endpoint of the microservice that you subscribe to. At intervals, you send an HTTP `GET` request to the /events endpoint to check whether there are any events you haven't processed yet.

Figure 4.9 is an overview of the Loyalty Program microservice, which shows that it consists of two processes. We've already talked about the web process, but the event-subscriber process is new.

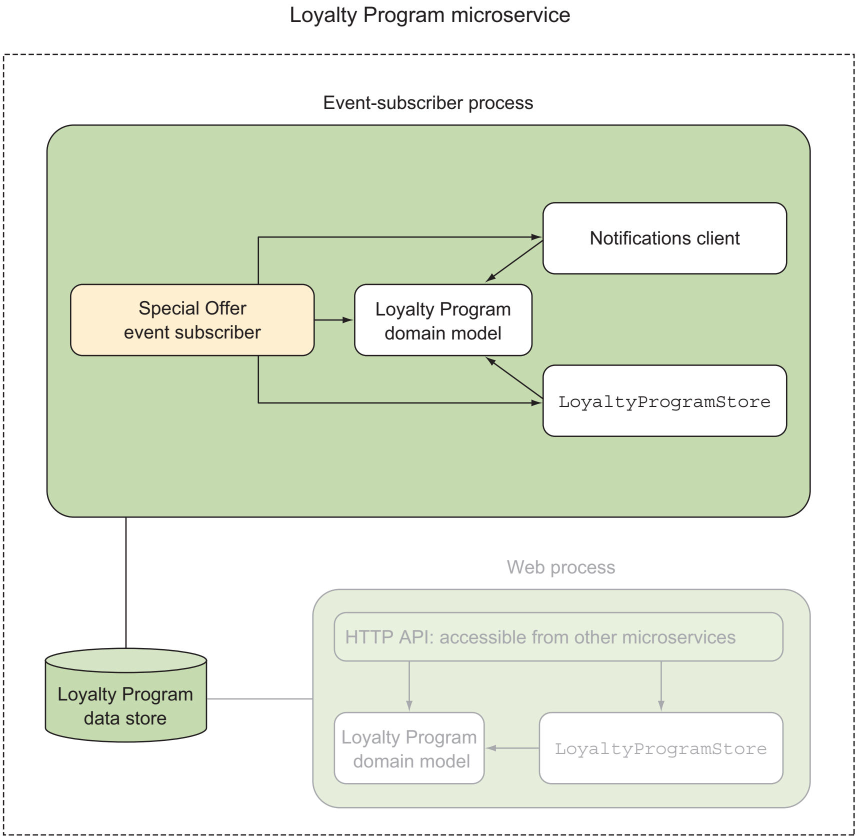

Loyalty Program microservice
Event-subscriber process
Notifications client
Special Offer event subscriber
Loyalty Program domain model
LoyaltyProgramStore
Web process
HTTP API: accessible from other microservices
Loyalty Program data store
Loyalty Program domain model
LoyaltyProgramStore

Figure 4.9 The event subscription in the Loyalty Program microservice is handled in a event-subscriber process.

The event-subscriber process is a background process that periodically makes requests to the event feed on the Special Offers microservice to get new events. When it gets back new events, it processes them by sending commands to the Notifications microservice to notify registered users about new special offers. The `SpecialOffersSubscriber` component is where the polling of the event feed is implemented, and the `NotificationsClient` component is responsible for sending the command to Notifications.

This is the way you implement event subscriptions: microservices that need to subscribe to events have a subscriber process with a component that polls the event feed. When new events are returned from the event feed, the subscriber process handles the events based on business rules.

Events over queues

An alternative to publishing events over an event feed is to use a queue technology, like RabbitMQ or Service Bus for Windows Server. In this approach, microservices that publish events push them to a queue, and subscribers read them from the queue. Events must be routed from the publisher to the subscribers, and how that's done depends on the choice of queue technology. As with the event-feed approach, the microservice subscribing to events has an event-subscriber process that reads events from the queue and processes them.

This is a perfectly viable approach to implementing event-based collaboration between microservices. But this book uses HTTP-based event feeds for event-based collaboration because it's a simple yet robust and scalable solution.

4.1.3 Data formats

So far, we've focused on exchanging data in JSON format. I've mentioned in passing that XML is supported equally by all the endpoints you've implemented with Nancy. (Nancy comes with JSON and XML serialization and deserialization out of the box.) These two options cover most situations, but there are reasons you might want something else:

- If you need to exchange a lot of data, a more compact format may be needed. Text-based formats such as JSON and XML are a lot more verbose than binary formats like protocol buffers.
- If you need a more structured format than JSON that's still human readable, you may want to use YAML.
- If your company uses proprietary data formatting, you may need to support that format.

In all these cases, you need endpoints capable of receiving data in another format than XML or JSON, and they also need to be able to respond in that other format. As an example, a request to register a user with the Loyalty Program microservice using YAML in the request body looks like this:

```
POST /users HTTP/1.1
Host: localhost:5000
Accept: application/yaml
Content-Type: application/vyaml

Name: Christian
Settings:
  Interests:
  - whisky
  - cycling
  - software design
```

Specifies that the request body is in YAML format

Asks for the response in YAML format

Provides a YAML-formatted request body

The response to this request also uses YAML:

```
HTTP/1.1 201 Created
Content-Type: application/yaml
Location: http://localhost:5000/users/1

Id: 1
Name: Christian
Settings:
  Interests:
  - whisky
  - cycling
```

Specifies that the response body is in YAML format

Provides a YAML-formatted response body

Both the preceding request and response have YAML-formatted bodies, and both specify that the body is YAML in the `Content-Type` header. The request uses the `Accept` header to ask for the response in YAML. This example shows how microservices can communicate using different data formats and how they can use HTTP headers to tell which formats are used.

4.2 Implementing collaboration

This section will show you how to code the collaborations you saw earlier in figure 4.2. I'll use the Loyalty Program microservice as a starting point, but I'll also go into some of its collaborators—the API Gateway microservice, the Invoice microservice, and the Special Offers microservice—in order to show both ends of the collaborations.

Three steps are involved in implementing the collaboration:

1. Set up a project for Loyalty Program. Just as you've done before, you'll create an empty ASP.NET 5 application and add Nancy to it. The only difference this time is that you'll add a little Nancy configuration code.
2. Implement the command- and query-based collaborations shown in figure 4.2. You'll implement all the commands and queries that Loyalty Program can handle, as well as the code in collaborating microservices that use them.
3. Implement the event-based collaboration shown in figure 4.2. You'll start with the event feed in Special Offers and then move on to implement the subscription in Loyalty Program. In the process, you'll add an extra project—and an extra process—to Loyalty Program. After these steps, you'll have implemented all the collaborations of Loyalty Program.

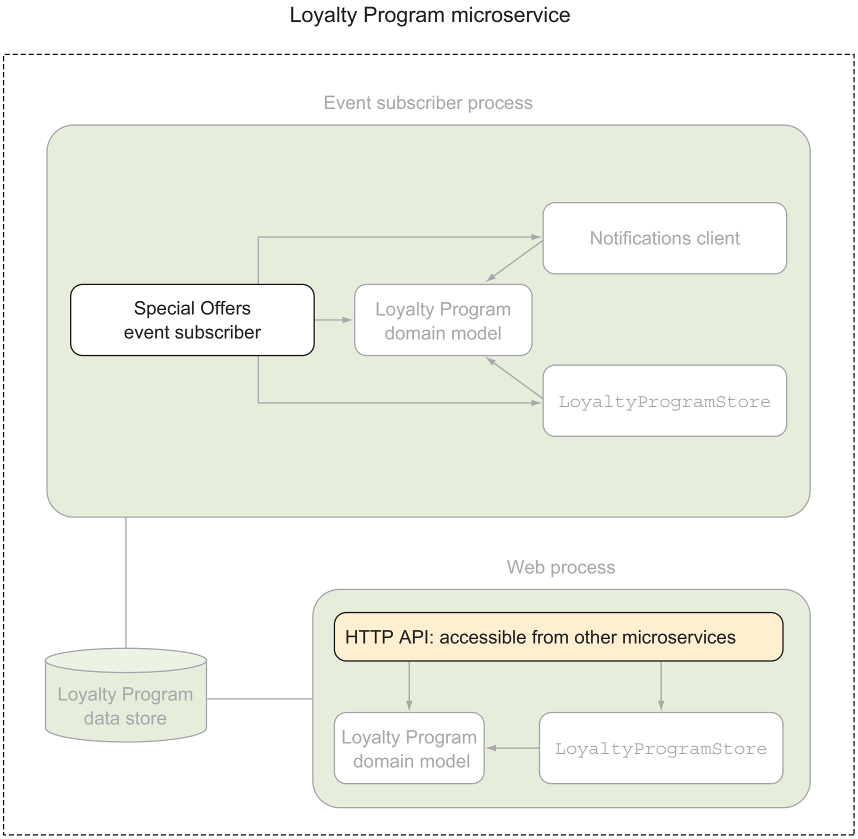

Loyalty Program microservice
Event subscriber process
Notifications client
Special Offers event subscriber
Loyalty Program domain model
LoyaltyProgramStore
Web process
HTTP API: accessible from other microservices
Loyalty Program data store
Loyalty Program domain model
LoyaltyProgramStore

Figure 4.10 The Loyalty Program microservice has a web process that follows the structure you've seen before and an event-subscriber process that handles the subscription to events from the Special Offers microservice. I'll only show the code for the highlighted components in this chapter.

The Loyalty Program microservice consists of a web process that has the same structure you've seen before. This is illustrated at the bottom of figure 4.10. Later, when you implement the event-based collaboration, you'll add another process that I call the *event-subscriber process*. This process is shown at the top of figure 4.10.

In the interest of focusing on the collaboration, I won't show all the code in the Loyalty Program microservice. Rather, I'll include the code for the HTTP API in the web process, and the special offer event subscriber in the event-subscriber process.

4.2.1 Setting up a project for Loyalty Program

The first thing to do in implementing the Loyalty Program microservice is to create an empty ASP.NET 5 application and add Nancy to it as a NuGet package. You've already done this a couple of times—in chapters 1 and 2—so I won't go over the details again here.

This time around, there's one more piece of setup to do: you'll override how Nancy handles responses with a 404 Not Found status code. By default, Nancy puts the HTML for an error page in the body of a 404 Not Found response; but because the clients of the Loyalty Program microservice aren't web browsers but other microservices, you don't need an error page. I'd rather have a response with a 404 Not Found status code and an empty body. Toward this end, add a file to the project called Bootstrapper.cs. In this file, put the following class that inherits from `DefaultNancyBootstrapper`.

Listing 4.2 Nancy bootstrapper

```
namespace LoyaltyProgram
{
  using System;
  using Nancy;
  using Nancy.Bootstrapper;

  public class Bootstrapper : DefaultNancyBootstrapper
  {
    protected override
      Func<ITypeCatalog, NancyInternalConfiguration> InternalConfiguration =>
       NancyInternalConfiguration
         .WithOverrides(builder => builder.StatusCodeHandlers.Clear());
  }
}
```

Remove all default status-code handlers so they don't alter the responses.

Nancy will automatically discover this class at startup, call the `InternalConfiguration` getter, and use the configuration returned from that. You reuse the default configuration except that you clear all `StatusCodeHandlers`, which means you're removing everything that might alter a response because of its status code.

The Nancy bootstrapper

Nancy uses the bootstrapper during application startup to configure both the framework itself and the application. Nancy allows applications to reconfigure the entire framework, and you *can* swap any part of Nancy for your own implementation in your bootstrapper. In this regard, Nancy is open and flexible. In many cases, you don't need to configure the framework—Nancy has sensible defaults—and when you do, you rarely need to swap out entire pieces of Nancy.

To create a bootstrapper, all you have to do is create a class that implements the `INancyBootstrapper` interface, and Nancy will discover it and use it. You won't usually implement that interface directly, because although the interface itself is simple, a fully functional implementation of it isn't. Instead of implementing `INancyBootstrapper` directly, you can take advantage of the default bootstrapper that Nancy comes with out of the box (`DefaultNancyBootstrapper`) and extend it. That class has a number of virtual methods that you can override to hook into different parts of Nancy. There are, for instance, methods to configure the dependency injection container that Nancy uses, methods to set up specialized serialization and deserialization, methods to add error handlers, and more.

(continued)
You'll use the Nancy bootstrapper several times throughout the book, but for the most part you'll rely happily on Nancy's defaults. If an application doesn't have a Nancy bootstrapper, Nancy uses the default one: `DefaultNancyBootstrapper`.

4.2.2 Implementing commands and queries

You now have a web project ready to host the implementations of the endpoints exposed by the Loyalty Program microservice. As listed earlier, these are the endpoints:

- An HTTP `GET` endpoint at URLs of the form /users/{userId} that responds with a representation of the user. This endpoint implements both queries in figure 4.3.
- An HTTP `POST` endpoint at /users/ that expects a representation of a user in the body of the request and then registers that user in the loyalty program
- An HTTP `PUT` endpoint at URLs of the form /users/{userId} that expects a representation of a user in the body of the request and then updates an already-registered user.

You'll implement the command endpoints first and then the query endpoint.

4.2.3 Implementing commands with HTTP POST or PUT

The code needed in the Loyalty Program microservice to implement the handling of the two commands—the HTTP `POST` to register a new user and the HTTP `PUT` to update one—is similar to the code you saw in chapter 2. You'll start by implementing a handler for the command to register a user. A request to Loyalty Program to register a new user is shown in the following listing.

Listing 4.3 Request to register a user named Christian

```
POST /users HTTP/1.1
Host: localhost:5000
Content-Type: application/json
Accept: application/json

{
  "id":0,
  "name":"Christian",
  "loyaltyPoints":0,
  "settings":{ "interests" : ["whisky", "cycling"] }
}
```

JSON representation of the user being registered

To handle the command for registering a new user, you need to add a Nancy module to Loyalty Program by adding a file called UserModule.cs and putting the following code in it.

Listing 4.4 `POST` endpoint for registering users

```
using System.Collections.Generic;
using Nancy;
using Nancy.ModelBinding;


public class UsersModule : NancyModule
{
  public UsersModule() : base("/users")
  {
    Post("/", _ =>
    {
      var newUser = this.Bind<LoyaltyProgramUser>();
      this.AddRegisteredUser(newUser);
      return this.CreatedResponse(newUser);
    });
  }

  private dynamic CreatedResponse(LoyaltyProgramUser newUser)
  {
    return
      this.Negotiate
        .WithStatusCode(HttpStatusCode.Created)
        .WithHeader(
          "Location",
          this.Request.Url.SiteBase + "/users/" + newUser.Id)
        .WithModel(newUser);
  }

  private void AddRegisteredUser(LoyaltyProgramUser newUser)
  {
    // store the newUser to a data store
  }
}
```

The request must include a LoyaltyProgramUser in the body. If it doesn't, the request is malformed.

Negotiate is an entry point to Nancy's fluent API for creating responses.

Uses the 201 Created status code for the response

Returns the user in the response for convenience

Adds a location header to the response because this is expected by HTTP for 201 Created responses

The response to the preceding request looks like this:

```
HTTP/1.1 201 Created
Content-Type: application/json; charset=utf-8
Location: http://localhost:5000/users/4

{
  "id": 4,
  "name": "Christian",
  "loyaltyPoints": 0,
  "settings": { "interests": ["whisky", "cycling"]
  }
}
```

The status code is 201 Created.

Nancy's content negotiation sets the Content-Type.

The Location header points to the newly created resource.

The main new thing to notice in listing 4.4 is the use of `Negotiate` to create the response to the command. `Negotiate` is a property on the `NancyModule` class that you use as a base class for `UserModule`. Here, it mainly works as an entry point to Nancy's nice, fluent API for creating responses. In the handler, you use that API to set the status code, add a `Location` header, and add the user object to the response. The API will also allow you to do more things to the response, such as setting other headers and specifying a view that will be used when responding to requests that ask for HTML in the `Accept` header.

`Negotiate` also triggers Nancy's content-negotiation functionality. Content negotiation is how HTTP specifies that the format of data in responses should be decided. It essentially means reading the `Accept` header in the request and serializing to a format indicated there. In listing 4.3, the accept header is `Accept: application/json`, meaning the response should serialize data to JSON.

With the handler for the register-user command in place, let's turn our attention to implementing a handler for the update-user command. That handler is added to `UserModule`.

Listing 4.5 `PUT` endpoint for registering users

```
public class UsersModule : NancyModule
{
  public UsersModule() : base("/users")
  {
    Post("/", _ => ...);

    Put("/{userId:int}", parameters =>
    {
      int userId = parameters.userId;
      var updatedUser = this.Bind<LoyaltyProgramUser>();
      // store the updatedUser to a data store
      return updatedUser;
    });
  }
  ...
}
```

Nancy turns the user object into a complete response.

There's nothing in this code you haven't seen before.

The handlers for the commands are only one side of the collaboration. The other side is the code that sends the commands. Figure 4.2 shows that the API Gateway microservice sends commands to the Loyalty Program microservice. You won't build a complete API Gateway microservice here, but in the code download for this chapter, you'll find a console application that acts as API Gateway would with regard to collaborating with Loyalty Program. Here, we'll focus only on the code that sends the commands.

In the API Gateway microservice, you'll create a class called `LoyaltyProgramClient` that's responsible for dealing with communication with the Loyalty Program microservice. That class encapsulates everything involved in building HTTP requests, serializing data for requests, understanding HTTP responses, and deserializing response data.

The code for sending the registered-user command takes a LoyaltyProgramUser as input and creates an HTTP POST with the LoyaltyProgramUser object in the body, and it sends that to the Loyalty Program microservice. After it checks the response status code and confirms that it's 201 Created, it deserializes the body of the response to a LoyaltyProgramUser and returns it. If the status code is anything else, the method returns null. The following listing shows the implementation.

Listing 4.6 The API Gateway microservice registering new users

```
using System;
using System.Text;
using System.Threading.Tasks;
using System.Net;
using System.Net.Http;
using Newtonsoft.Json;

public class LoyaltyProgramClient
{
  public async Task<LoyaltyProgramUser>
  ➥ RegisterUser(LoyaltyProgramUser newUser)
  {
    using(var httpClient = new HttpClient())
    {
      httpClient.BaseAddress = new Uri($"http://{this.hostName}");
      var response = await
        httpClient.PostAsync(
          "/users/",
          new StringContent(
            JsonConvert.SerializeObject(newUser),
            Encoding.UTF8,
            "application/json"));
      ThrowOnTransientFailure(response);
      return JsonConvert.DeserializeObject<LoyaltyProgramUser>(
          await response.Content.ReadAsStringAsync());
    }
  }
}
```

Sends the command to Loyalty Program

Serializes newUser as JSON

Sets the Content-Type header

Deserializes the response if the command was handled successfully

Similarly, LoyaltyProgramClient has a method for sending the update-user command. This method also encapsulates the HTTP communication involved in sending the command.

Listing 4.7 The API Gateway microservice updating users

```
public async Task<LoyaltyProgramUser> UpdateUser(LoyaltyProgramUser user)
{
  using(var httpClient = new HttpClient())
  {
    httpClient.BaseAddress = new Uri($"http://{this.hostName}");
    var response = await
```

```
        httpClient.PutAsync(
          $"/users/{user.Id}",
          new StringContent(
            JsonConvert.SerializeObject(user),
            Encoding.UTF8,
            "application/json"));
      ThrowOnTransientFailure(response);
      return JsonConvert.DeserializeObject<LoyaltyProgramUser>(
        await response.Content.ReadAsStringAsync());
    }
  }
```

Sends the update-user command as a PUT request

This code is similar to the code for the register-user command, except this HTTP request uses the PUT method. With the command handlers implemented in the Loyalty Program microservice and a LoyaltyProgramClient implemented in the API Gateway microservice, the command-based collaboration is implemented. API Gateway can register and update users, but it can't yet query users.

4.2.4 Implementing queries with HTTP GET

The Loyalty Program microservice can handle the commands it needs to handle, but it can't answers queries about registered users. Remember that Loyalty Program only needs one endpoint to handle queries. As mentioned previously, the endpoint handling queries is an HTTP GET endpoint at URLs of the form /users/{userId}, and it responds with a representation of the user. This endpoint implements both queries in figure 4.4.

Listing 4.8 GET endpoint to query a user by ID

```
public class UsersModule : NancyModule
{
  private static IDictionary<int, LoyaltyProgramUser> registeredUsers =
    new Dictionary<int, LoyaltyProgramUser>();

  public UsersModule() : base("/users")
  {
    Post("/", _ => ...);

    Put("/{userId:int}", parameters => ...);

    Get("/{userId:int}", parameters =>
    {
      int userId = parameters.userId;
      if (registerUsers.ContainsKey(userId))
        return registerUsers[userId];
      else
        return HttpStatusCode.NotFound;
    });
  }
  ...
}
```

There's nothing about this code that you haven't already seen several times. Likewise, the code needed in the API Gateway microservice to query this endpoint shouldn't come as a surprise:

```
public class LoyaltyProgramClient
{
  ...

  public async Task<LoyaltyProgramUser> QueryUser(int userId)
  {
    var userResource = $"/users/{userId}";
    using(var httpClient = new HttpClient())
    {
      httpClient.BaseAddress = new Uri($"http://{this.hostName}");
      var response = await httpClient.GetAsync(userResource);
      ThrowOnTransientFailure(response);
      return JsonConvert.DeserializeObject<LoyaltyProgramUser>(
        await response.Content.ReadAsStringAsync());
    }
  }
}
```

This is all that's needed for the query-based collaboration. You've now implemented the command- and query-based collaborations of the Loyalty Program microservice.

4.2.5 Data formats

Suppose you want the endpoints you just implemented to support YAML. You shouldn't implement support for a third data format in the endpoint handlers—it's not a concern of the application logic, it's a technical concern.

Nancy lets you support deserialization of another format by implementing the `IBodyDeserializer` interface. In typical Nancy style, any implementation of that interface is picked up at application startup and is hooked into Nancy's model binding. Likewise, to support serialization of response bodies in a third format, you can implement `IResponseProcessor`, which also is automatically discovered by Nancy and gets hooked into Nancy's content negotiation.

To implement YAML support in the Loyalty Program microservice, you'll first install the `YamlDotNet NuGet` package in the project. Then, you'll add a file called YamlSerializerDeserializer.cs. You'll use this file to implement both the deserialization and the serialization. The deserialization looks like this.

Listing 4.9 Deserializing from YAML

```
namespace LoyaltyProgram
{
  using System.IO;
  using Nancy.ModelBinding;
  using Nancy.Responses.Negotiation;
  using YamlDotNet.Serialization;
  public class YamlBodyDeserializer : IBodyDeserializer
  {
    public bool CanDeserialize(
      MediaRange mediaRange, BindingContext context)
        => mediaRange.Subtype.ToString().EndsWith("yaml");

    public object Deserialize(
      MediaRange mediaRange, Stream bodyStream, BindingContext context)
    {
      var yamlDeserializer = new Deserializer();
      var reader = new StreamReader(bodyStream);
      return yamlDeserializer.Deserialize(
        reader, context.DestinationType);
    }
  }
}
```

Tells Nancy which content types this deserializer can handle

Tries to deserialize the request body to the type needed by the application code

This code mainly uses the YamlDotNet library to deserialize the data from the body of the request.

The implementation of the serialization support isn't as simple, but it's still only a matter of implementing two methods and a property.

Listing 4.10 Serializing to YAML

```
namespace LoyaltyProgram
{
  using System;
  using System.Collections.Generic;
  using System.IO;
  using Nancy;
  using Nancy.Responses.Negotiation;
  using YamlDotNet.Serialization;
  ...

  public class YamlBodySerializer : IResponseProcessor
  {
    public IEnumerable<Tuple<string, MediaRange>> ExtensionMappings
    {
      get
      {
        yield return new Tuple<string, MediaRange>(
          "yaml", new MediaRange("application/yaml"));
      }
    }

  public ProcessorMatch CanProcess(
    MediaRange requestedMediaRange, dynamic model, NancyContext context)
    =>
      requestedMediaRange.Subtype.ToString().EndsWith("yaml")
      ? new ProcessorMatch
        {
          ModelResult = MatchResult.DontCare,
```

Tells Nancy which file extensions can be handled by this response processor. You don't use this feature.

Tells Nancy that this processor can handle accept header values that end with "yaml"

```
          RequestedContentTypeResult = MatchResult.NonExactMatch
        }
      : ProcessorMatch.None;

  public Response Process(
    MediaRange requestedMediaRange, dynamic model, NancyContext context)
    =>
      new Response
      {
        Contents = stream =>
        {
          var yamlSerializer = new Serializer();
          var streamWriter = new StreamWriter(stream);
          yamlSerializer.Serialize(streamWriter, model);
          streamWriter.Flush();
        },
        ContentType = "application/yaml"
      };
  }
}
```

Creates a new response object to use in the rest of Nancy's pipeline

Sets up a function that writes the response body to a stream

Writes the YAML serialized object to the stream Nancy uses for the response body

The serialization is also handled by the YamlDotNet library. The code in `ExtensionMappings` and `CanProcess` in `YamlBodySerializer` tells Nancy which responses it applies to. The code in `Process` creates a response with a YAML-serialized body. This response may be processed more if the code in the handler customizes the response further. For instance, the response to the register-user command is created like this:

```
return
  this.Negotiate
    .WithStatusCode(HttpStatusCode.Created)
    .WithHeader(
      "Location",
      this.Request.Url.SiteBase + "/users/" + newUser.Id)
    .WithModel(newUser);
```

This code customizes the response through the `.With*` extension methods. After `YamlBodySerializer` has created the response, including the YAML-formatted body, the `WithStatusCode` and `WithHeader` methods further customize the response. As you've seen, all it takes to make your Nancy-based microservices support another data format is an implementation of `IBodyDeserializer` and an implementation of `IResponseProcessor`.

4.2.6 Implementing an event-based collaboration

Now that you know how to implement command- and query-based collaborations between microservices, it's time to turn our attention to the event-based collaboration. Figure 4.11 repeats the collaborations that the Loyalty Program microservice is involved in. Loyalty Program subscribes to events from Special Offers, and it uses the events to decide when to notify registered users about new special offers.

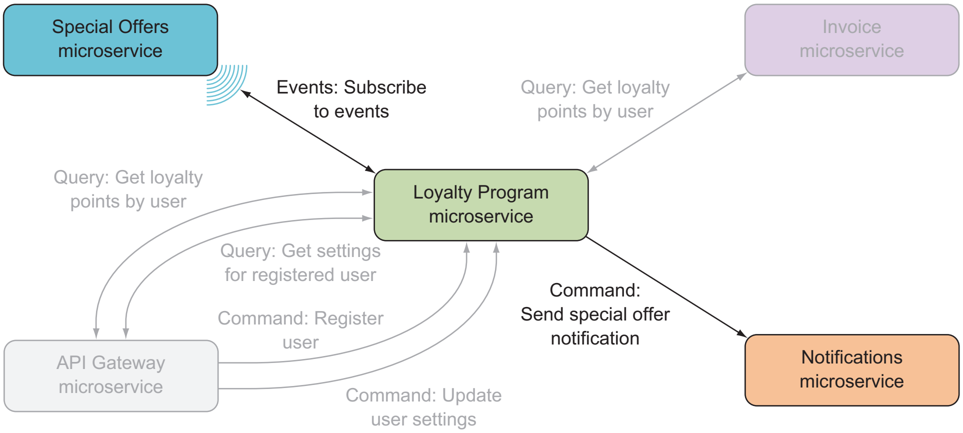

Special Offers microservice
Invoice microservice
Events: Subscribe to events
Query: Get loyalty points by user
Query: Get loyalty points by user
Loyalty Program microservice
Query: Get settings for registered user
Command: Send special offer notification
Command: Register user
API Gateway microservice
Notifications microservice
Command: Update user settings

Figure 4.11 The event-based collaboration in the Loyalty Program microservice is the subscription to the event feed in the Special Offers microservice.

We'll first look at how Special Offers exposes its events in a feed. Then, you'll return to Loyalty Program and add a second process to that service, which will be responsible for subscribing to events and handling events.

IMPLEMENTING AN EVENT FEED

You saw a simple event feed in chapter 2. The Special Offers microservice implements its event feed the same way: it exposes an endpoint—`/events`—that returns a list of sequentially numbered events. The endpoint can take two query parameters—`start` and `end`—that specify a range of events. For example, a request to the event feed can look like this:

```
GET /events?start=10&end=110 HTTP/1.1

Host: specialoffers.mycompany.com
Accept: application/json
```

The response to this request might be the following, except that I've cut off the response after two events:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

[
  {
    "sequenceNumber": 10,
    "occuredAt": "2015-10-02T18:37:00.7070659+00:00",
    "name": "NewSpecialOffer",
    "content": {
      "offerId": 123,
      "offer": {
        "productCatalogueId": 1,
        "productName": "Basic t-shirt",
        "description": "Get an awesome t-shirt at half price!",
      }
    }
  },
  {
    "sequenceNumber": 11,
    "occuredAt": "2015-10-02T20:01:00.3050629+00:00",
    "name": "UpdatedSpecialOffer",
    "content": {
      "offerId": 124,
      "offer": {
        "productCatalogueId": 10,
        "productName": "Hot teacup",
        "description": "Get a Cup<T>. Because you know you want to.",
        "update": "Now with 10% more inference"
      }
    }
  }
}
```

Notice that the events have different names (`NewSpecialOffer` and `UpdatedSpecialOffer`) and the two types of events don't have the same data fields. This is normal: different events carry different information. It's also something you need to be aware of when you implement the subscriber in the Loyalty Program microservice. You can't expect all events to have the exact same shape.

The implementation of the /events endpoint in the Special Offers microservice is a simple Nancy module, just like the one in chapter 2.

Listing 4.11 Endpoint that reads and returns events

```
namespace SpecialOffers.EventFeed
{
  using Nancy;

  public class EventsFeedModule : NancyModule
  {
    public EventsFeedModule(IEventStore eventStore) : base("/events")
    {
      Get("/", _ =>
      {
        long firstEventSequenceNumber, lastEventSequenceNumber;
        if (!long.TryParse(this.Request.Query.start.Value,
          out firstEventSequenceNumber))
          firstEventSequenceNumber = 0;
        if (!long.TryParse(this.Request.Query.end.Value,
          out lastEventSequenceNumber))
          lastEventSequenceNumber = long.MaxValue;

        return
          eventStore.GetEvents(
            firstEventSequenceNumber,
            lastEventSequenceNumber);
      });
    }
  }
}
```

This module only uses Nancy features that we've already discussed. You may notice, however, that it returns the result of `eventStore.GetEvents` directly, which is an `IEnumerable<Event>`; Nancy serializes it as an array. The `Event` is a struct that carries a little metadata and a `Content` field that's meant to hold the event data.

Listing 4.12 `Event` class that represents events

```
public struct Event
{
  public long SequenceNumber { get; }
  public DateTimeOffset OccuredAt { get; }
  public string Name { get; }
  public object Content { get; }

  public Event(
    long sequenceNumber,
    DateTimeOffset occuredAt,
    string name,
    object content)
  {
    this.SequenceNumber = sequenceNumber;
    this.OccuredAt = occuredAt;
    this.Name = name;
    this.Content = content;
  }
}
```

The `Content` property is used for event-specific data and is where the difference between a `NewSpecialOffer` event and an `UpdatedSpecialOffer` event appears. The former has one type of object in `Content`, and the latter has another.

This is all it takes to expose an event feed. This simplicity is the great advantage of using an HTTP-based event feed to publish events. Event-based collaboration can be implemented over a queue system, but that introduces another complex piece of technology that you have to learn to use and administer in production. That complexity is warranted in some situations, but certainly not always.

CREATING AND RUNNING AN EVENT-SUBSCRIBER PROCESS

The first step in implementing an event-subscriber process is to create a console application. You're using ASP.NET Core, which is based on .NET Core, for the web processes in the example microservices, so you'll create a console application that's .NET Core–based and call it `LoyaltyProgramEventConsumer`. You can create a .NET Core–based console application in Visual Studio 2015 by selecting the Console Application (Package) project type in the New Project dialog box. Alternatively, you can go to a PowerShell prompt, run the Yeoman ASP.NET generator,[1] and select the option to generate a Console Application.

Listing 4.13 Generating a console app with the Yeoman ASP.NET generator

```
PS> yo aspnet

     _-----_
    |       |    .--------------------------.
    |--(o)--|    |      Welcome to the      |
   `---------´   |  marvellous ASP.NET Core |
    ( _´U`_ )    |      1.0 generator!      |
    /___A___\    '--------------------------'
     |  ~  |
   __'.___.'__
 ´   `  |° ´ Y `

? What type of application do you want to create?
  Empty Web Application
> Console Application
  Web Application
  Web Application Basic [without Membership and Authorization]
  Web API Application
  Nancy ASP.NET Application
  Class Library
  Unit test project (xUnit.net)
```

Move the cursor here, and press Enter to generate a console app.

Whether you create the `LoyaltyProgramEventConsumer` with Visual Studio or Yeoman, you can run it by going to the project folder—the folder where the project.json file is—in PowerShell and using `dotnet`:

```
PS> dotnet run
```

The application is empty, so nothing interesting happens yet. Running `LoyaltyProgramEventConsumer` like that from PowerShell is something you'll only do for testing. In production, you might run `LoyaltyProgramEventConsumer` as a Windows service. If the production environment is based on Windows Servers that you (or your organization) run, a Windows service may well be the right choice; but if your production environment is in a cloud, in may not be.

> **WARNING** I'm implementing `LoyaltyProgramEventConsumer` as a Windows service, which only works on Windows. If you want to run on Linux, you can create a similar `LoyaltyProgramEventConsumer` as a Linux daemon.

Creating a Windows service is straightforward and is no different with a .NET Core–based console application than it was before .NET Core. The project already has a Program.cs file containing a `Program` class. The `Program` class has a `Main` method, which is

[1] See appendix A for instructions on installing Yeoman and the Yeoman ASP.NET generator.

the entry point to the application. To turn it into a Windows service, the `Program` class just has to inherit from `ServiceBase` and override the `OnStart` and `OnStop` methods, as in the following listing.

Listing 4.14 Making `Program` run as a Windows service

```
using System.ServiceProcess;

public class Program : ServiceBase
{
  private EventSubscriber subscriber;

  public void Main(string[] args)
  {
    // more to come
    Run(this);
  }

  protected override void OnStart(string[] args)
  {
    // more to come
  }

  protected override void OnStop()
  {
    // more to come
  }
}
```

Starts running as a Windows service

Called when the Windows service is started

Called when the Windows service is stopped

If you're coding along with this example, you'll get compile errors from the preceding code: the type `ServiceBase` isn't known. To load the assembly that contains the `ServiceBase` class, you have to add a line to the `dependencies` section of your project.json file and edit the `frameworks` section to indicate that this application uses the full .NET framework. The `frameworks` section should look like this:

```
"dependencies": {
  "Newtonsoft.Json": "8.0.3",
  "System.ServiceProcess.ServiceController": "4.1.0",
  "System.Net.Http": "4.1.0"
},

"frameworks": {
  "net461": { }
},
```

That should make the application compile again. To run it, you need to install it as a Windows service. And toward that end you need a binary version, so you need to explicitly compile the project. You do that with the `dotnet` command-line tool:

```
PS> dotnet build
```

This compiles the project into a bin folder under the project. You can run the compiled output by calling the compiled executable:

```
PS> .\bin\Debug\net452\LoyaltyProgramEventConsumer
```

Now you have a binary version, and you can install it as a Windows service using the `sc.exe` Windows utility. You must tell `sc.exe` the name of the Windows service and the command to execute as a Windows service. In this case, the command is the `LoyaltyProgramEventConsumer` executable. You end up with this command:

```
PS> sc.exe create loyalty-program-event-consumer binPath="<path-to-
     project>\bin\Debug\net452\LoyaltyProgramEventConsumer"
```

Once `LoyaltyProgramEventConsumer` is installed as a Windows service, you can start and stop it like any other Windows service.

Subscribing to an event feed

You now have a `LoyaltyProgramEventConsumer` console application that you can run as a Windows service. Its job is to subscribe to events from the Special Offers microservice and use the Notifications microservice to notify registered users of special offers. Figure 4.12 shows the collaboration of Loyalty Program, with the ones you've already implemented grayed out.

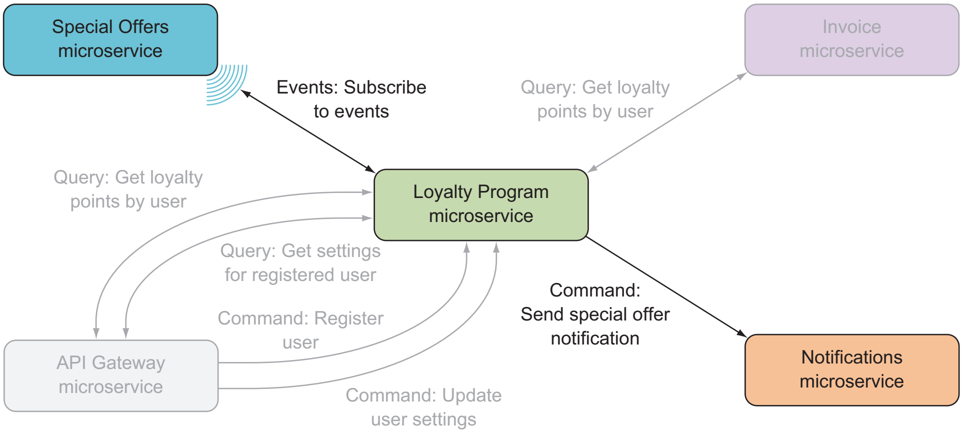

Special Offers microservice
Events: Subscribe to events
Query: Get loyalty points by user
Invoice microservice
Query: Get loyalty points by user
Loyalty Program microservice
Query: Get settings for registered user
Command: Register user
Command: Send special offer notification
API Gateway microservice
Notifications microservice
Command: Update user settings

Figure 4.12 The event-based collaboration in the Loyalty Program microservice is the subscription to the event feed in the Special Offers microservice.

Subscribing to an event feed essentially means you'll poll the events endpoint of the microservice you subscribe to. At intervals, you'll send an HTTP `GET` request to the /events endpoint to check whether there are any events you haven't processed yet.

You'll start the implementation from the top down. The first thing to do is introduce a class called `EventSubscriber` and have it set up a timer that elapses after 10 seconds.

Listing 4.15 Starting a timer and setting up a callback function

```
public class EventSubscriber
{
  private readonly string loyaltyProgramHost;
  private long start = 0;
  private int chunkSize = 100;
  private readonly Timer timer;

  public EventSubscriber(string loyaltyProgramHost)
  {
    this.loyaltyProgramHost = loyaltyProgramHost;
    this.timer = new Timer(10 * 1000);
    this.timer.AutoReset = false;
    this.timer.Elapsed += (_, __) => SubscriptionCycleCallback().Wait();
  }
}
```

Sets up the timer to elapse after 10 seconds

Called every time the timer elapses

After 10 seconds, you check for new events, handle any new events, and then sleep 10 seconds again before checking for new events. Every time the timer elapses, listing 4.15 calls `SubscriptionCycleCallback`, which tries to read new events from the event feed and then handles new events. Both these tasks are delegated to other methods that we'll get to in a moment. For now, here's the code for `SubscriptionCycleCallback`.

Listing 4.16 Reading and handling events

```
private async Task SubscriptionCycleCallback()
{
  var response = await ReadEvents().ConfigureAwait(false);
  if (response.StatusCode == HttpStatusCode.OK)
    HandleEvents(response.Content);
  this.timer.Start();
}
```

Awaits the HTTP GET to the event feed

The `ReadEvents` method makes the HTTP `GET` request to the event feed. It uses `HttpClient`, which you've seen several times already.

Listing 4.17 Reading the next batch of events

```
private async Task<HttpResponseMessage> ReadEvents()
{
  using (var httpClient = new HttpClient())
  {
    httpClient.BaseAddress =
      new Uri($"http://{this.loyaltyProgramHost}");
    var response = await httpClient.GetAsync(
      $"/events/?start={this.start}&end={this.start + this.chunkSize}")
       .ConfigureAwait(false);
    return response;
  }
}
```

Awaits getting new events

Uses query parameters to limit the number of events read

This method reads the events from the event feed and returns them to the `SubscriptionCycleCallback` method. If the request succeeded, the `HandleEvents` method is called. The events are first deserialized, and then each event is handled in turn.

Listing 4.18 Deserializing and then handling events

```
private void HandleEvents(string content)
{
  var events = JsonConvert
    .DeserializeObject<IEnumerable<SpecialOfferEvent>>(content);
  foreach (var ev in events)
  {
    dynamic eventData = ev.Content;
    // handle 'ev' using the eventData.
    this.start = Math.Max(this.start, ev.SequenceNumber + 1);
  }
}
```

❶ Treats the content property as a dynamic object

❷ Keeps track of the highest event number handled

There are a few things to notice here:

- This method keeps track of which events have been handled ❷. This makes sure you don't request events from the feed that you've already processed.
- You treat the `Content` property on the events as `dynamic` ❶. As you saw earlier, not all events carry the same data in the `Content` property, so treating it as dynamic allows you to access the properties you need on `.Content` and not care about the rest. This is a sound approach because you want to be liberal in accepting incoming data—it shouldn't cause problems if the Special Offers microservice decides to add an extra field to the event JSON. As long as the data you *need* is there, the rest can be ignored.
- The events are deserialized into the type `SpecialOfferEvent`. This is a different type than the `Event` type uses to serialize the events in Special Offers. This is intentional and is done because the two microservices don't need to have the exact same view of the events. As long as Loyalty Program doesn't depend on data that isn't there, all is well.

The `SpecialOfferEvent` type used here is simple and contains only the fields used in Loyalty Program:

```
public struct SpecialOfferEvent
{
  public long SequenceNumber { get; set; }
  public string Name { get; set; }
  public object Content { get; set; }
}
```

To tie the `EventSubscriber` code back into the Windows service you set up in listing 4.14 at the beginning of implementing the event-subscriber process, you'll add two more methods to the EventSubscriber: one that starts the timer and one that stops it. These two methods effectively start and stop the event subscription:

```
public void Start()
{
  this.timer.Start();
}

public void Stop()
{
  this.timer.Stop();
}
```

The Windows service can now create an EventSubscriber at startup and then call the Start and Stop methods when the Windows service is started or stopped. Filling in the missing pieces from listing 4.14, the Windows service becomes as follows.

Listing 4.19 Windows service to start and stop the subscription

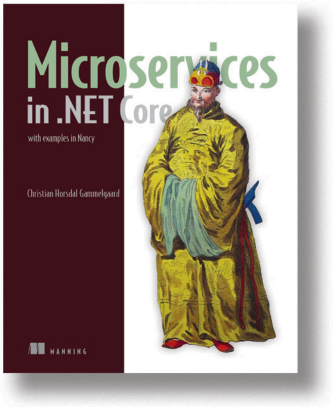
```
public class Program : ServiceBase
{
    private EventSubscriber subscriber;

    public void Main(string[] args)
    {
      this.subscriber = new EventSubscriber("localhost:5000");
      Run(this);
    }

    protected override void OnStart(string[] args)
    {
      this.subscriber.Start();
    }

    protected override void OnStop()
    {
      this.subscriber.Stop();
    }
}
```

This concludes your implementation of event subscriptions. As you've seen, subscribing to an event feed means polling it for new events at intervals and then handling any new events.

4.3 Summary

- There are three types of microservice collaboration:
  - Command-based collaboration, where one microservice uses an HTTP POST or PUT to make another microservice perform an action

  - Query-based collaboration, where one microservice uses an HTTP `GET` to query the state of another microservice
  - Event-based collaboration, where one microservice exposes an event feed that other microservices can subscribe to by polling the feed for new events
- Event-based collaboration is more loosely coupled than command- and query-based collaboration.
- You can hook into Nancy's model binding and content negotiation to support data formats other than XML and JSON.
- The Nancy bootstrapper is used to configure Nancy itself and Nancy applications.
- You can use `HttpClient` to send commands to other microservices and to query other microservices.
- You can use Nancy to expose the endpoints for receiving and handling commands and queries.
- Nancy can expose a simple event feed.
- You can create a process that subscribes to events by
  - Creating a .NET Core console application
  - Implementing and installing a console application as a Windows service
  - Using a timer to make the console application poll an event feed
  - Using `HttpClient` to read events from an event feed



Microservice applications are built by connecting single-capability, autonomous components that communicate via APIs. These systems can be challenging to develop because they demand clearly defined interfaces and reliable infrastructure. Fortunately for .NET developers, OWIN (the Open Web Interface for .NET), and the Nancy web framework help minimize plumbing code and simplify the task of building microservice-based applications.

Microservices in .NET Core provides a complete guide to building microservice applications. After a crystal-clear introduction to the microservices architectural style, the book will teach you practical development skills in that style, using OWIN and Nancy. You'll design and build individual services in C# and learn how to compose them into a simple but functional application back end. Along the way, you'll address production and operations concerns like monitoring, logging, and security.

What's inside

- Design robust and ops-friendly services
- Build HTTP APIs with Nancy
- Expose events via feeds with Nancy
- Use OWIN middleware for plumbing

This book is written for C# developers. No previous experience with microservices required.

Your First Akka.NET Application

When implementing microservices, it is beneficial to choose lightweight yet robust technologies because these technologies support a quick turnaround that benefits continuous delivery and microservices. One such technology in the .NET space is Akka.NET, which is a versatile actor framework that offers an interesting way to implement microservices.

Chapter 3 from *Reactive Applications with Akka.NET* by Anthony Brown.

Your First Akka.Net Application

This chapter covers

- Setting up an actor system
- How to define an actor
- How to send a message to that actor
- A number of alternative actor implementations available to use

The first few chapters covered the key reasons- why you'll likely want to use reactive architecture, as well as what reactive architecture means. We've seen how the overall aim of a reactive system is to create applications which are responsive to the end-user, and how this requires applications to work, even when struggling with the demands of scale or malfunctioning components. We've covered the key things we need to consider when we design a reactive application to ensure our application follows the traits of a reactive application.

From here on, we'll consider how to write reactive systems which follow the traits laid out by the reactive manifesto. As we saw, the reactive manifesto is a series of guidelines designed to suggest solutions to their problems which many organisations have found effective. As such, there are many means of developing reactive systems; we'll focus on one. We'll use the actor model as the underlying basis for our reactive systems, and the implementation is in the form of Akka.Net, a framework designed for writing concurrent applications using the actor model in .Net.

To build these reactive systems we'll write code. Akka.Net runs on the .Net framework and, whilst any language which runs on the .Net framework can use Akka.Net, the main content of this book uses C# to write our applications. Because Akka.Net provides a pragmatic API for F#, which ultimately features some key differences to the C# API, these will be covered in the appendix at the end of the book. The concepts you'll learn in the book will be the same regardless of C# or F#, but the implementation of these concepts will depend upon the language used.

By the end of this chapter you'll have a basic actor created, which can receive messages, and we'll send this actor some messages. You can adapt this actor and build your own actor, capable of performing more complex functions.

3.1 Setting up an application

Akka.Net feels like a framework, but it markets itself as a toolkit and runtime which forms the basis of concurrent applciations. Ultimately, Akka.Net requires no special application configuration to run and can be hosted in any of the normal .Net runtime environments, whether this is console applications, Windows service, IIS, WPF or Windows Forms applications. Throughout this book, examples are given in the form of console applications unless it's been specified otherwise.

All of the components required to run Akka.Net are distributed through the NuGet package management system. As Akka.Net relies on many modern features of the .Net runtime, it requires a minimum of .Net v4.5 to run. Akka.Net also has full Mono support allowing it to run in Linux and Mac OSX environments.

To install the libraries, a NuGet client is required; there are several options available for dependency management with a NuGet client:

- Visual Studio package management GUI: If you're developing applications using Visual Studio then dependencies can be managed directly through the references node of a project in the Solution Explorer.
- Command line tooling: In environments where you don't have access to Visual Studio, a number of command line tooling options are available including the official NuGet client or third-party alternatives such as Paket.

To develop applications in a single machine scenario, the only NuGet package required is the `Akka` package. This provides all of the core functionality which is required to create and host actors and then send messages to these actors.

3.2 Actors

When considering Akka.Net, it's important to realise that the underlying ideas surrounding the framework are those relating to concurrency. Ultimately, the actor model is designed to allow multiple tasks to operate independently of each other. The actor model is designed such that it abstracts away many of the underlying multithreading constructs which ensure concurrency is possible. At the heart of this is the concept of an actor.

3.2.1 What does an actor embody?

The concept of an actor is something which has been discussed several times, but now we can consider what an actor is in the context of Akka.Net. The actor model is a model of computation designed to make concurrency as easy as possible by abstracting away the difficulties associated with threading, including the likes of mutexes, semaphores and other multithreading concepts.

We can think of actors the same way we think of people. Every day we communicate with hundreds or thousands of people in a variety of methods. People send messages to those surrounding them and react to messages they've received. This communication is in the form of message passing, where a message can include several types, such as body language or verbal cues. When a person receives a message, they can process the information and make decisions based on something. The decisions a person makes might include sending a message to the person who originally communicated with it, such as saying "hello" in response to another greeting, or it may be to interact with other parts of the world, such as taste or feel to get more information. Finally, a person is able to save memories or information in their mind. For example, they're able to recognise faces and names or store facts for later recollection.

When we talk about actors, this is the simplest idea. The overall ideas of how people can be broken down into three key concepts which form the basis of the actor model. These three concepts are communication, how they send messages between each other, processing, how the actor responds whenever it receives a new message, and finally state, the information that an actor can store when processing.

Communication

When considering the principles of reactive applications, we saw the advantages of using a message-passing architecture in order to help build systems which are scalable and fault tolerant. By default, all actors within the actor model communicate asynchronously through the use of message passing.

Each actor within an application has a unique identifier through which it can be contacted. We can think of the actor's address exactly like an email address; it provides us with a known endpoint where we can send messages to. Ultimately, the end user can receive their email at any address and the same exists with an actor address. We can send a message to an address and it automatically gets routed to the intended processing for that actor. This address is connected to a mailbox, which is a queue of the messages an actor has received at it's address. This mailbox is where every message gets added as it's received, until the actor is able to process them sequentially.

When we think of an email, it can be one of several types. It can contain the likes of text, media or even contact information. Akka.Net has a similar concept, but it relies on using data types as the basis of messages. We can choose to use any type we like as the basis of our messages. Only one requirement exists; messages must be immutable. If a message isn't immutable, we could potentially modify it either in the processing stage or even in the queue. In either scenario, this breaks the concurrency safety guarantees provided by Akka.Net.

Processing

Once a message has been received, an actor needs to be able to do something with that data. This is the job of the processing component of an actor within Akka.Net. As a message is received, the processing stage is started by the framework, which invokes the appropriate method to handle the object. Akka.Net guarantees only one message will be processed at a time, and due to the queue provided by the framework the processing stage receives the messages in the exact order they were sent to the actor.

Due to the differing programming methodologies supported by Akka.Net, different techniques for using the APIs can be found that fit best with your paradigm. For example, the C# APIs revolve around the use of inheritance.

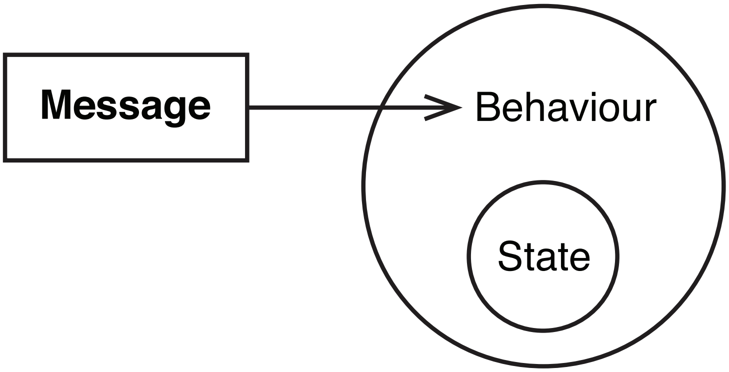
State

When we think back to our analogy of actors as people, we touched on the notion of memories and information saved in their brain. If we want to access the data we can't directly query it from somebody else; we need to ask them about the data they know about. The same concepts apply with actors. An actor is free to store whatever state is appropriate and form a sealed boundary around it. The only thing within the application with access to the data stored on the actor is the processing element associated with that actor.

The primary reason for this is due to the ultimate aims of the use of actors. Actors are a construct which are designed to reduce the complexity of multithreaded applications. By removing shared access to data, it reduces vast numbers of potential concurrency bugs, such as deadlocks or race conditions. It also means we can quickly scale an application built on actors because we can deploy actors into entirely new locations when required.

Combined result

When these three constructs are combined, we're left with the concept of an actor – a high-level approach to dealing with concurrency, whether the tasks running concurrently are on separate threads or in separate datacentres. The diagram below shows the interaction between the three key concepts and how they relate. As you can see, the state is entirely enclosed within the bounds of the actor and isn't accessible from outside of that actor instance. The only means we have of manpulating or retrieving the data is through the use of behavior, which we define within the bounds of the actor. This behavior is only invoked as required once a new message is received by the actor's inbox.



3.2.2 What can an actor do?

We've seen that actors are small, isolated entities which share nothing with the world outside of them, and each is scheduled to process the messages in it's mailbox. We can think of actors as tiny applications with a built-in communication channel. Because of this, actors are able to perform any operation which an application may normally perform. We can generalize the actions that an actor is likely to perform into one of three categories.

- *Sending a message:* When we designed a reactive system, we saw that applications are typically built as a dataflow, whereby applications propagate events which they've received and responded to. In order to manage this, actors need to be able to send messages to other addresses within the actor system. This task isn't necessarily related solely to sending messages to actors within the actor system; it could alse include communication through external services with other transport protocols, such as HTTP.
- *Spawning other actors:* In the case of actors which perform long running computations whilst also needing to process large numbers of messages, it's common to spawn a new actor which is responsible for handling all of the significant processing. For this to happen, actors need to be able to spawn new actors. It also serves uses in other areas, such as having a supervisionary actor spawn new children to perform dangerous work, which may lead to errors.
- *Setting behaviour for the next message:* A key role of an actor is to be able to respond to any messages it receives, whilst reactive applications strive to react to changes in their environment. Ultimately changes in an environment are likely to lead to changes in the way messages need to be processed, and as such actors should be able to set how they should process new messages within the actor.

These are some of the most common tasks actors can typically perform, but it's likely that actors will be performing other tasks as well. This might include jobs such as connecting with external web services, interacting with devices like graphics on the host machine, or potentially interacting with external input and output on the machine it's running on.

A restriction on the type of work that an actor is capable of performing exists. Actors should try to avoid performing long-running blocking operations, particularly in cases where several actors may perform blocking operations at the same time. In this situation, it prevents the Akka.Net scheduler from running any of the processing for other actors. The work that actors do should aim to be asynchronous and operate primarily through message passing. An example of a blocking operation is waiting for a user to enter some text into a console window through the use of Console.ReadLine.

3.2.3 Defining an actor

Having now seen the underlyng principles of actors, we're able to see how the core components fit together. We can now define our actor. Let's think back to our original actor analogy which looked at it's similarity with how we, as people, communicate. Let's build up an example of how we can model this interaction through the use of actors. We'll create an actor which represents the sort of actions a person might take upon receiving a greeting.

When writing an actor in C#, we rely upon inheritance of certain actor classes and override certain methods which get called whenever a new message arrives. The simplest possible means of implementing an actor is through the use of the UntypedActor class. Using this approach, it's possible to execute a single method any time a new message arrives similar to the following.

```
class PersonActor : UntypedActor
{
    protected override void OnReceive(object message)
    {
         Console.WriteLine("Received a message: {0}", message);
    }
}
```

Whilst this example is the basics of how we can write an actor using Akka.Net, it's likely that we'll want to do something with the actor whenever it receives a message. We can use any type within the CLR as a message, with the only requirement being that the class must be immutable. We'll create two potential messages that our person can receive; either a Wave message or a VocalGreeting.

```
class Wave {}

class VocalGreeting
{
    private readonly string _greeting;
    public string Greeting { get { return _greeting; }}

    public VocalGreeting(string greeting)
    {
        _greeting = greeting;
    }
}
```

These are the two message types which our actor is now capable of receiving. Our original actor can now be changed to perform different actions when it receives a message of a given type. For example, when we receive a VocalGreeting message, we can print a message out into the console.

```
   class PersonActor : UntypedActor
   {
       protected override void OnReceive(object message)
       {
           if(message is VocalGreeting)
           {
               var msg = (VocalGreeting)message;
               Console.WriteLine("Hello there!");
           }
       }
}
```

When we're creating a message for each type, we end up with a lot of duplication in the handling of the message. For example, in our example, we've got 2 types of messages, in each instance, we need to check if the message is of a certain type and then cast it to that type. We can also end up with a lot of code duplication when we want to check a condition within the message itself. In order to prevent this, Akka.Net provides an API which allows us to pattern match on the message type. The example below shows how, using the Akka.Net pattern matching API, we can invoke a handler dependent upon the message received.

```
    class PersonActor : UntypedActor
   {
        protected override void OnReceive(object message)
        {
               message.Match()
                   .With<VocalGreeting>
                       (x => Console.WriteLine("Hello there"));
        }
}
```

Akka.Net also provides a further abstraction on top of the basic actor which we can use to declaratively handle messages. The ReceiveActor combines many of the aspects of pattern matching whilst continuing to abstract away as much of the logic surrounding message type handling as possible. With the UntypedActor we had to override a certain method, which would be executed upon receipt of a message. The ReceiveActor requires us to register a message handler for each of the given message types we want to support. The example below shows how the previous example using an UntypedActor can be converted to the ReceiveActor implementation.

```
   class PersonActor : ReceiveActor
   {
       public PersonActor()
       {
           Receive<VocalGreeting>
               (x => Console.WriteLine("Hello there"));
       }
}
```

Akka.Net is a model for concurrently performing asynchronous operations, and is an alternative to the .Net Task Parallel Library (TPL). Typically, when dealing with asynchronous operations, we'll pipe the results back to the actor's mailbox as a message, but the ReceiveActor provides the ability to interoperate with the TPL through asynchronous message handlers. An asynchronous message handler works exactly the same as a regular message handler, except it returns a Task instead of void.

```
    class PersonActor : ReceiveActor
    {
        public PersonActor()
        {
            Receive<VocalGreeting>(async x =>
                {
                    await Task.Delay(50);
                    Console.WriteLine("Hello there");
                });
        }
}
```

The approaches shown for creating actors have relied upon the use of delegates as a means of handling messages. But Akka.Net provides an additional means of creating actors in the form of the `TypedActor`. The `TypedActor` allows for stricter contracts to be built up for the types of messages an actor should be able to receive by implementing an interface for each of them. Upon receiving a message of a given type, the corresponding method implementing the interface for that message type is executed with an instance of the received message.

```
    class PersonActor : TypedActor,
                        IHandle<VocalGreeting>
    {
        void Handle(VocalGreeting greeting)
        {
            Console.WriteLine("Hello there");
        }
    }
```

All of the actor defintions here allow us to build up bigger and more advanced actors, capable of performing more complex operations. We saw in our definition of an actor that we're able to store state within an actor. The actor definitions are classes in C# which override specific methods. We're able to store state within an actor using either properties or fields on the class.

When we store any state within an actor, it's only accessible from within that actor. It's impossible to access any properties or fields from outside the actor boundaries. This means that regardless of where an actor exists, there's no need to worry about synchronising access to the state, because messages are only processed one at a time.

```
    class PersonActor : ReceiveActor
    {
        private int _peopleMet = 0;
        public PersonActor()
        {
            Receive<VocalGreeting>(x =>
                {
                    _peopleMet++;
                    Console.WriteLine("I've met {0} people today",
                                     _peopleMet);
                });
        }
}
```

Upon receiving a message, it's common to require metadata about either the message which was received, such as the original address of the sender, or about the actor processing the message, such as the address behaviour stored within the actor. Within any of the actor types, we can access this through the `Context` property within an actor. For example, if we wanted to retrieve the original sender of the message, we can access this through the `Sender` property on the context. Given the sender, we can send messages as a response to a message we received. For example, if somebody waves at us, then we'll wave back at them by sending them a Wave message.

```
class PersonActor : ReceiveActor
    {
        public PersonActor()
        {
            Receive<Wave>(x =>
                {
                    Context.Sender.Tell(
                        new VocalGreeting("Hello!"));
                });
        }
     }
```

Many ways can be used to define actors which are specific to certain aspects of Akka.Net, and we'll cover those in later chapters.

3.2.4 Summary

When we discussed design considerations within a reactive system, one of the key considerations was that operations should be done within the smallest unit of work. In the context of Akka.Net, the actor's the encapsulation around that smallest unit of work. One of the key takeaways when dealing with actors is that, due to it's original design intentions as a concurrency model, any operations within the confines of an actor are thread safe. This ensures that we're able to automatically scale out our application across as many threads, machines or datacentres as we like, and the framework can handle any and all scaling issues. This is handled by messages being processed one at a time through a queue, ensuring that messages are processed in the order they're received.

3.3 Spawning an actor

Having defined an actor, we need to be able to start it running within our application. In order to do this, we'll start to dig into the underlying framework and look at how we can use Akka.Net to start instances of actors that can react to messages we send it. In order to do this we'll look at the concept of an actor system and what needs to be done to deploy an actor into this actor system.

3.3.1 The actor system

If actors are people then actor systems are the countries within which they live. The actor system is the host within which all of your actors will be deployed. Once actors are deployed, they're able to perform any tasks which have been assigned to them. Like people and government, actors need some form of management and restrictions in place to ensure that they are good and valuable citizens within society. These tasks fall within the realm of the ActorSystem, which isn't only our actor host, but also the scheduler and routing system. You don't need to know about the internals of the actor system to be able to develop applications with Akka.Net, as it abstracts all of that away from the user. There's more to it than those few elements, but some of the key roles it's in charge of include:

- *Scheduling:* Actors as a multithreading construct run at a higher level than a regular thread, and as such there needs to be some means of coordinating these actors. The actor system ensures that all messages have a fair chance at processing their messages within a reasonable amount of time. In contrast, it also ensures that heavily-used actors aren't able to starve the system of resources, causing less frequently used actors to be unable to processs data.
- *Message Routing:* All of the messaging through Akka.Net is location transparent, meaning the caller doesn't need to have any knowledge of the location of the recipient. There must be some part of the system with knowledge of message locations, and this is the actor system. The actor system's capable of routing messages to a large number of different locations whether they're on a separate thread, running on a remote system, or running on a machine in a cluster.
- *Supervision:* The actor system also acts as the top level supervisor of your application, able to recover any component which has crashed. We'll look into this in a later chapter as we look to incorporate the notion of fault tolerance into our application.
- *Extensions:* Akka.Net supports a vast range of extensibility points throughout the processing pipeline. The actor system's also responsible for managing all of these extensibility points and ensuring that any extensions are correctly incorporated into the application.

This is a small subset of the large number of tasks the actor system's responsible for, and as such it's common to have only one running per application. Actor systems are identified on a machine by using a unique name, meaning that it's possibile for more than one actor system to exist on each machine.

Actors in Akka.Net operate under the concept of a hierarchy, whereby all actors are the children of one other actor in the hierarchy. The reasoning for this is to allow for easier fault tolerance when developing applications, and the intricacies of this will be covered in a later chapter on fault tolerance. When instantiating an actor system, Akka.Net initially creates a number of actors used by the system. These top level actors are:

- `/user`: This actor holds all of the actors which you spawn into your actor system. Even if you spawn your actor without a parent, it has a parent in the form of the user actor which performs supervision of your top-level actors.
- `/system`: This actor is the top-level actor under which all of the system level actors are stored. This is typically those actors which are used for tasks such as logging or those deployed as part of some configuration.
- `/deadletters`: As actors are free to send messages to any address at any stage of the application, there's always the possibility that no actor instance's available at the path specified. In this case, the messages are directed to the dead letters actor.
- `/temp`: At times Akka.Net spawns short lived actors. This is typically for scenarios such as retrieving data, which will be covered later in the chapter.
- `/remote`: When joining multiple actor systems using Akka.Net remoting, there are some scenarios whereby Akka.Net needs to create actors to perform the task of supervisors when a supervisor exists on a separate machine. In these cases, the remote top level actor is used; this will be covered in a later chapter.

These actors all form part of the hierarchy, in the figure below, you can see the deployment of them into the hierarchy. The actors themselves form a tree structure similar to a file system with files and folders. The figure below shows an example deployment of a simple actor system. In this case, the user has deployed three actors into the actor system, actorA and actorB where actorA has a child actor spawned beneath it which is known as childA.

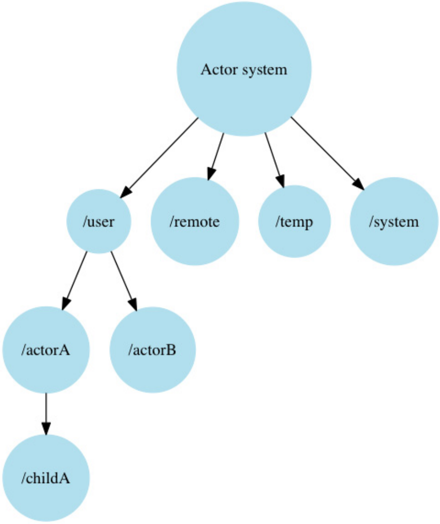


Figure 3.1 An Example actor hierarchy within an Akka.Net application.

The decision to use actors for all top-level work within Akka.Net itself ensures that a uniform interface exists throughout the application. In any of these system-created actors, users are free to send a message to them in the same way that a user might expect to send a message to an actor which they've instantiated.

3.3.2 Spawning an actor

Now that we've defined an actor which is able to work, we need to deploy it into an application to use it. Before we're able to deploy our actor, we need something which is capable of hosting an actor. In order to do this, we need to initialise an actor system. As we've seen, the actor system is the component of Akka.Net which is responsible for many of the tasks relating to how actors are run within the framework.

Instantiating an actor system in which we can host actors is a simple task and it requires calling the Create static method on the actor system. The only requirement when creating an actor system is to name it in a way that actors can be identified based on which actor system they live in.

```
var actorSystem = ActorSystem.Create(“MyActorSystem”);
```

An actor system can also be created with a configuration section in order to customise certain parts of the framework. This will be covered in a later chapter. For now, we create an actor system without a configuration file in C#, which causes a fallback onto the default configuration.

As we've seen, the actor system is responsible for many of the internal tasks and scheduling associated with the Akka.Net framework. Because of this the actor system ends up becoming a heavyweight object, and we typically only spawn one actor system per application. The actor system is the main means of interacting with the actors operating within the framework. In most scenarios, it's typical for the actor system to either reside in a static class or singleton object, or be injected as a dependency into those methods which require it.

Once an actor system has been created, we're free to deploy new actors into it. To deploy an actor into the actor system we use the ActorOf method, which requires the actor type to instantiate as a generic type argument. The example below shows how we can deploy our actor from earlier into the actor system to interact with it.

```
var actorRef = actorSystem.ActorOf<GreeterActor(“actorA”);
```

Once this method has been called, Akka.Net will create and initialize this new actor into the actor system. We pass it a string, which we can use to uniquely identify a given actor instance within the actor system. In this case we've chosen to refer to the actor as actorA. This means that given this name, we're able to retrieve references to it directly from the actor system.

3.3.3 Summary

The actor system forms the basis of your host within which your actors live. Whilst you don't need to understand all of the intricacies of what happens deep within the framework, it’s beneficial to have an understanding of some of the features provided by the actor system. The actor system is also the key extensibility point of an Akka.Net application and allows more advanced features to be implemented, many of which we’ll look at in later chapters.

3.4 Communicating with actors

Once you’ve spawned an actor into your actor system, you’ll want to be able to communicate with it. Whilst we have an actor deployed into our actor system, it’s currently doing no work and sits in memory, doing nothing. By communicating with it, the framework will invoke the message processign on that actor. The actor model relies upon message passing as a means of communication between actors. A message is a generic term for a collection of data which is packaged and sent to an actor instance somewhere in the actor system, as represented by the address. We saw in our example earlier that our messages will be a data type we’ve created.

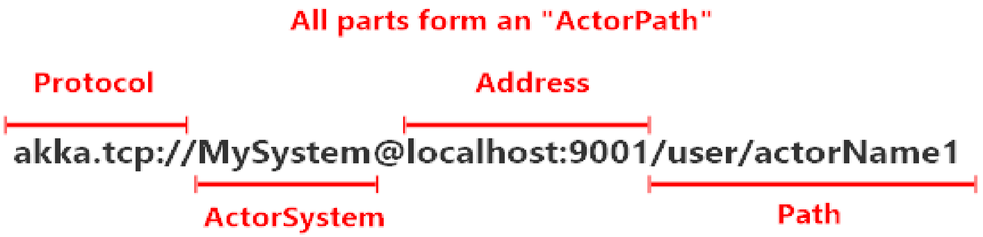
3.4.1 Actor addresses and references

Upon spawning our actor, the actor system returned a direct reference to the actor through an IActorRef. This actor reference isn’t a direct reference to the actor’s location in memory, but is a reference to the actor as used by Akka.Net. It’s ultimate use is to facilitate sending messages to the inbox of the referenced actor. The Akka.Net framework provides a number of built-in means of referencing actors out of the box. These include the likes of actor references for clusters and remote actor systems. We won’t be seeing these until later chapters.

The most commonly used actor reference is LocalActorRef, whose job is to operate on actor systems that only operate on a single machine. The key component of the actor reference is the storage of the address of the actor itself. Upon deployment, every actor is given a unique address through which the actor is reachable. The address is reminiscent of a URI which might be used to identify files in a file system or web pages on a web site. In this case it represents the address of an actor in an actor system. Figure <X> below shows the components of an address. An actor address is made up of four key components:

- *Protocol identifier*: The protocol identifier is used to reference how a connection is made to that actor system. This is similar to how http and https are used in web addresses to identify which system should be used. For a single machine, this is typically through an identifier like akka://, but for handling concepts such as remoting, there are other commonly used examples, such as akka.tcp://.
- *Actor system name:* When we created an actor system, we gave it a unique name to refer to that actor system instance. This part of the address relates to that name.
- *Address:* This is only used when dealing with the concept of remoting, but it still forms a key part of the actor path and is used to identify the machine upon which an actor system resides.

- *Path:* The final part of the address is the path, which is used to identify an actor. All user defined actors start with the /user/ for this part of the path, but other system defined actors inhabit other root addresses.


All parts form an "ActorPath"
Protocol
Address
akka.tcp://MySystem@localhost:9001/user/actorName1
ActorSystem
Path

The concept of an actor reference starts to ensure that our application is loosely coupled, but it still causes problems. In order to send a message to a given actor reference, we'll need to pass the actor reference around the application. When we considered the benefits of a message-driven architecture, one of the more important benefits was the ability to have loosely-coupled systems which didn't rely upon intimate knowledge of other actors. In order to solve this problem with Akka.Net, we're able to send messages to an address rather than an actor reference directly. Given an address we're able to send a message to that address. For example, to send a message to an actor known as ActorA in our actor system, we're able to retrieve a reference to it's address as below.

```
var address = system.ActorSelection("/user/ActorA");
```

When we deployed our actor, we saw that it was deployed into a hierarchy. If we deployed our actor as the child of another actor, then we can continue to address it similar to how we find files which are within a folder in a file system. If ActorA has a child actor called Child, then we can send messages to it as follows.

```
var childAddress = system.ActorSelection("/user/ActorA/Child");
```

The addressing system within Akka.Net also respects the usage of certain path traversal elements which are typically be associated with URIs. For example, a common case is to retrieve the parent of the current actor, to allow messages to be sent to a sibling of the current actor. This can be achieved by using the .. syntax to retrieve the parent within an actor as follows.

```
var address = Context.ActorSelection("../ActorB");
```

Whilst it might seem that the concept of an actor selection and an actor reference are the same, there's a significant difference, in that an actor reference points to a specific incarnation of an actor whilst an actor selection points to an address. This address may be shared with multiple instantiations of an actor. For example, given a reference to a specific actor, if that actor is destroyed and recreated, then any messages sent to that actor reference won't be delivered to the target, even if they both share the exact same path across instantiations. Given an actor selection, messages can be sent to it even if an actor is destroyed and recreated; all messages will be delivered.

This distinction allows for more complex paths to be used in the context of an actor address. An example of this is through the use of wildcards in the path to a given actor in order to select large numbers of actors at once. Once these actors have been selected, it's possible to send the same message to all in the wildcard with a single method call. Paths in Akka.Net support two kinds of wildcards in an actor address based on standard wildcard syntax common across other languages and tools:

- `?`: The question mark replaces a single instance of any given character in a path. For example, the path `c?t` would match paths such as cat, but not coat or cost.
- `*`: The asterisk matches any string of characters usable as a path. For example, the path /parent/*/ would send a message to all children of the actor called parent.

On occasion it's beneficial to have a direct reference to an actor instance rather than a generic address. In order to cater to these situations, Akka.Net provides a number of different means of retrieving a reference from an address.

- *Calling ActorOf to spawn a new actor:* Upon spawning a new actor, a direct reference to that actor is returned which represents the incarnation which has been spawned.
- *Sending a message to an actor:* By sending a message to an actor, it's possible to use the sender property of a received message to identify which actor replied to the request for information. Akka.Net provides built-in support for this through the Identify message, and through an abstraction on the ActorSelection, which can be used to resolve an instance.

Whilst there's many cases whereby it's appropriate to send messages to an address, it can frequently be beneficial to pass around a reference to a specific actor. For example, given a long-running actor which is valid throughout the lifecycle of the application and performs a specific purpose, it's typical to pass an actor reference in the constructor of those actors that depend on it.

It's important to understand the difference in an actor reference and an actor address due in part to the actor lifecycle, something covered in a later chapter. For our uses either option is an appropriate means of messaging a specific actor.

3.4.2 Sending a message

Upon spawning our actor into the system, we're able to communicate with it by sending messages to it's mailbox. In order to send a message to it, we need something capable of receiving a message. As we saw in the differences between an address and a reference, we're able to send a message to either. Once an actor is spawned, the actor system returns a reference to that actor instance which we can send a message to. The actor reference defines a method called `Tell` which takes an instance of any type and passes it through the Akka.Net framework. If you're using F#, there's a custom operator defined for sending a message. For example, if we wanted to send a vocal greeting message to the actor we defined earlier then we can do it as follows.

```
actorRef.Tell(new VocalGreeting("Hello"));
```

There may be times we don't have an actor reference and on those occasions we'll look up an actor by it's address. In order to look up an actor, we need something capable of providing references to other actors. This may be the actor system hosting the actor, or it may be the Context associated with a specific actor. In order to select the actor deployed earlier, we can use the actor system to select the actor by address.

```
var selection = actorSystem.ActorSelection("actorA");
```

In each of these cases, the actor system provides the root location from which actors will be retrieved, which for the actor system is directly beneath the user actor. If we'd a second actor deployed alongside our first, we could use our first actor reference as our anchor to other actor locations.

```
var selection = actorRef.ActorSelection("../actorB");
```

Once we've got an address, we can pass messages to it in the same way we do with an actor reference.

```
selection.Tell(new VocalGreeting("Hello"));
```

Actors are designed to completely encapsulate any state to ensure that nothing outside of the system is capable of mutating it. This ensures that Akka.Net retains full control over the processing stage, by only allowing one message to be processed at a time. This leaves all code thread-safe, but it adds difficulty to access the data. To access data from outside the system, we need to send a message that specifically requests the data to be sent back. Akka.Net provides another method which allows for request-reply scenarios to be used through the use of Ask. Ask is an asynchronous method designed to form a layer of abstraction over the top of the messaging which is required.

```
var response = await selection.Ask(new Wave());
```

As Ask is an asynchronous construct, in your code you'll need to factor the length of time it takes to get a response. By default, Ask has a timeout of ten seconds within which the actor needs to respond to your initial request message, otherwise the request times out with an exception. It's important to realise that your actor has no way of knowing that the sender is expecting a reply, and it's down to how you, the developer, handle this scenario.

3.4.3 Summary

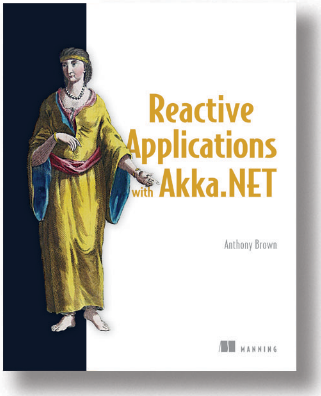
Messages form an integral part of the design of a system using Akka.Net and are the key to communication between multiple actors, or even other entities outside of the actor system. As such it's important to model your domain effectively through the commands, which actors will need to be able to respond to. In later chapters, we'll look at techniques such as Event Sourcing and Domain Driven Design as a means of modeling certain interactions between actors. At this stage it's likely that most actors will react to events or respond to commands.

Whilst the name message is used, Akka.Net doesn't require anything special with regards to the design of a message and they can be .Net classes or structs. The only requirement when designing these messages is that they be immutable to ensure that the thread safety guarantees specified by Akka.Net can't be broken anywhere within the application.

3.5 Summary

This chapter has covered the basics of creating the core components of an application built on top of the actor based concurrency provided by Akka.Net. The rest of the book focuses on how we can use these components and build on top of them to create more advanced applications which follow the traits specified by the reactive manifesto. In this chapter you've seen:

- How to define an actor and how each part relates to the actor model
- How to deploy that actor within your application
- How to communicate with that actor by passing messages



Developing applications in a reactive style ensures that the experience is always responsive. Akka.NET is a framework for building distributed, message-driven applications which are able to stay responsive for the user even in the face of failure or when faced with more users. It makes it easy for .NET developers to write applications which are able to react to changes in their environment.

Reactive Applications with Akka.NET begins with an overview of reactive and a sample application written in the reactive style. You'll learn concepts of the actor model and what these mean in a real-world reactive context. This hands-on book builds on fundamental concepts to teach you how to create reliable and resilient applications. You'll also learn useful Akka.NET features for building real-world applications. By the end of the book, you'll be able to look at a problem domain and understand how to create applications which are able to withstand the demands of a modern day application.

What's inside:

- Real-world applications of Akka.NET and reactive applications
- Designing an Internet of Things architecture with reactive in mind
- Building applications to handle the demands of the modern era of software
- Integrating Akka.NET with your existing .NET stack
- Deploying Akka.NET applications to the cloud

Readers should be comfortable with C# or F# and the .NET framework. No previous reactive experience needed.

Deployment

Microservices are not very interesting if they only run on developer machines. They must be deployed to a production environment to provide business value. This chapter outlines how to approach deployment specifically within a microservice context.

Chapter 5 from *The Tao of Microservices* by Richard Rodger.

Deployment

This chapter covers

- Understanding nature of failure in complex systems
- Developing a simple mathematical model of failure
- Taking a realistic risk-management perspective
- Using frequent low-cost failure to avoid infrequent high-cost failure
- Using continuous delivery to measure and manage risk
- Understanding the deployment patterns for microservices
- Reviewing the wider considerations of microservices in production

The organizational decision to adopt the microservice architecture often represents an acceptance that change is necessary and that current work practices aren't delivering. This is an opportunity to not only adopt a more capable software architecture, but also to introduce a new set of work practices for that architecture.

You can use microservices to adopt a scientific approach to risk management. Microservices make it easier to measure and control risk, as they give you small units of control. The reliability of your production system then becomes quantifiable, allowing you to move beyond ineffective manual sign-offs as the primary risk reduction strategy. Because traditional processes regard software as a bag of features that are either broken or fixed, and don't incorporate the concept of failure thresholds and failure rates, they are much weaker protections against failure[1] .

5.1 Things fall apart

Things fail catastrophically. The decline isn't gradual. Decline is certain, but when death comes, it comes quickly. Structures need to maintain a minimum level of integrity before they fall apart. Cross that threshold, and the essence is gone.

This is more than poetic symbolism. Disorder always increases[2]. Systems can tolerate some disorder, and can even convert chaos into order in the short term, but in the long run, we're all dead, because disorder inevitably forces the system over the threshold of integrity into failure.

What is failure? From the perspective of enterprise software, this question has many answers. Most visible are the technical failures of the system to meet up-time requirements, to meet feature requirements, and to meet acceptable performance and defect levels. Less visible, but more important, are failures to meet business goals.

Organizations obsess about technical failures, often causing business failures as a result. The argument of this chapter is that it's better to accept many small failures to prevent large scale catastrophic failures. It's better that 5% of users see a broken web page than that the business goes bankrupt having failed to compete in the marketplace. Nothing lasts forever, but you can last long enough.

The belief that software systems can be free from defects, and that this is possible through sheer professionalism, is pervasive in the enterprise. An implicit assumption that perfect software can be built at a reasonable cost prevails. This is to ignore the basic dynamics of the law of diminishing marginal returns: the cost of fixing the next bug grows ever higher, and is unbounded. In practice, all systems go into production with known defects. The danger of catastrophe comes from an institutional consensus to pretend that this isn't the case.

Can the microservice architecture speak to this problem? Yes, because it makes it easier to reduce the risk of catastrophic failure by allowing you to make small changes that have low impact. The introduction of microservices also provides you, as an architect, with the opportunity to re-frame the discussion around acceptable failure rates, and the management of risk. Unfortunately, there's no forcing function, and microservice deployments can easily become mired in the traditional risk management approach of enterprise operations. It's essential to understand the possibilities for risk reduction that the architecture creates.

[1] To be bluntly cynical, traditional practices are more about territorial defense and blame avoidance than building effective software.

[2] More ways exist to be disorganized than to be organized. Any given change is more likely to move you further into disorder.

5.2 Learning from history

To understand how software systems fail, and how we can make deployment better, we need to understand how complex systems fail. A large scale software system isn't unlike a large scale engineering system; numerous components interact in many ways. With software, we've the additional complication of deployment—we keep changing the system. At least with something like a nuclear power plant, you only build it once. Let's start by examining such a complex system, in production.

5.2.1 Three Mile Island

On March 28 1979 the second unit of the nuclear power plant located on Three Mile Island near Harrisburg, Pennsylvania suffered a partial meltdown, releasing radioactive material into the atmosphere[1]. The accident was blamed on operator error. From a complex systems perspective, this conclusion is neither fair nor useful. With complex systems, failure is inevitable, and it's only a matter of time. The question isn't, "is nuclear energy safe?", but rather "what level of accidents and contamination can we live with?". This is also the question we should ask of software systems.

To understand Three Mile Island, you need to understand how a reactor works, at a high level, and a low level, where necessary. Your skills as a software architect will serve you well. The reactor heats water, turning it into steam. The steam drives a turbine, that spins to produce electricity. The reactor heats the water using a controlled fission reaction. The nuclear fuel, uranium, emits neutrons that collide with other uranium atoms, releasing even more neutrons. This is a chain reaction that must be controlled by absorbing excess neutrons, otherwise bad things happen.

The uranium fuel is stored in a large, sealed, stainless steel containment vessel, about the height of three story building. The fuel is stored as vertical rods, about the height of a single story. Interspersed are control rods, made of graphite. These absorb the neutrons. To control the reaction, you raise and lower the control rods. The reaction can be completely stopped by lowering all the control rods fully. This is known as "scramming". An obvious safety feature is that, if there's a problem, pretty much any problem, drop the rods![2]. Nuclear reactors are designed with many such Automatic Safety Devices (ASD) that activate without human intervention, caused by the input signals from sensors. You can see the opportunity for unintended cascading behavior in the ASDs already, I'm sure.

The heat from the core (all the stuff inside the containment vessel, including the rods) is extracted using water. This coolant water is radioactive, and you can't use it directly to drive the turbine. You must use a heat exchanger to transfer the heat to another set of water pipes, and that water, which isn't radioactive, drives the turbine. You've a primary coolant system, with radioactive water, and a secondary coolant sys-

[1] For full details, see the Report of the President's Commission on the Accident at Three Mile Island; http://www.threemileisland.org/downloads/188.pdf

[2] The technical term "scram" comes from the early days of research reactors. If any went wrong, you dropped the rods, shouted "scram", and then you ran. Fast.

tem, with "normal" water. Everything's under high pressure, and high temperature, including the turbine, which is itself cooled by the secondary system. The secondary water must be pure, and contain almost no microscopic particles, to protect the turbine blades, which are precision engineered. Observe how complexity lives in the details. A simple fact, that water drives the turbine, hides the complexity that it must be "special" purified water. Time for a high-level diagram:

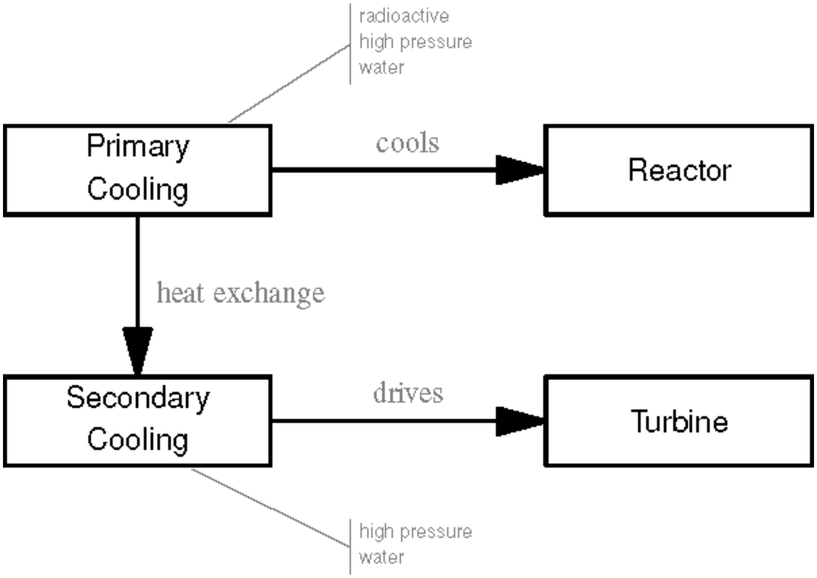


Figure 5.1 High level components of a nuclear reactor

Now we go a little deeper. That "special" secondary water doesn't happen by magic. You need something called a *condensate polisher* to make it happen. This purifies the water using filters. Like many parts of the system, the condensate polisher's valves, which allow water to enter and leave, are driven by compressed air. That means the plant, in additional to water pipes for the primary and secondary cooling systems, also has compressed air pipes for a pneumatic system. Where does the secondary water come from? Feed pumps are used to pump water from a local water source, in this case, the Susquehanna river, into the cooling system. Emergency tanks, with emergency feed pumps, are there in case the main feed pumps fail. The valves for these are also driven by the pneumatic system.

We must also consider the core, filled with high temperature radioactive water under high pressure[1]. High pressure water is extremely dangerous and can damage the containment vessel, and the associated pipe-work, leading to a dreaded Loss of Containment Accident (LOCA). You don't want holes in the containment vessel. To alleviate water pressure in the core, a *pressurizer* is used. This is a large water tank connected to the core and filled about half and half with water and steam. The pressur-

[1] What fun!

izer itself also has a drain, which allows water to be removed from the core entirely. The steam at the top of the pressurizer tank is compressible, and acts as a shock absorber. You can control core pressure by controlling the volume of the water in the lower half of the pressurizer. But you must never ever allow the water level to reach 100% (this is called, "going solid"). Then you've no steam, and no shock absorber, and then you're going to have a LOCA, because pipes will burst. This fact is drilled into operators from day one. We can now expand our diagram.

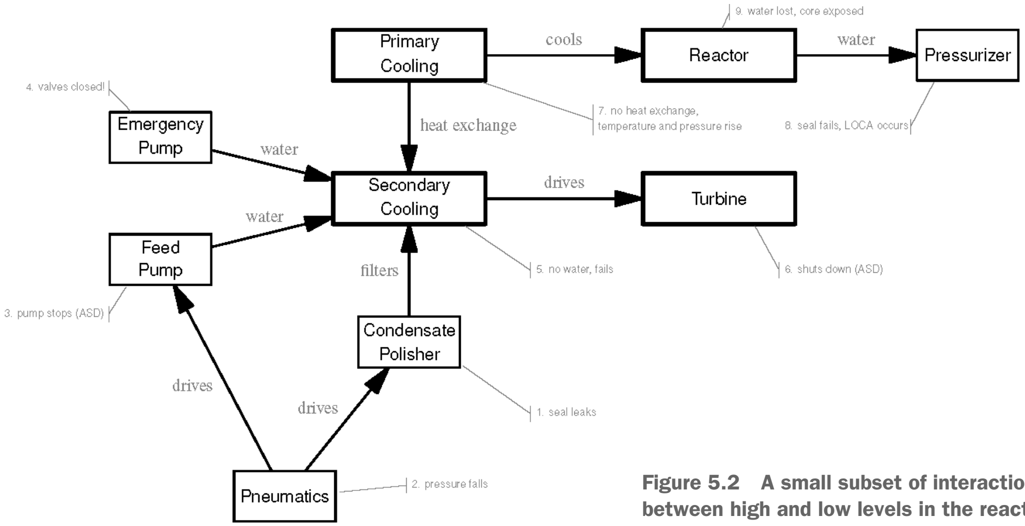


Figure 5.2 A small subset of interactions between high and low levels in the reactor.

The timeline of the accident

At 4:00 AM the steam turbine "tripped". It stopped automatically because the feed pumps for the secondary cooling system that cools the turbine stopped. With no water entering the turbine, the turbine was in danger of overheating, and is programmed to stop under these conditions. The feed pumps had stopped because the pneumatic air system that drives the valves for the pumps became contaminated with water from the condensate polisher. A leaky seal in the condensate polisher allowed some of the water to escape into the pneumatic system. The end result was that a series of ASDs (Automatic Safety Devices), operating as designed, triggered a series of ever larger failures. More was to come.

With the turbine down, and no water flowing in the secondary coolant system, no heat could be extracted from the primary coolant system. The core couldn't be cooled. This is extremely dangerous, and if not corrected, ends with a meltdown.

There was an ASD for this scenario. Emergency feed pumps take water from an emergency tank. The emergency pumps kick in automatically. Unfortunately, the pipes to the emergency pumps were blocked, because two valves were left closed, in error, during recent maintenance. The emergency pumps supplied no water. The complexity of the system and its inter-dependencies become apparent here. It isn't only the machinery, but also the management and maintenance thereof, which is part of the dependency relationship graph.

The system entered a cascading failure mode. The steam turbine boiled dry. The reactor "scrammed" automatically, dropping all the control rods to stop the fission reaction completely. This doesn't reduce heat to safe levels, as the decay products from the reaction still need to cool down. Normally this takes several days, and requires a functioning cooling system. With no cooling, extremely high temperatures and pressures build in the containment vessel, which is then in danger of breaching.

Naturally, there are ASDs for this scenario. A relief valve, known as the Pilot-Operated Relief Value (PORV) opens under high pressure, and allows the core water to expand into the pressurizer vessel. The PORV is unreliable, because valves for high pressure radioactive water fail about 1 time in 50 The PORV opened in response to the high pressure conditions, but then failed to close fully after the pressure was relieved. The status of the PORV is important for the operators to know, and it had recently been fitted with a status sensor and indicator. This sensor also failed, leading the operators to believe that the PORV was closed. The reactor was now under a Loss of Containment Accident (LOCA), and over one-third of the primary cooling water drained away over the next 2 hours and 20 minutes. The actual status of the PORV wasn't noticed until a new shift of operators started.

As water drained away, pressure in the core reduced, but by too much. Steam pockets formed. These not only block water flow, but are also far less efficient at removing heat. The core continued to overheat. At this point, we are now only **5 seconds** into the accident, and the operators are completely unaware of the LOCA, seeing only a transient pressure spike in the core. At two minutes into the event, pressure dropped sharply as core coolant turned to steam. At this point the fuel rods in the core were in danger of becoming exposed, as there was barely sufficient water to cover them. Another ASD kicked in—injection of cold high pressure water. This is a last resort to save the core by keeping it covered. The problem is that too much cold water can crack the containment vessel. Also, and far worse, too much water makes the pressurizer "go solid". Without a pressure buffer, pipes would crack. The operators, as they were trained, slowed the coldwater injection rate[1].

The core became partially exposed as a result, and a partial meltdown occurred. Although the PORV was eventually closed, and the water brought under control, the core was badly damaged. Chemical reactions inside the core led to the release of hydrogen gas, which caused a series of explosions, and ultimately radioactive material was released into the atmosphere[2].

[1] Notice that the operators were using a mental model that diverged from reality. Much the same happens with the operation of software systems under high load.

[2] An excellent analysis of this accident, and many others, can be found in the book **Normal Accidents** (1999), Princeton University Press, by Charles Perrow. This book also develops a failure model for complex systems which is relevant to software systems.

Learning from the accident

Three Mile Island is one of the most studied complex systems accidents. Some blame the operators, who "should" have understood what was happening, and who "should" have closed the valves after maintenance, who "should" have left the high pressure cold water injection running[1]. Have you ever left your home and can't remember if you've locked the front door? Imagine having 500 front doors. On any given day, in any given reactor, some small percentage of valves will be in the wrong state. Others blame the sloppiness of the management culture. There should've been lock sheets for the valves. But more paperwork to track work practices has only reduced valve errors in other reactors, not eliminated them. Some blame the design of the reactor. Too much complexity and coupling and inter-dependence. A simpler design has fewer failure modes, but hidden complexity is inherent to systems engineering, and truly simple designs aren't possible.

None of these judgments are useful, because they're all obvious and true to some degree. The real learning is that complex systems are fragile, and will fail. No amount of safety devices and procedures can solve this problem, because the safety devices and procedures are **part** of the problem. Three Mile Island makes this clear. The interactions of all the components of the system (including the humans) led to failure.

This is a clear analogy to software systems. We build architectures that have a similar degree of complexity, the same kinds of interactions and tight couplings. We try to add redundancy and fail safes, and then find that they fail anyway, as they haven't been sufficiently tested. We try to control risk with detailed release procedures and strict quality assurance, and still end up having to do releases at the weekend, with inevitable downtime. In one way, we are worse than nuclear reactors—with every release we change fundamental core components!

You can't remove risk by trying to contain complexity. Eventually you'll have a LOCA.

5.2.2 A model for failure in software systems

Let's try to understand the nature of failure in software systems using a simple model. We need to quantify our exposure to risk to understand how different levels of complexity and change affect a system.

A software system can be thought of as a set of components, with dependency relationships between the components. The simplest case is a single component. Under what conditions does the component, and the entire system, fail? To answer that question, we should clarify the term **failure**. In this model, failure isn't an absolute binary condition, but a quantity we can measure. Success might be 100% up-time over a given period, and failure is any up-time less than 100%. But we could be quite happy

[1] Or should they have? It might have cracked the containment vessel, causing an accident far worse. Expert opinion is conflicted on this point.

with a failure rate of 1%, giving us 99% up-time as the threshold of success. We could count the number of requests that have correct responses. Out of every 1000 requests, perhaps 10 fail, and we've a failure rate of 1%. Again, we could be quite happy with this. Loosely, we can define the failure rate as the proportion of some quantity (continuous or discrete) that fails to meet a specific threshold value. Remember that we are building as simple a model as we can, and what the failure rate is a **failure of** is excluded from the model. All we care about is the rate, and meeting the threshold. Failure is failure to meet the threshold, not failure to operate.

For our one component system, if the component has a failure rate of 1%, then the system has a failure rate of 1%. Is the system failing?

If the acceptable failure threshold is 0.5%, then the system is failing. If the acceptable failure threshold is 2%, then the system *isn't* failing, it's succeeding, and we can go home.

This model reflects an important change of perspective: accepting that software systems are in a constant state of low-level failure. A failure rate always exists. Valves are always left closed somewhere. The system fails only when a threshold of pain is crossed. This new perspective is different from the embedded organizational assumption that software can be perfect and operate without defects. The obsession with tallying defective features seems quaint from this viewpoint. Once you gain this perspective, you can begin to understand how the operational costs of the microservice architecture are out-weighed by the benefit of superior risk management.

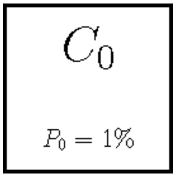


Figure 5.3 A single component system, where P_0 is the failure rate of component C_0.

Two components

Now consider a two-component system. One component depends on the other, and both must function correctly for the system to succeed. Let's set the failure threshold at 1%. Perhaps this is the proportion of failed purchases. Perhaps we're counting many different kinds of error, and purchases are one type: it isn't relevant to the model. Let's also make the assumption that both components fail independently of each other[1]. One failing doesn't make the other more likely to fail. Both components have their own failure rate. Below is a two-component system, and a given function can only succeed if **both** components succeed. Both are needed.

[1] This assumption is important to internalize. Components are like dice. They don't affect each other, and they have no memory. If one component fails, it doesn't make another component more likely to fail. It may **cause** the other component to fail, but this is different, because the failure has an external cause. We are concerned with internal failure, **independent** of other components.

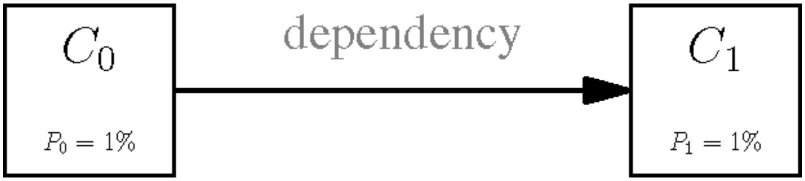


Figure 5.4 A two component system, where P_i is the failure rate of component C_i

As the components fail independently, the rules of probability tell us that we can multiply the probabilities. Four cases are possible: both fail, both succeed, the first fails and the second succeeds, the first succeeds and the second fails. We want to know the failure rate of the system. This is the same as asking for the probability that a given transaction will fail. In the four cases, three are failing, and only one succeeds. This makes our calculation easier: multiply the success probabilities together to get the probability for the case where the entire system succeeds. The failure probability is found by subtracting the success probability from 1[1]. Keeping the numbers simple, assume that each component has the same failure probability of 1%. This gives a failure probability of:

$$\begin{aligned} 1 - (99\% \times 99\%) &= 1 - 98.01\% \quad (1) \\ &= 1.99\% \quad (2) \end{aligned}$$

Despite the fact that both components are 99% reliable, the system as a whole is only 98% reliable, and fails to meet the success threshold of 99%. You can begin to see that meeting an overall level of system reliability, where that system is composed of components, all essential to operation, is harder than it looks. Each component needs to be a lot more reliable than the system as a whole.

We can extend this model to any number of components, if the components depend on each other in a serial chain. This is a simplification from the real software architectures we know and love, but let's work with this model to build some understanding of failure probabilities. Using our assumption of failure independence, where we can multiply the probabilities together, we get the following formula for the overall probability of failure of a system with an arbitrary number of components in series:

$$P_F = 1 - \prod_{i=1}^{n} (1 - P_i)$$

[1] The system can only be in two states, success or failure. The probabilities of both must sum to one. This means you can find one if you can find the other, and you get to choose the one with the easier formula.

Where P_F is the probability of system failure, n is the number of components, and P_i is the probability that component i fails.

If we chart this formula against the number of components in the system, you can see how the probability of failure grows quickly with the number of components. Even though each component is quite reliable at 99% (we give each component that same reliability to keep things simple), the system is unreliable. For example, reading from the chart, a 10-component system has under a 10% failure rate. It's a long way from the desired 1%.

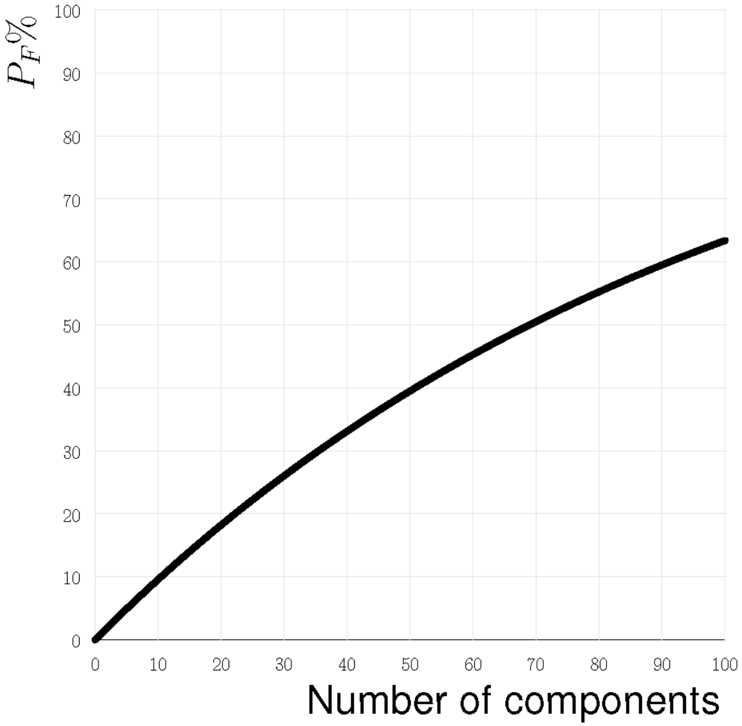


Figure 5.5 Probability of system failure against number of components where all components are 99% reliable.

The model demonstrates that our intuitions about reliability can often be quite incorrect. A convoy of ships is as slow as its slowest ship, but a software architecture isn't as unreliable as its most unreliable component. It's **much more unreliable**, because the other components can fail too.

The system in the Three Mile Island reactor wasn't linear. It was a complicated set of components, with many inter-dependencies. Real software is much more like Three Mile Island, and software components tend to be even more tightly coupled, with no tolerance for errors. Let's extend our model to see how this affects reliability. Consider a system with four components, one of which is a sub-component not on the main line. Three have a serial dependency, but the middle component depends on the fourth. Here's the configuration:

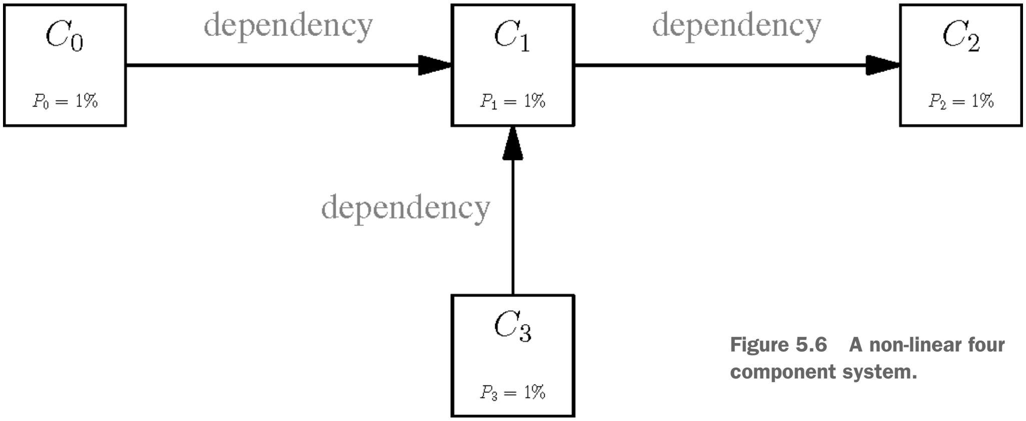


Figure 5.6 A non-linear four component system.

Again, we give all four components that same reliability of 99%. How reliable is the system as a whole? We solve the serial case with the formula introduced above. The reliability of the middle component must take into account its dependency on the fourth component. This is a serial system as well, contained inside the main system. It's a two-component system, and we've seen that this has a reliability of 100%—1.99% = 98.01%. The failure probability of the system is:

$$\begin{aligned} 1 - (99\% \times 98.01\% \times 99\%) &= 1 - 96.06\% && (1) \\ &= 3.94\% && (2) \end{aligned}$$

What about an arbitrary system with many dependencies? Or systems where multiple components depend on the same sub-component? We can make another simplifying assumption to handle this case. We assume that all components are necessary, and there are no redundancies. Every single component must work. This seems unfair, but think of how the Three Mile Island accident unfolded. Supposedly redundant systems, such as the emergency feed pumps, turned out to be crucial as stand-alone components. Yes, the reactor could work without them, but it was literally an accident waiting to happen.

If all components are necessary, then the dependency graph can be **ignored**. Every component is effectively on the main line. It's easy to overlook sub-components, or assume they don't affect reliability as much, but this is a mistake. Interconnected systems are much more vulnerable to failure than you think, because there are a lot more sub-component relationships than you think. The humans that run and build the system are one such sub-component relationship. After all, you can only blame "human error" for failures if you consider humans to be part of the system. Ignoring the dependency graph only gives you a first-order approximation of the failure rate, using the formula above, but given how quickly independent probabilities compound, that estimate is more than sufficient.

5.2.3 Redundancy doesn't do what you think it does

You can make your systems more reliable by adding redundancy. Instead of one instance of a component that might fail, have many. Keeping to our simple model, where failures are independent, this makes the system much more reliable. To calculate the failure probability of a set of redundant components, you multiply the individual failure probabilities, as all must fail for the entire assemblage to fail[1]. Now you find that probability theory is your friend. In the one-component system, adding a second redundant component gives you a failure rate of

$$1\% \times 1\% = 0.01\%$$

Figure 5.7 (1% x 1% = 0.01%)

It seems that all you need to do is add lots of redundancy, and all your problems go away. Unfortunately this is where our simple model breaks down. Few failure modes in a software system exist where failure of one instance of a component is independent of other components of the same kind. Yes, individual host machines can fail[2] , but most failures affect all software components equally. The data center is down. The network is down. The same bug applies to all instances. High load causes instances to fall like dominoes, or to flap[3]. A deployment of a new version fails on production traffic.

Simple models are also useful when they break. They can reveal hidden assumptions. Load balancing over multiple instances doesn't give you strong redundancy, it merely gives you capacity. It barely moves the reliability needle, because multiple instances of the same component ***aren't*** independent[4] .

Automatic safety devices are unreliable

Another way to reduce the risk of component failure is to use ASDs. But as we saw in the story of Three Mile Island, these bring their own risks. In the model, they are nothing more than additional components that can themselves fail.

Many years ago, I worked on a content driven website. The site added about 30 or 40 news stories a day. It wasn't a breaking news site, and a small delay in publishing

[1] The failure probability formula in this case is $P_F = \prod_{i=1}^{n} P_i$

[2] Physical power supplies fail all the time, as do hard drives; network engineers will keep stepping on cables from now until the heat death of the universe; and we'll never solve the Halting Problem (it's mathematically impossible to prove that any given program will halt instead of executing forever—you can thank Mr. Alan Turing for that) and there'll always be input that triggers infinite loops.

[3] Flapping occurs when services keep getting killed and restarted by the monitoring system. Under high load, newly started services are still "cold" (they have empty caches), and their tardiness in responding to requests is interpreted as failure, and they are killed. And then more services are started. And eventually there are no services that aren't either starting or stopping, and work comes to a halt.

[4] This statement, that multiple instances of the same software component don't fail independently, is proposed as an empirical fact from the observed behavior of real systems, and isn't proposed as a mathematical fact.

a story was acceptable. This gave me the brilliant idea to build a 60 second cache. Most pages could be generated once, and cached for 60 seconds. Once expired, any news updates would appear on the regenerated pages, and then the next 60 second caching period would begin.

This seemed like a cheap way to build what was effectively an ASD for high load. The site could handle things like election day results without needing to increase server capacity much.

The 60 second cache was implemented as an in-memory cache on each web server. Nothing fancy. It was load tested and everything appeared to be fine. But in production, servers kept crashing. There was a memory leak, and it didn't manifest unless you left the servers running for at least a day, storing over 1440 copies of each page, for each article, in memory. The first week we went live was a complete nightmare. We babysat dying machines on a 24/7 rotation.

5.2.4 Change is scary

Let's not throw the model out yet. Software systems aren't static, and suffer from catastrophic events known as "deployments". In a deployment, many components are changed at the same time. In many systems this can't be done without downtime. Let's model this as a simultaneous change of a random subset of components. What does this do to the reliability of the system?

By definition, the reliability of a component is the measured rate of failure in production. A given component only drops 1 work item in a 100, and has 99% reliability. Once a deployment is completed and live for a while, we can measure production to get the reliability rate. But this isn't much help in advance. We want to know the probability of failure of the new system **before** the changes are made.

Our model isn't strong enough to provide a formula to answer this question. But we can use another technique: Monte Carlo simulation. We run lots of simulations of the deployment, and add up the numbers to see what happens. Let's use a concrete example. Assume we've a four-component system, and the new deployment consists of updates to all four components. In a static state, before deployment, the reliability of the system's given by our standard formula:

$$0.99^4 = .9605 = 96.1\%$$

Figure 5.8 (0.99^4 = .9605 = 96.1%)

To calculate the reliability after deployment, we need to estimate the *actual* reliabilities of each component. Because we don't know what they are, we must **guess** them. Then we run the formula using the guesses.

If we do this several times, we'll be able to plot the distribution of the system reliability. We'll be able to say things like: in 95% of simulations, the system has at least

99% reliability—deploy! Or, in only 1% of simulations the system has at least 99% reliability—unplanned downtime ahead! Bear in mind that these numbers are for discussion. You'll need to decide your own numbers that reflect the risk tolerance of your own organization.

How do you guess the reliability of a component? You need to do this in a way that makes the simulation useful. Reliability isn't normally distributed, like a person's height[1]. Reliability is skewed because components are mostly reliable—most components come in around 99%, and can't go much higher. A lot of space exists below 99% to fail in. Your team is doing unit testing, and staging, and code reviews, and all that good stuff. The QA department signs-off on releases, and the head of QA is strict. The probability is high that your components are reliable, but you can't test for everything, and production is a much crueler environment than a developer's laptop, or a staging system.

You can use a skewed probability distribution[2] to model "mostly reliable". Here's a chart showing how the failure probabilities are distributed. To make a guess, pick a random number between 0 and 1, and plot its corresponding probability. You can see that most guesses will give a low failure probability.

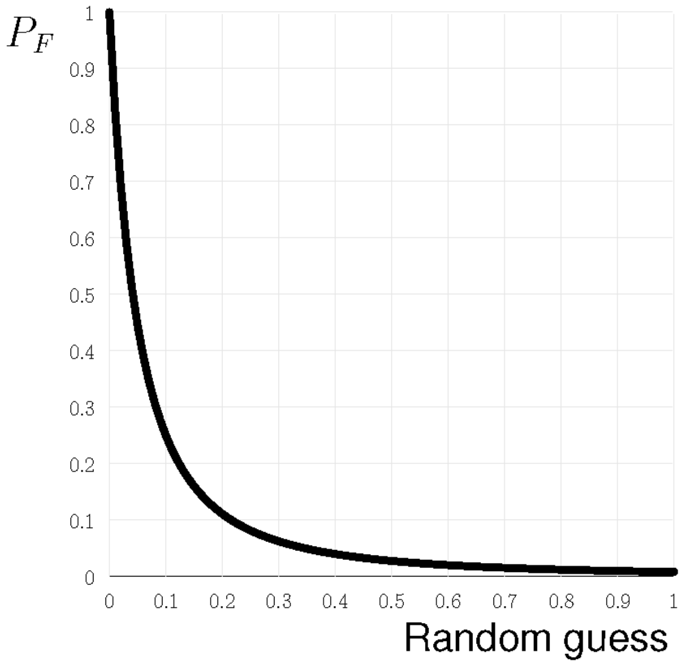


Figure 5.9 A skewed estimator of failure probability.

[1] The normal distribution assumes that any given instance will be close to the average, and has as much chance of being above average as below.

[2] The Pareto distribution is used in this example, as it's a good model for estimating failure events.

For each of the four components, you get a reliability estimate. Multiply these together in the usual manner. Now do this many times. Over many simulation runs, you can chart the reliability of the system. Here's the output from a sample exercise[1]. Although the system is often fairly reliable, it has mostly poor reliability compared to a static system. In only 0.15% of simulations the system have reliability of 95% or more.

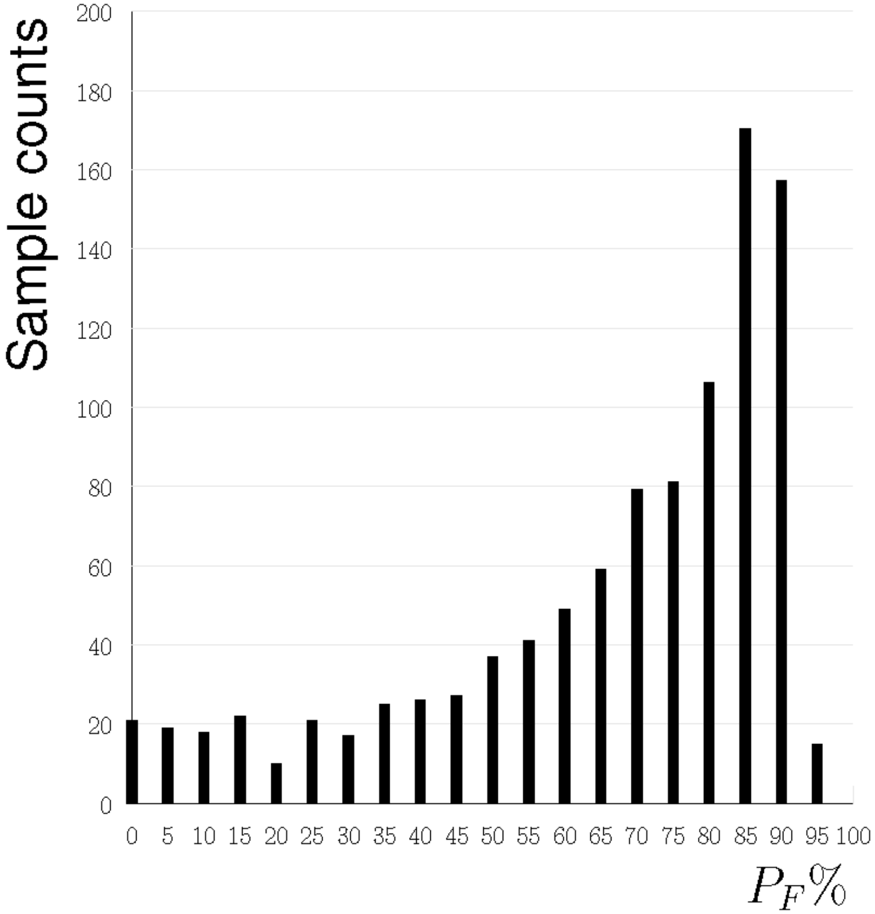


Figure 5.10 Estimated reliability of the system when four components change simultaneously.

The model shows us that simultaneous deployment of multiple components is inherently risky. It almost always fails the first time. This is why we've "scheduled" downtime, or end up frantically working at the weekend to complete a deployment. What's happening is multiple repeated deployment attempts, trying to resolve production issues that are almost impossible to predict.

The numbers don't work in our favor. We're playing a dangerous game. Our releases might be low frequency, but they've high risk[2] . And it seems that microser-

[1] In the sample exercise, 1000 runs were executed, and then categorized into 5% intervals

[2] The story of the deployment failure suffered by Knight Capital in Chapter 1 is a perfect example of this danger.

vices must introduce even more risk, as we've many more components. Yet, as we shall discover in this chapter, microservices also provide the flexibility for a solution. If we're prepared to accept high frequency releases of single components, then we'll get much lower risk exposure.

I've labored the mathematics to make a point. No software development methodology can defy the laws of probability at reasonable cost. Engineering, not politics, is the key to risk management.

5.3 The center cannot hold

The collective delusion of enterprise software development is that perfect software can be delivered complete and on-time, and deployed to production without errors, through force of management. Any defects are a failure of professionalism on the part of the team. Everybody buys into this. Why?

This book **doesn't** take the trite and lazy position that it's managements fault. Certainly, no punches are pulled calling out bad behavior, but we must be careful to see organizational behavior for what it is: rational.

We can analyze corporate politics using Game theory[1] . Why doesn't anybody point out the absurdities of enterprise software development, even when there are mountains of evidence? How many more books must be written on the subject? Thankfully we live in an age where the scale of the software systems we must build is slowly forcing enterprise software development to face reality.

Traditional software development processes are an unwanted Nash equilibrium in the game of corporate politics. It's a kind of prisoner's dilemma[2]. If all stakeholders acknowledged that failure rates must exist, and used that as a starting point, then continuous delivery is a natural solution, but nobody is willing to do this. It'd be a career limiting move. Failure isn't an option! We're stuck with a collective delusion, because we can't communicate honestly. This book aims to give you some solid principles to start that honest communication.

> **WARNING** It isn't advisable to push for change unless there's a forcing function. Wait until failure is inevitable under the old system, and then be the white knight. Pushing for change when you've no leverage, is indeed a career limiting move.

[1] The part of mathematics that deals with multi-player games and the limitations of strategies to maximize results.

[2] A Nash equilibrium is a state of a game where no player can improve their position by changing strategy unilaterally. The prisoner's dilemma is a compact example: Two guilty criminals, who robbed a bank together, are captured by the police, and placed in separate cells where they can't communicate. If they both stay silent, then they walk free. But if one betrays the other (getting a shorter sentence as a reward) then the betrayed criminal serves a life sentence on the evidence of the other. The only rational strategy is to betray, and take the shorter sentence, because your partner might betray you. If only they could communicate!

5.3.1 The cost of perfect software

The software that controlled the space shuttle was some of the most perfect software ever written. It's a good example of how expensive such software truly is, and calls out the absurdity of the expectations for enterprise software. It's also a good example of how much effort is required to build redundant software components.

The initial cost estimate for the shuttle software system was $20m. The final bill was $200m. This is the first clue that defect-free software is an order-of-magnitude more expensive than even software engineers estimate. The full requirements specification has 40 000 pages for a mere 420 000 lines of code. By comparison, Google's Chrome web browser is over 5 million lines of code. How perfect is the shuttle software? On average, there was one bug per release. It wasn't completely perfect either!

The software development process was incredibly strict. It was a traditional process with highly detailed specifications. Strict testing, verification and code reviews were enforced. Bureaucratic signatures were needed for release. Many stakeholders in the enterprise software development process truly believe that this level of delivery is what they're going to get.

It's the business of business to make return-on-investment decisions. You spend money to make money, but you must have a business case. This breaks down if you don't understand your cost model. It's the job of the software architect to make these costs clear, and to provide alternatives, where the cost of software development is matched to the expected returns of the project.

5.4 Anarchy works

The most important question in software development is: "What is the acceptable error rate?" This is the first question to ask at the start of a project. It drives all the other questions and decisions. It also makes clear to all stakeholders that the process of software development is about controlling, not conquering, failure.

The primary consequence is that large scale releases can never meet the acceptable error rate. Reliability is compromised by the uncertainty of a large release, that large releases must be rejected as an engineering approach. This is mathematics, and no amount of QA can overcome it.

Small releases are less risky. The smaller the better. Small releases have small uncertainties, and we can keep under the failure threshold. Small releases also mean frequent releases. Enterprise software must constantly change to meet market forces. These small releases must go all the way to production to fully reduce risk. Collecting them into large releases takes you back to square one. This is how the probabilities work.

A system under constant failure isn't fragile. Every component expects others to fail, and is built to be more tolerant of failure. The constant failure of components exercises redundant systems and backups, ensures that you know they work. You've an accurate measure of the failure rate of the system. It's a known quantity that can be controlled. The rate of deployment can be adjusted as risks grow and shrink.

How does our simple risk model work under these conditions? You may only be changing one component at a time, but aren't you still subject to large amounts of risk? You know that your software development process isn't going to deliver updated components that are as stable as those that have been baked into production for a while.

Let's say updated components are 80% reliable on first deployment. You're not going to meet a reliability threshold of 99% in any of the systems we've looked at. Redeploying a single component still isn't a small enough deployment. This is an engineering and process problem that we'll address in the remainder of this chapter—how to make changes to a production software system whilst maintaining a desired risk tolerance.

5.5 Microservices and Redundancy

An individual component of a software system should never be run as a single instance. A single instance is vulnerable to failure. The component itself could crash. The machine that it's running on could fail. The network connection to that machine could be accidentally misconfigured. No component should be a single point of failure.

To avoid being a single point of failure, you can run multiple instances of the component. Now you can handle load and you're more protected against some kinds of failure. You aren't protected against software defects in the component itself, which affect all instances. Even then, such defects can be usually mitigated by automatic restarts[1]. Once a component has been running in production for a while, you've enough data to get a good measure of its reliability.

How do you deploy a new version of a component? In the traditional model, you try, as quickly as possible, to replace all the old instances with a full set of new ones. The blue-green deployment strategy, as it's known, is an example of this. You've a running version of the system, call this the *blue* version. You spin up a new version of the system, call this the *green* version. Then you choose a specific moment to redirect all traffic from blue to green. Now, if something goes wrong, you can quickly switch back to blue, and assess the damage. At least you're still up.

One way to make this less risky is to redirect only a small fraction of traffic to green at first. If you're satisfied that everything still works, redirect greater and greater volumes of traffic until green has completely taken over.

The microservice architecture makes it easy to adopt this strategy, and reduce risk even further. Instead of spinning up a full quota of new instances of the green version of the service, spin up one instance. This one new instance gets a small portion of all production traffic, and the existing blues look after the main bulk of traffic. You can observe the behavior of the single green instance. If it's badly behaved, you can decommission it. Although a small amount of traffic has been affected, and although there's been small increase in failure, you're still in control. You can fully control the level of exposure by controlling the amount of traffic that you send to that single new instance.

[1] Restarts don't protect you against nastier kinds of defect, such as poison messages.

Microservice deployments are nothing more than the introduction of a single new instance. If the deployment fails, rollback is the decommissioning of a single instance. Microservices give you well-defined primitive operations on your production system: add/remove a service instance. Nothing more's required. These primitive operations can be used to construct any deployment strategy you desire. For example, blue-green deployments break down into a list of add and remove operations on specific instances.

The definition of a primitive operation is a powerful mechanism for achieving control. If everything is defined in terms of primitives, and you can control the composition of the primitives, then you can control the system. The microservice instance is our primitive, and the unit with which we build our systems. Let's examine the journey of that unit from development to production.

5.6 Continuous Delivery

The ability to safely deploy a component to production at any time is powerful because it lets you control risk. Continuous delivery in a microservices context means the ability to create a specific version of a microservice, and to run one or more instances of that version in production, on demand. The essential elements of a continuous delivery pipeline are:

- A version-controlled local development environment for each service, supported by unit testing, and the ability to test the service against an appropriate local subset of the other services, using mocking if necessary.
- A staging environment to both validate the microservice, and build, reproducibly, an artifact for deployment. Validation is automated, but scope is allowed for manual verification if necessary.
- A management system, used by the development team to execute combinations of primitives against staging and production, implementing the desired deployment patterns in an automated manner.
- A production environment that's constructed from deployment artifacts as much as possible, with an audit history of the primitive operations applied. The environment is self-correcting and able to take remedial action, such as restarting crashed services. The environment also provides intelligent load balancing, allowing traffic volumes to vary between services.
- A monitoring and diagnostics system that verifies the health of the production system after the application of each primitive operation, and allows the development team to introspect and trace message behavior. Alerts are generated from this part of the system.

The pipeline assumes that the generation of defective artifacts is a common occurrence. The pipeline attempts to filter them out at each stage. This is done on a per-artifact basis, rather than trying to verify an entire update to the system. As a result the verification is both more accurate, and more credible, as confounding factors have been removed.

Even when a defective artifact makes it to production, this is still considered a normal event. The behavior of the artifact is continuously verified in production after deployment, and the artifact is removed if not acceptable. Risk is controlled by progressively increasing the proportion of activity that the new artifact handles.

Continuous delivery is based on the reality of software construction and management. It delivers:

- Lower risk of failure by favoring low-impact high-frequency single instance deployments over high-impact low-frequency multiple instance deployments.
- Faster development by enabling high frequency updates to the business logic of the system, giving a faster feedback loop, and faster refinement against business goals.
- Lower cost of development as the fast feedback loop reduces time wasted on features that have no business value.
- Higher quality as less code is written overall, and that which is written is immediately verified.

The tooling to support continuous delivery, and the microservice architecture, is still in the early stages of development[1]. Though an end-to-end continuous delivery pipeline system is necessary to fully gain the benefits of the microservice architecture, it's possible to live with pipeline elements that are less than perfect.

At the time of writing, all teams working with this approach are using multiple tools to implement different aspects of the pipeline, as comprehensive solutions don't exist. The microservice architecture requires more than current Platform-as-a-Service vendors offer. Even when comprehensive solutions emerge, they'll still present tradeoffs in implementation focus[2] . It's probable that you'll continue to need to put together a context-specific tool-set for each microservice system that you build. As we work through the rest of this chapter, the desirable properties for these tools will be the thing to focus on. You will almost certainly need to invest in the development of some of your own tooling. At the least this will be integration scripts for the third-party tools you've selected.

5.6.1 Pipeline

The purpose of the continuous delivery pipeline is to provide feedback to the development team as quickly as possible. In the case of failure, that feedback should indicate the nature of the failure. It must be easy to see the failing tests, the failing performance results, or the failed integrations. You should be able to see a history of the verifications and failures of each microservice. This isn't the time to roll your own tooling—there are many capable continuous integration tools[3]. The key requirement for you is that your chosen tool can handle many projects easily, as each microservice is built separately.

[1] Things are improving all the time. Take a look at http://deis.com, for example.

[2] The Netflix suite, http://netflix.github.io, is a good example of a comprehensive, but opinionated, tool chain.

[3] Two quick mentions: if you want to run something yourself, try http://hudson-ci.org; if you want to outsource, try http://travis-ci.org(used by the example in Chapter 9).

The continuous integration tool is one stage of the pipeline, usually operating before a deployment to the staging systems. You need to be able to trace the generation of microservices throughout the pipeline. The continuous integration server generates an artifact that'll be deployed into production. Before that happens, the source code for the artifact needs to be marked and tagged to create **hermetic** artifact generation—you must be able to reproduce any build from the history of your microservice. After artifact generation, you need to be able to trace the deployment of that artifact over your systems from staging to production. This tracing mustn't be only at the system level, but also within the system, tracing the number of instances run, and when. Until third party tooling solves this problem, you'll need to build this part of the pipeline diagnostics yourself. It‘s an essential and worthwhile investment for investigating failures.

The unit of deployment is a microservice, and the unit of movement through the pipeline is a microservice. The pipeline should prioritize the focus on the generation and validation of artifacts that represent a microservice. A given version of a microservice is instantiated as a fixed artifact that never changes. Artifacts are **immutable**. The same version of a microservice always generates the same artifact, at a binary encoding level. It's natural to store these artifacts for quick access[1]. Nonetheless you need to retain the capability to hermetically rebuild any version of a microservice, as the build process is an important element of defect investigation.

The development environment also needs to make the focus on individual microservices fluid and natural. This affects the structure of your source code repositories. We'll look at this more deeply in Chapter 7. Local validation is also important, as the first measure of risk. Once the developer's satisfied that a viable version of the microservice is ready, it's the developer that initiates the pipeline to production.

The staging environment reproduces the development environment validation in a controlled environment, keeping it free from the variances in local developer machines. Staging also performs scaling and performance tests, and can use multiple machines to simulate production, to a limited extent. Staging's core responsibility is to generate an artifact with an estimated failure risk within a defined tolerance.

Production is the live, revenue generating part of the pipeline. Production is updated by accepting an artifact, and a deployment plan, and applying the deployment plan under measurement of risk. To manage risk, the deployment plan is a progressive execution of deployment primitives—activating and deactivating microservice instances. Tooling for production microservices is the most mature at present, as it is the most critical part of the pipeline. A great variety of orchestration and monitoring tools are available to help[2].

5.6.2 Process

It's important to distinguish continuous delivery from continuous deployment. Continuous deployment is a form of continuous delivery, where commits, though they

[1] Amazon S3 isn't a bad place. More focused solutions are on the market, such as JFrog's Artifactory product.

[2] Common choices here are Kubernetes, Mesos, Docker, and so forth. Although these tools fall into a broad category, they operate a different levels of the stack, and aren't mutually exclusive. The case study in Chapter 9 will use on Docker and Kubernetes.

may be automatically verified, are still pushed directly and immediately to production. Continuous delivery operates at a coarser grain, where sets of commits are packaged into immutable artifacts. In both cases, deployments can be effectively "real-time" and occur multiple times per day.

Continuous delivery is more suited to the wider context of enterprise software development, as it allows the team to accommodate compliance and process requirements which are difficult to change within the lifetime of the project. Continuous delivery is also more suited to the microservice architecture, as it allows the focus to be on the microservice rather than code.

If "continuous delivery" is considered a continuous delivery of microservice instances, this understanding drives other virtues. Microservices should be kept small, and that verification, particularly human verification such as code reviews, is possible with the desired time-frames of multiple deployments per day.

5.6.3 Protection

The pipeline protects you from exceeding failure thresholds by providing measures of risk at each stage to production. It isn't necessary to extract a failure probability prediction from each measure[1]. You"ll know the feel of the measures for your system, and you can use a scoring approach effectively.

In development, your key risk measuring tools are code reviews and unit tests. Using modern version control for branch management[2] means you can adopt a development workflow where new code is written on a branch, and then merged into the mainline. The merge is only performed if the code passes a review. The review can be performed by a peer, rather than a senior. Peer developers on a project have more information and are better able to asses the correctness of a merge. This workflow means that code review is a normal part of the development process, and has low friction. Microservices keep the friction even lower because the units of review are smaller and have less code.

Unit tests are critical to risk measurement. You should take the perspective that unit tests must pass before branches can be merged, or code is committed on the mainline. This keeps the mainline potentially deployable, as a build on staging has a good chance of passing. Unit tests in the microservice world are concerned with demonstrating the correctness of the code. The other benefits of unit testing, such as making refactoring safer, are less relevant.

Unit tests aren't sufficient for accurate risk measurement, and are subject to diminishing marginal returns. Moving from 50% test coverage to 100% reduces your deployment risk much less than moving from 0% to 50%. Don't get suckered into the fashion for 100% test coverage. It's a fine badge of honor (literally!) for Open Source utility components, but is superstitious theater for business logic.

[1] You could use statistical techniques such as Bayesian estimation to do this if desired.

[2] Using a distributed version control system such as git is essential. You need to be able to use pull requests to implement code reviews.

On the staging system, you can measure the behavior of a microservice in terms of its adherence to the message flows of the system. Ensuring that the correct messages are sent by the service, and the correct responses given, is also a binary pass/fail test, which you can score with a 0 or 1 The service must meet expectations fully. Though these message interactions are tested via unit tests in development, they also need to be tested on the staging system, as this is a closer simulation of production.

Integrations with other parts of the system can also be tested as part of the staging process. Those parts of the system that aren't microservices, such as stand-alone databases, network services such as mail servers, external web service end-points, and others, are simulated or run in small scale. The microservice's behavior with respect to these can then be measured. Other aspects of the service, such as performance, resource consumption, and security, need to be measured in a statistical way, taking samples of behavior, and using these to predict risk of failure.

Finally, even in production, the risk of failure continues to be measured. Even before going into production, you can establish manual gates—formal code reviews, penetration testing, user acceptance, and so forth. These may be legally unavoidable, but they can still be integrated into the continuous delivery mind-set.

Running services can be monitored and sampled. Key metrics, particularly those relating to message flow rates, can be used to determine service and system health. A great deal more will be said about this aspect of the microservice architecture in Chapter 6.

5.7 Running a microservice system

The tooling to support microservices is developing quickly, and new tools are emerging at a high rate. It isn't useful to examine in detail that which will be out-of-date soon. This chapter focuses on general principles, allowing you to compare and assess tools and select those most suitable for your context. You should expect and prepare to build some tooling yourself. This isn't a book on deployment in general, and it doesn't discuss best practices for the deployment of system elements, such as database clusters, that aren't microservices. It is still recommended that these be subject to automation, and if possible controlled by the same tooling. The focus of this chapter is on deployment of your own microservices, encoding the business logic of the system, and making them subject to a higher degree of change compared to other elements.

5.7.1 Immutability

It's a core principle that, of the approach described here, microservice artifacts are immutable. This preserves their ability to act as primitive operations. A microservice artifact could be a container, a virtual machine image, or some other abstraction[1]. The essential characteristic of the artifact is that it can't be changed internally, and only has two states, active, and inactive.

[1] For huge systems, you might even consider an AWS auto-scaling group to be your base unit.

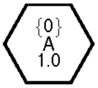
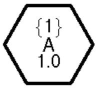
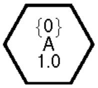
The power of immutability is that it excludes side-effects from your system. The behavior of the system, and microservice instances, is much more predictable because you can be sure that they aren't affected by changes that you are unaware of. An immutable artifact contains everything the microservice needs to run, at fixed versions. You can be certain that your language platform version, and your libraries, and other dependencies, are exactly as you expect. Nobody can manually login to the instance, and make un-audited changes. This predictability allows you to calibrate your risk estimations more accurately.

Running immutable instances also forces you to treat your microservices as disposable. An instance that's developed a problem, or that contains a bug, cannot be "fixed". It can only be deactivated, and replaced by the activation of a new instance. Matching capacity to load isn't about building new installations on bigger machines, it's about running more instances of the artifact. No individual instance is in any way special. This approach is a basic building block for building reliable systems on unreliable infrastructure

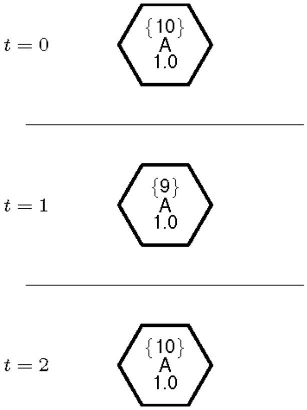
MICROSERVICE DEPLOYMENT PATTERNS

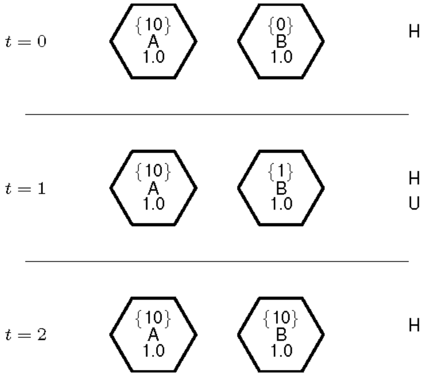
This section is a reference overview of microservice deployment patterns. You'll need to compare these patterns against the capabilities of the automation tooling that you use. Unfortunately you should expect to be disappointed, and you'll need to augment your tooling to fully achieve the desired benefits of the patterns.

Feel free to treat this section more as a recipe book for cooking up your own patterns, rather than a prescription. You can skim the deployment patterns without guilt[1].

Name	Rollback
Motive	Recover from a deployment that's caused one or more failure metrics to exceed their thresholds. This enables you to deploy new service versions where the estimated probability of failure is higher than your thresholds, whilst maintaining overall operation within thresholds.
Description	Apply the activate primitive for a given microservice artifact to a system. Observe failure alerts. Deactivate the same artifact. Deactivation can be manual or automatic. Logs should be preserved. Deactivation should return the system to health. Recovery is expected but may not occur in cases where the offending instance has injected defective messages into the system (for this, see the Kill Switch pattern).
Sequence	$t = 0$  There are 0 active instances of A. A is at version 1.0. $t = 1$  Activate 1 instance of A. Failures exceed thresholds. $t = 2$  Deactivate 1 instance of A.

[1] In production, I'm most fond of the Progressive Canary.

Name	Homeostasis
Motive	Maintain desired architectural structure and capacity levels.
Description	A declarative definition of your architecture, including rules for increasing capacity under load's implemented by application of activation and deactivation primitives to the system. Simultaneous application of primitives is permitted, although care must be taken to implement and record this correctly. Homeostasis can also be implemented by allowing services to issue primitive operations, and defining local rules for doing this, see the patterns Mitosis and Apoptosis.
Sequence	 $t = 0$ {10} A 1.0 $t = 1$ {9} A 1.0 $t = 2$ {10} A 1.0 There should always be 10 active instances of / An instance of A fails. Monitoring detects failure Automatically activate 1 instance of A.

Name	History
Motive	Provide diagnostic data to aid understanding of failing and healthy system behavior.
Description	Maintain an audit trail of the temporal order of primitive operation application. A complication's that you may allow simultaneous application of sets of primitives. The audit history allows you to diagnose problems by inspecting the behavior of previous versions of the system—these can be resurrected by applying the primitives to simulated systems. Defects introduced but undetected initially can also dealt with by moving backwards over the history.
Sequence	 $t = 0$ {10} A 1.0 {0} B 1.0 $t = 1$ {10} A 1.0 {1} B 1.0 $t = 2$ {10} A 1.0 {10} B 1.0 History = [] History = [(1,start,B/1.0,1,'alice')] User alice activated 1 instance of B/1.0 at $t = 1$ History = [(1,start,B/1.0,1,'alice'), (2,start,B/1.0,9,'bob')].

5.7.2 Automation

Microservice systems in production have too many moving parts to be managed manually. This is part of the trade-off of the architecture. You must commit to using tooling to automate your system. This is a never-ending task. Automation doesn't cover all activities from day one, nor should it, as you need to allocate most of your development effort to creating business value. Over time, you'll need to automate more and more.

To determine which activity to automate next, divide your operational tasks into two categories. In the first category, Toil[1], place those tasks where effort grows at least linearly with the size of the system. To put it another way, from a computational complexity perspective, these are tasks where human effort is at least **O(n)**, where **n** is the number of microservice types (not instances). For example, configuring log capture for a new microservice type might require manual configuration of the log collection subsystem.

In second category, Win, place tasks that are less than **O(n)** in the number of microservice types. For example, adding a new database secondary reader instance to handle projected increased data volumes.

The next task to automate is the most annoying task from the Toil pile. Define annoying as "most negatively impacting the business goals". Don't forget to include failure risk in your calculation of negative impact.

Automation is also necessary to execute the microservice deployment patterns. Most of the patterns require the application of large numbers of primitive operations, over a scheduled period, and under observation for failure. These are tasks that can't be performed manually at scale.

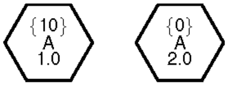
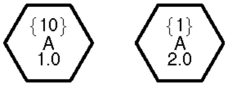
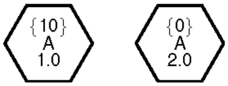
Automation tooling is relatively mature, and dovetails with the requirements of modern large scale enterprise applications. A wide range of choice exist, and your decision should be driven by your comfort levels with custom modification or scripting, as you'll need this to fully execute the microservice deployment patterns[2].

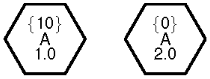
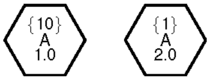
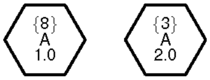
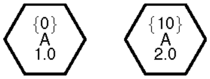
MORE MICROSERVICE DEPLOYMENT PATTERNS

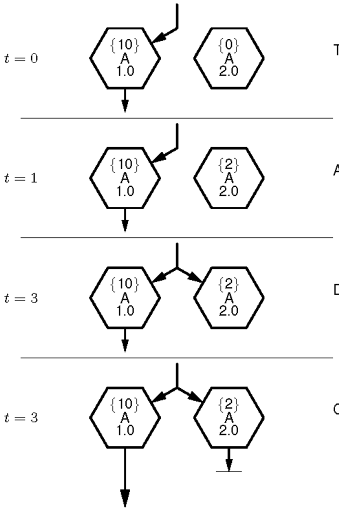
Name	Canary
Description	New microservices, and new versions of existing microservices, introduce considerable risk to a production system. It's unwise to deploy the new instances and immediately allow them to take large amounts of load. Instead, run multiple instances of known-good services, and slowly replace these with new ones. The first step's to validate that the new microservice both functions correctly, and isn't destructive. Activate a single new instance, and direct a small amount of message traffic to this instance. Then watch your metrics to make sure the system behaves as expected. Apply the Rollback pattern if not.

[1] This usage of the term originates with the Google Site Reliability Engineering team.

[2] Look to evaluate tools such as Puppet, Chef, Ansible, Terraform, and AWS CodeDeploy

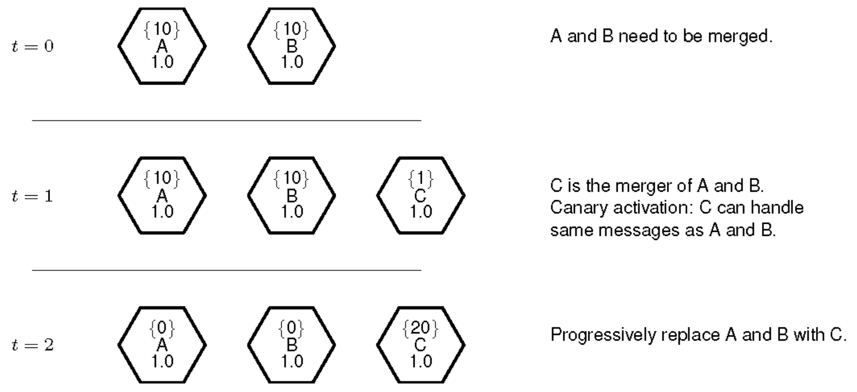
Sequence	$t = 0$  H = [] $t = 1$  H = [(1,start,A/2.0,1)] Activate an instance of A version 2.0. It fails. $t = 2$  H = [(1,start,A/2.0,1), (2,stop,A/2.0,1)]

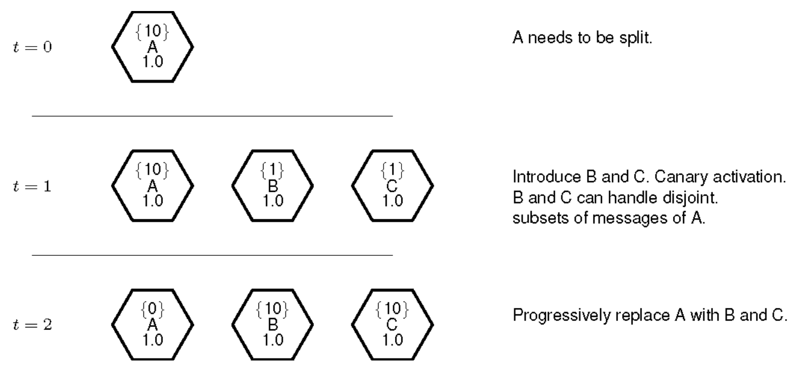
Name	Progressive Canary
Motive	Reduce the risk of a full update by applying changes progressively in larger and larger tranches.
Description	Though Canary can validate the safety of a single new instance, particularly with respect to unintended destructive behavior, it doesn't guarantee good behavior at scale. Deploy a progressively increasing number of new instances to take progressively more traffic, continuing to validate during the process. This balances the need for full deployment of new instance versions to occur at reasonable speed, with the need to manage the risk of the change. Primitives are applied concurrently in this pattern. The Rollback can be extended to handle decommission of multiple instances if a problem arises.
Sequence	$t = 0$  H = [] Target state is 10 of A/2.0 and 0 of A/1.0 $t = 1$  H = [(1,start,A/2.0,1)] Activate an instance of A version 2.0. It succeeds. $t = 2$  H = [(1,start,A/2.0,1), (2,stop,A/1.0,2), (2,start,A/2.0,2)] Stop 2 of A/1.0 and start 2 of A/2.0 Start with small sets of instances. $t = 3$  H = [(1,start,A/2.0,1), (2,stop,A/1.0,2), (2,start,A/2.0,2), (3,stop,A/1.0,8), (3,start,A/2.0,7)] End with larger sets of changes.

Name	Bake
Motive	Reduce the risk of failures that have severe downsides.
Description	This is a variation of Progressive Canary that maintains a full complement of the existing instances, but also sends a copy of inbound message traffic to the new instances. The output from the new instances is compared with the old to ensure that deviations are below a desired threshold. The output from the new is discarded until this criterion is met. The system can continue in this configuration, validating against production traffic, until sufficient time has passed to reach the desired risk level. This pattern is most useful when the output must meet a strict failure threshold, and where failure places the business at risk. Consider using when you are dealing with access to sensitive data, financial operations, and resource intensive activities that are hard to reverse[1]. This pattern requires intelligent load balancing and additional monitoring to implement.
Sequence	 $t = 0$ {10} A 1.0 {0} A 2.0 $t = 1$ {10} A 1.0 {2} A 2.0 $t = 3$ {10} A 1.0 {2} A 2.0 $t = 3$ {10} A 1.0 {2} A 2.0 Target state is 10 of A/1.0 and 2 of A/2.0 for baking. Activate 2 instances of A/2.0. They succeed. Duplicate traffic Compare output. Discard A/2.0 output.

[1] The canonical description of this technique is given by Github's Zach Holman: https://zachholman.com/talk/move-fast-break-nothing

Name	Merge
Motive	Performance is impacted by network latency.
Description	As the system grows and load increases, certain message pathways become bottlenecks. Latency caused by the need to send messages over the network between microservices may become unacceptable. Also, security concerns could arise that require encryption of the message stream, causing further latency. To counteract this, trade-off some of the flexibility of microservices for the necessity of performance by merging microservices in the critical message path. By using a message abstraction layer, and pattern matching, as discussed in earlier chapters, this can be achieved with minimal code changes. Don't merge microservices together wholesale. Look to isolate the message patterns of concern into a combined microservice. By executing a message pathway within a single process, you remove the network from the equation. This is a good example of the benefit of the microservices-first approach. In the earlier part of an applications life-cycle more flexibility is needed as understanding of the business logic is less solid. Later, optimizations may be needed to meet performance goals.

Sequence	 $t = 0$ {10} A 1.0 {10} B 1.0 $t = 1$ {10} A 1.0 {10} B 1.0 {1} C 1.0 $t = 2$ {0} A 1.0 {0} B 1.0 {20} C 1.0 A and B need to be merged. C is the merger of A and B. Canary activation: C can handle same messages as A and B. Progressively replace A and B with C.

Name	Split
Motive	Microservices grow over time as more business logic is added, necessitating new kinds of service to avoid building technical debt.
Description	In the early life-cycle of a system, microservices are small and handle general cases. As time goes by, more special cases are added to the business logic. Rather than handling these cases with more complex internal code and data structures, it's better to split out special cases into focused microservices. Pattern matching on messages makes this practical, and is one of the core benefits of the pattern matching approach. This pattern captures one the core benefits of the microservice architecture: the ability to handle frequently changing and under-specified requirements. Always be looking for opportunities to split, and avoid the temptation to use more familiar language constructs (such as object-oriented design patterns), as these build technical debt over time.
Sequence	 $t = 0$ {10} A 1.0 $t = 1$ {10} A 1.0 {1} B 1.0 {1} C 1.0 $t = 2$ {0} A 1.0 {10} B 1.0 {10} C 1.0 A needs to be split. Introduce B and C. Canary activation. B and C can handle disjoint. subsets of messages of A. Progressively replace A with B and C.

5.7.3 Resilience

Chapter 3 discussed some of common failure modes of microservice systems. A production deployment of microservices needs to be resilient to these failure modes. Although the system can never be entirely safe, mitigations should be put in place. As always, the extent and cost of the mitigation should correspond to the desired level of risk.

In monolithic systems, failure is often dealt with by restarting instances of the monolith that fail. This approach is heavy-handed, and often ineffective. The microservice architecture offers a finer-grained menu of techniques for handling failure. The abstraction of a messaging layer is helpful, as this layer can be extended to provide automatic safety devices (ASDs). Bearing in mind that ASDs aren't silver bullets, and may themselves cause failure, they're still useful for many modes of failure.

SLOW DOWNSTREAM

In this failure mode, taking the perspective of a given client microservice instance, responses to its outbound messages have latency or throughout levels that are outside of acceptable levels.

The client microservice can use the following dynamic tactics, roughly in order of increasing sophistication:

Timeouts

Consider messages failed if a response isn't delivered within a fixed timeout period. This prevents resource consumption on the client microservice.

Adaptive timeouts

Use timeouts, but don't set them as fixed configuration parameters. Instead, dynamically adjust the timeouts based on observed behavior. As a simplistic example, timeout if the response delay is more than three standard deviations from the observed mean response time. Adaptive timeouts reduce the occurrence of false positives when the overall system is slow, and avoid delays in failure detection when the system is fast.

Circuit breaker

Persistently slow downstream services should be considered effectively dead. Implementation requires the messaging layer to maintain meta data about downstream services. This tactic avoids unnecessary resource consumption, and unnecessary degradation of overall performance. It increases the risk of overloading healthy machines by redirecting too much traffic to them, causing a cascading failure similar to the unintended effects of the ASDs at Three Mile Island.

Retries

If failure to execute a task has a cost, and there's tolerance for delay, then it may make more sense to retry a failed message by sending it again. This is an ASD that has great potential to go wrong. Large volumes of retries are a self-inflicted denial-of-service attack. Use a retry budget to avoid this by only retrying a limited number of times, and if the meta data is available, only doing it for a limited number of times per downstream. Retries should also use a randomized exponential back-off delay before being sent, as this gives the downstream a better chance of recovery by spreading load over time.

Intelligent round-robin

If the messaging layer is using point-to-point transmission to send messages, then it necessarily has sufficient meta data to implement round-robin load-balancing amongst the downstreams. Simple round-robins keeps a list of downstreams and cycles through them. This ignores differences in load between messages, and can lead to individual downstreams becoming overloaded. Random round-robin is found empirically to be little better, probably because the same clustering of load is possible. If the downstream microservices provide back pressure meta data, then the round robin algorithm can make more informed choices. It can choose the least loaded downstream. It can weight downstreams based on known capacity. It can restrict downstreams to a subset of the total to avoid a domino effect from a circuit breaker that trips too aggressively.

Upstream overload

This is other end of the overload scenario. The downstream microservice is getting too many inbound messages. Some of the tactics to apply are:

Adaptive throttles

Don't attempt to complete all work as it comes in. Instead, queue the work to a maximum rate than can be safely handled. This prevents the service from **thrashing**. Services with severe resource constraints spend almost all their time swapping between tasks rather than working on the tasks. On thread-based language platforms this consumes memory. On event-driven platforms, this manifests as a single task hogging the CPU, and stalling all others. As with timeouts, it's worth making throttles adaptive to optimize resource consumption.

Back-pressure

Provide client microservices with meta data describing current load levels. This meta data can be embedded in message responses. The downstream service doesn't actively handle load, but relies on the kindness of its clients. The meta data makes client tactics for slow downstreams more effective.

Load shedding

Refuse to execute tasks once a dangerous level of load is reached. This is a deliberate decision to fail a certain percentage of messages. This tactic gives most messages reasonable latency, and some total failure, rather then allowing many messages to have high latency with sporadic failure. Appropriate meta data should be returned to client services to ensure it interprets load shedding correctly and doesn't trigger a circuit breaker. The selection of tasks to drop, or add to the queue, or execute immediately, can be determined algorithmically and is context dependent. Nonetheless, even a simple load shedder prevents many kinds of catastrophic collapse.

In addition to the dynamic tactics above, upstream overload can be reduced on a longer time-frame by applying the Merge deployment pattern.

Lost actions

Apply the Progressive Canary deployment pattern, measuring message flow rates to ensure correctness. There'll be more discussion on measurement in Chapter 6.

Poison Messages

A microservice generates a poisonous message that triggers a defect in other microservices, causing some level of failure. If the message is continuously retried against different downstream services, they'll all suffer failures.

Drop duplicates

Downstream microservices should track message correlation identifiers, and keep a short-term record of recently seen inbound messages. Duplicates should be ignored.

Validation

Trade-off the flexibility of schema-free messages for stricter validation of inbound message data. This has a less detrimental effect later in the project when requirement change has slowed.

Consider building a dead-letter service. Problematic messages are forwarded to this service for storage and later diagnosis. This allows you to monitor message health across the system.

Guaranteed delivery

Message delivery may fail in many ways. Messages may not arrive, or arrive multiple times. Dropping duplicates helps within a service. Duplicated messages sent to multiple services are more difficult to mitigate. If the risk associated with such events is too high, extra development effort should be allocated to implementing idempotent message interactions[1].

Emergent Behavior

A microservice system is a system with many moving parts. Message behavior may have unintended consequences, such as triggering additional workflows. Correlation identifiers can be used for after-the-fact diagnosis, but not to actively prevent side-effects.

Time-to-live

Use a decrementing counter reduces it each time an inbound message triggers the generation of outbound messages. This prevents unbounded side-effects from proceeding without any checks. It stops infinite message loops. It won't fully prevent all side-effects, but limits their effects. You'll need to determine the appropriate value of the counter in the context of your own system, but prefer low values. Microservice systems should be shallow, not deep.

[1] Be careful not to over-think your system in the early days of a project. It's often better to accept the risk of, and actual, data corruption to get to market sooner. Be ethical, and only make this decision openly with your stakeholders. Chapter 7 has more discussion on making these decisions.

Catastrophic collapse

Some emergent behavior can be exceptionally pathological, placing the system into an unrecoverable state, even though the original messages are no longer present. Even with the Homeostasis pattern in place, service restarts are unable to bring the system back to health.

For example, a defect may crash numerous services in rapid succession. New services are started as replacements, but have empty caches, and are unable to handle current load levels. These new services crash and are themselves replaced. The system can't establish enough capacity to return to normal. This is known as the **thundering herd** problem.

Static responses

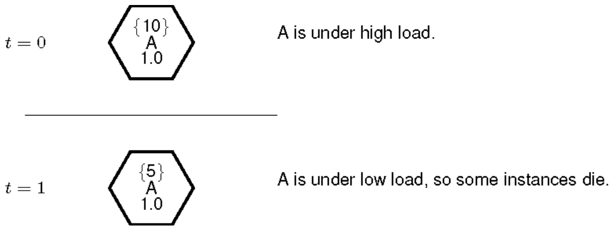
Use low resource emergency microservices that return hard-coded responses to temporarily take load.

Kill Switch

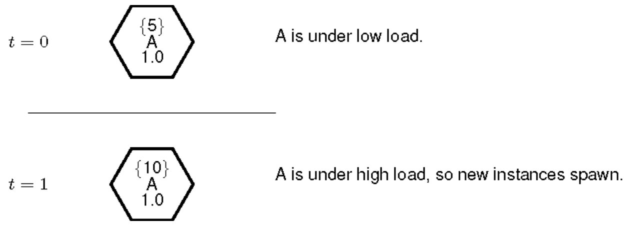
Establish a mechanism to selectively stop large subsets of services. This gives you the ability to quarantine the problem. You can then restart into a known-good state.

In addition to the dynamic tactics above, prepare for disaster by deliberately testing individual services with high load to determine their failure points. Software systems tend to fail quickly rather than gradually, and you need to establish safe limits in advance.

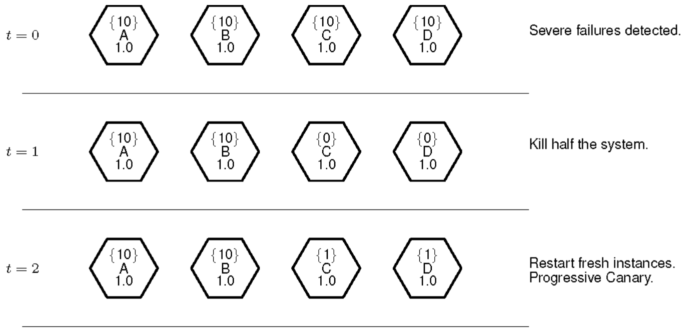
Even more microservice deployment patterns

Name	Apoptosis[1]
Motive	Remove malfunctioning services quickly. Reduce capacity organically.
Description	Microservices can perform self-diagnosis and shut themselves down if their health is unsatisfactory. For example, all message tasks may be failing because local storage is full. The service can maintain internal statistics to calculate health. This also enables a graceful shutdown by responding to messages with meta data indicating failure during the shutdown, rather than triggering timeouts. Apoptosis is useful to match capacity with load. It's costly to maintain active resources far in excess of the levels necessary to meet current load. Services can choose to self terminate, using a probabilistic algorithm to avoid mass shutdowns. Load is redistributed over the remaining services.
Sequence	 $t = 0$ {10} A 1.0 — A is under high load. $t = 1$ {5} A 1.0 — A is under low load, so some instances die.

[1] Living cells literally commit suicide if they become too damaged. This prevents cell damage from accumulating and causing cancers.

Name	Mitosis[1]
Motive	Respond to increased load organically without centralized control.
Description	Individual microservices have the most accurate measure of their own load levels. Trigger the launching of new instances if local load levels are too high. This should be done using a probabilistic approach to avoid large volumes of simultaneous launches. The newly launched service takes some the load, bringing levels back to acceptable levels. Mitosis (and Apoptosis) should be used with care, and with built-in limits. You don't want unbounded growth or a complete shutdown. Launch and shutdown should occur via primitive operations executed by the infrastructure tooling, not by the microservices themselves.
Sequence	 $t = 0$ {5} A 1.0 A is under low load. $t = 1$ {10} A 1.0 A is under high load, so new instances spawn.

[1] Living cells replicate by splitting in two. Mitosis is the name of this process.

Name	Kill Switch
Motive	Disable large parts of the system to limit damage.
Description	Microservices systems are complex systems, like the Three Mile Island reactor. Failure events at all scales are to be expected. Empirically these events follow a power law in terms of occurrence. Eventually an event with potential for significant damage occurs. To limit the damage, rapid action is needed. As it's impossible to understand the event during its occurrence, the safest course of action is to "scram" the system. You should be able to shutdown large parts of the system using secondary communication links to each microservice. As the event progresses, you may need to progressively shutdown more and more of the system to eventually contain the damage.
Sequence	 $t = 0$ {10} A 1.0 {10} B 1.0 {10} C 1.0 {10} D 1.0 Severe failures detected. $t = 1$ {10} A 1.0 {10} B 1.0 {0} C 1.0 {0} D 1.0 Kill half the system. $t = 2$ {10} A 1.0 {10} B 1.0 {1} C 1.0 {1} D 1.0 Restart fresh instances. Progressive Canary.

5.7.4 Validation

Continuous validation of the production system is the key practice that makes microservices successful. No other activity provides as much risk reduction and value. It's the only way to run a continuous delivery pipeline responsibly.

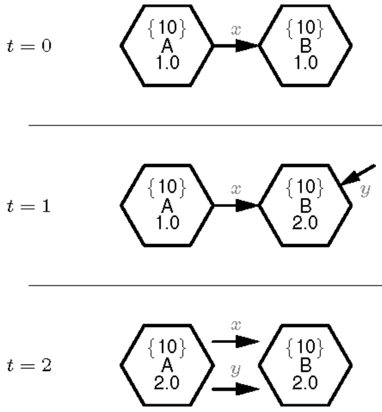
What do you measure in production? CPU load levels? Memory usage? These are useful, but not essential. Far more important is validation that the system is behaving as desired. Messages correspond to business activities, or parts of business activities, and the behavior of messages is the thing to focus on. Message flow rates tell you a great deal about the health of the system. On their own, they're still less useful than you think, as they fluctuate with the time of day and with seasonality.

It's more useful to compare message flow rates with each other. A given business process is encoded by an expanding set of messages generated from an initial triggering message. Message flow rates are related to each by ratios. For every message of certain kind, you expect to see two messages of another kind. These ratios don't change no matter what the load level of the system, or the amount of services present. They are **invariant**.

Invariants are your primary indicator of health. When you deploy a new version of a microservice using the Canary pattern, what you check is that the invariants are within expected bounds. If the new version contains a defect, the message flow ratios changes, as some messages won't be generated. This is an immediate indicator of failure. Invariants, after all, can't vary. We'll come back to this topic in Chapter 6 and provide an example in Chapter 9.

YET MORE MICROSERVICE DEPLOYMENT PATTERNS

Name	Version Update
Motive	Safely update a set of communicating microservices.
Description	Microservices A and B communicate using messages of kind x. New business requirements introduce the need for messages of kind y between the services. It's unwise to attempt a simultaneous update of both. It's preferable to use the Progressive Canary deployment pattern to make the change safely. First update the listening service B, allowing it to recognize the new message y. No other services generate this message in production, yet, but we can validate that the new version of B doesn't cause damage. Once the new B is in place, update A, which emits y messages. This multi-stage update (composed of Progressive Canaries for each stage) can be used for many scenarios where the message interactions must change. It can be used when the internal data of the messages change (B in this case must retain the ability to handle old messages until the change is complete). It can be used to inject a third service between two existing services, by applying the pattern first to one side of the interaction, and then the other. This is a common way to introduce a cache, such as the one seen in Chapter 1.

Sequence	 $t = 0$ {10} A 1.0 x {10} B 1.0 $t = 1$ {10} A 1.0 x {10} B 2.0 y $t = 2$ {10} A 2.0 x y {10} B 2.0 A sends B messages of kind x. B/2.0 can also handle y messages. A/1.0 continues to send only x messages. A/2.0 can now send x and y messages. No downtime was required for this transition.

Name	Chaos
Motive	Ensure the system is resistant to failure by constantly failing at a low rate.
Description	Services can develop fragile dependencies on other services, despite your best intentions. When dependencies fail, even when that failure is below the threshold, this can cause threshold failures in services that rely on you. To prevent creeping fragility, deliberately fail your services on a continuous basis, in production. Calibrate the failure to be well below the failure threshold for the business to prevent a significant impact on business outcomes. This is effectively a form of insurance. You take small frequent losses to avoid large infrequent losses that are fatal. The most famous example of this pattern is the Netflix Chaos Monkey, which randomly shuts down services in the Netflix infrastructure. Another example is Google Game Days, where large-scale production failures are deliberately triggered to test failover capabilities.

5.7.5 Discovery

Pattern matching and transport independence give you decoupled services. When microservice A knows that microservice B receives its messages, then A is coupled to B. Unfortunately message transport requires knowledge of the location of B, otherwise messages can't be delivered. Transport independence hides the mechanism of transportation from A. Pattern matching hides the identity of B. Identity is coupling.

The messaging abstraction layer still needs to know the location of B, even as it hides this information from A. A (or least, the message layer in A) needs to discover the location of B (this is the set of locations of all the instances of B). This is a primary infrastructural challenge in microservice systems. Let's examine the common solutions:

Embedded configuration

Hard code service locations as part of the immutable artifact.

Intelligent load-balancing

Direct all message traffic through load-balancers that know where to find the services.

Service registries

Services register their location with a central registry, and other services look them up in the registry.

DNS

Use the DNS protocol to resolve the location of a service.

Message bus

Use a message bus to separate publishers from subscribers.

Gossip

Use a peer-to-peer membership gossip protocol to share service locations.

No solution is perfect, and each involves trade-offs:

Discovery	Advantages	Disadvantages
Embed config.	Easy implementation. Works (mostly) for small static systems.	Doesn't scale as large systems are under continuous change. Strong identity concept: raw network location.
Intelligent LB.	Scalable with proven production quality tooling[1]	Non-microservice network element that requires separate management. Load-balancers force limited transport options, and must still discover service locations themselves using one of the other discovery mechanisms. Retains identity concept: request URL.
Registry	Scalable with proven production quality tooling[2].	Non-microservice network element. High dependency on chosen solution as no common standards. Strong identity concept: service name key.
DNS	Unlimited scale and proven production quality tooling. Well understood. Can be used by other mechanisms to weaken identity by replacing raw network locations	Non-microservice network element. Management overhead. Weak identity concept: hostname.
Message Bus	Scalable with proven production quality tooling[3].	Non-microservice network element. Management overhead. Weak identity concept: topic name.
Gossip	No concept of identity! Doesn't require additional network elements. Early adopter stage, but shown to work at scale[4].	Message layer must have additional intelligence to handle load-balancing. Rapidly evolving implementations.

[1] Some examples are nginx, and Netflix's Hystrix

[2] Some examples are Consul, Zookeeper, and etcd

[3] Some examples are RabbitMQ, Kafka, and NServiceBus

[4] The SWIM algorithm has found success at Uber: https://eng.uber.com/intro-to-ringpop

5.7.6 Configuration

How do you configure your microservices. This isn't an innocent question. Configuration is one of the primary causes of deployment failure. Does configuration live with the service, immutably packaged into the artifact? Or does configuration live on the network, able to adapt dynamically to live conditions, and providing an additional way to control services?

If configuration is packaged with the service, then configuration changes aren't different from code changes, and must be pushed through the continuous delivery pipeline in the same manner. Although this provides better risk management, it also means you may suffer unacceptable delays when you need to make changes to configuration. You're also adding additional entries to the artifact store and audit logs for every single configuration change, which can clutter these databases and make them less useful. Finally, some network components, such as intelligent load-balancers, still need dynamic configuration if they're to be useful, and you can't place all configuration in artifacts.

On the other hand, network configuration removes your ability to reproduce the system deterministically, or to have the full benefit of the safety of immutability. The same artifact deployed tomorrow could fail even though it worked today. You need to define separate change control processes and controls for configuration as you won't be able to reuse the artifact pipeline for this purpose. You'll need to deploy network services and infrastructure to store and serve configuration. Even if most configuration is dynamic, you still need to bake in at least some configuration—the network location of the configuration store! When you look closely you'll find that many services have large amounts of potential configuration arising from third party libraries that you include in your services. You need to decide to what extent you'll expose this via the dynamic configuration store. You are unlikely to find much value in exposing all of it, and ends up with artifact generation in any case when you need to change this deeper configuration. You'll need to decide how to make this compatible with your management of higher level configuration.

The result will be a hybrid solution, as neither approach provides a total solution. The immutable packaging approach has the advantage of reusing the delivery pipeline as a control mechanism, and offering more predictable state. Placing most of your configuration into immutable artifacts is a reasonable trade-off. Nonetheless, you should plan for the provision and management of dynamic configuration.

Two dangerous anti-patterns should be avoided when it comes to configuration. Using microservices doesn't protect you from them; remain vigilant!

Automation workarounds

Configuration can be used to code around limitations of your automation tooling. For example, using feature flags rather than generating new artifacts. Too much of this and you create an uncontrolled secondary command structure that damages the properties of the system that makes immutability powerful.

Turing disease

Configuration formats tend to be extended with programming constructs over time, mostly as conveniences to avoid repetition[1]. Now you've a new unasked-for programming language in your system with no formal grammar, undefined behavior, and no debugging tools. Good luck!

5.7.7 Security

The microservice architecture doesn't offer any inherent security benefits, and introduces new attack vectors if care isn't taken. A common temptation is to share microservice messages with the outside world. This is dangerous, as it exposes every microservice as an attack surface.

There must be an absolute separation. You need a "Demilitarized Zone" (DMZ) between the internal world of microservice messages, and the outside world of third party clients. The DMZ must translate between the two. In practice this means that a microservice system should expose traditional integration points, such as REST APIs, and then convert requests to these APIs into messages. This allows for strict sanitization of input.

Internally, you can't ignore that microservices communicate over a network, and networks represent an opportunity for attack. Your microservice should live within their own private networks with well-defined ingress and egress routes. The rest of the system uses these routes to interact with the microservice system. The specific microservices to which messages are routed aren't exposed.

These precautions may still be insufficient, and you need to consider the case where an attacker has some level of access to the microservice network. You can apply the security principle of "defense in depth" to strengthen your security in layers. It's always a trade-off between stronger security and operational impact.

Let's build up a few layers. Microservices can be given a right of refusal. They can be made more pedantic in the messages they accept. Highly paranoid services lead to higher error rates, but can delay attackers and make attacks more expensive. For example, you can limit the amount of data that a service returns for any one request. This approach means custom work for each service.

Communication between services can require shared secrets, and as a further layer, signed messages. This protects against messages injected into the network by an attacker. The distribution and cycling of the secrets introduces operational complexity. The signing of messages requires key distribution and introduces latency.

If your data is sensitive, you can encrypt all communication between microservices. This also introduces latency and management overhead, and isn't to be undertaken lightly. Consider using the Merge pattern for extremely sensitive data flows to avoid the network as much as possible.

[1] Initially declarative domain specific languages, such as configuration formats, tend to accumulate programmatic features over time. It's surprisingly easy to achieve Turing completeness with a limited set of operations.

The need for secure storage and management of secrets and encryption keys is necessary if these layers are to be effective. It's pointless to encrypt messages if the keys are easily accessible within the network. And yet, microservices need to be able to access the secrets and keys. To solve this problem, you need to introduce another network element—a key management service that provides secure storage, access control, and audit capabilities[1].

5.7.8 Staging

The staging system is the control mechanism for the continuous delivery pipeline. It encompasses traditional elements, such as a build server for continuous integration. It can can also consist of multiple systems that test various aspects of the system, such as performance.

The staging system can also be used to provide manual gates to the delivery pipeline. These are often unavoidable, either politically, or legally. Over time, the effectiveness of continuous delivery in managing risk and delivering business value quickly can be used to create sufficient organizational confidence to relax overly ceremonial manual sign-offs.

The staging system provides a self-service mechanism for development teams to push updates all the way to production. The empowerment of developer teams to do this is a critical component in the success of continuous delivery. We'll discuss this human factor in Chapter 7.

Staging should collect statistics to measure the velocity and quality of code delivery over time. It's important to know how long it takes, on average, to take code from concept to production, for a given risk level, as this tells you how efficient your continuous delivery pipeline is.

The staging system has the most variance between projects and organizations. The level of testing, the number of staging steps, and the mechanism of artifact generation, are all highly context specific. As you grow the use of microservices and continuous delivery in your organization, you should avoid being too prescriptive in your definition of the staging function. You must allow teams to adapt to their own circumstances.

5.7.9 Development

The development environment needed for microservices should enable the developer to focus on a small set of services at a time, often a single service. The message abstraction layer comes into its own here as it makes it easy to mock the behavior of other services[2]. Instead of having to implement a complex object hierarchy, microservice mocking only requires implementing sample message flows. This makes it possible to unit test microservices in complete isolation from the rest of the system.

[1] Some examples are Hashicorp Vault, AWS KVS and if money is no object, HSMs (Hardware Security Modules).

[2] For a practical example, see the code in Chapter 9.

Microservices can be specified as a relation between inbound and outbound messages. This allows the developer to focus on a small part of the system. It also enables more efficient parallel work, as messages from other microservices (which may not even be written) can easily be mocked.

Isolation isn't always possible or appropriate. Developers often need to run small subsets of the system locally. Tooling is needed to make this practical. It isn't advisable for development to become dependent on running a full replica of the production system. As production grows to hundreds of different services, and beyond, it becomes extremely resource intensive to run services locally, and ultimately it becomes impossible.

If the developer is running only a subset of the system, how do you ensure that appropriate messages are provided for the other parts of the system? One common anti-pattern is to use the build or staging systems to do this. Developers end up working against shared resources that have extremely non-deterministic state. This is the same anti-pattern as having a shared development database.

Each developer should provide a set of mock messages for their service. Where do these mock messages live? At one extreme you can place all mock message flows in a common mocking service. All developers commit code to this service, but conflicts are still rare, as work isn't likely to overlap. At the other extreme you can provide a mock service along with each service implementation. The mock service is an extremely simple service that returns hard-coded responses.

The practical solution for most teams is somewhere in the middle. Start with a single universal mocking service, and apply the Split pattern whenever it becomes too unwieldy. Sets of services with a common focus tend to get their own mocking service. The development environment is typically a small set of actual services, along with one or two mocking services. This keeps the number of service processes needed on a developer machine to a minimum.

The mock messages are defined by the developers building a given microservice. This has the unfortunate side-effect that the developer focuses on expected behavior. Others use their service in unexpected ways, and the mocking is incomplete. If you allow other developers to add mock messages to services they don't own, then the mocks quickly diverge from reality. The solution is to add captured messages to the list of sample messages. Capture sample message flows from the production or staging logs, and add them to the mock service. This can be done manually for even medium-sized systems.

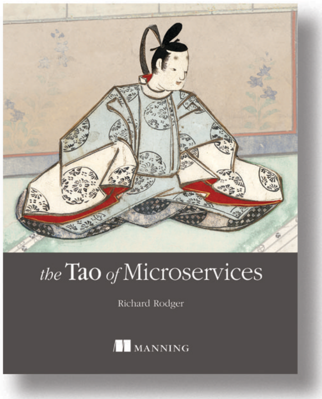
5.8 Summary

This chapter examined the nature of failure in complex systems. Failure is inevitable, and must be accepted. Starting from that perspective, you can work to distribute failure more evenly over time, and avoid high impact catastrophes.

- Traditional approaches to software quality are predicated on a false belief in perfectionism. Enterprise software systems aren't desktop calculators, and won't give the correct answer 100% of the time. The closer you want to get to 100%, the more expensive it gets.

- The risk of failure is much higher than generally believed. Simple mathematical modeling of risk probabilities in component based systems (such as enterprise software) brings this reality starkly into focus.
- Microservices provide an opportunity to measure risk more accurately. This enables you to define acceptable error rates that your system must meet.
- By packaging microservices into immutable units of deployment, you can define a set of deployment patterns that mitigate risk and can be automated for efficient management of your production systems.
- The accurate measurement of risk enables the construction of a continuous delivery pipeline that enables developers to push changes to microservices to production a high velocity and high frequency, whilst maintaining acceptable risk levels.

This chapter refrained from prescribing specific techniques and tooling. As a software architect you need the freedom to make these decisions yourself. Rather, the focus on fundamentals, and basic patterns gives you a framework to help make these decisions. You'll need to adapt the principles in this chapter to your own context.



Microservices are small, but they offer big value. A microservice is a very small piece of a larger system that can be coded by one developer within one iteration. Microservices can be added and removed individually, new developers can be immediately productive, and legacy code is easily replaced. Developers are no longer hampered by the communication and coordination overhead caused by monolithic systems. Savvy businesses are discovering that software development productivity can be greatly enhanced with the right engineering approach that enables even junior developers to double their productivity, while reducing delivery risk.

The Tao of Microservices teaches you the path to understanding how to apply microservices architecture with your own real-world projects. This high-level book offers you a conceptual view of microservice architectures, along with core concepts and their application. You'll also find a detailed case study for the nodezoo.com system, including all source code and documentation. By the end of the book, you'll have explored in depth the key ideas of the microservice architecture and will be able to design, analyze and implement systems based on this architecture.

What's inside:

- Key principles of the microservice architecture
- Applying these principles to real-world projects
- Implementing large-scale systems
- Detailed case study

This book is for developers, architects, or managers who want to deliver faster, meet changing business requirements, and build scalable and robust systems.

Running Software in Containers

Utilizing microservices leads to many, many deployments all practicing continuous delivery at the same time, so it is important to base deployments on easy-to-use yet robust technologies. One such technology is Docker. This chapter gives you a taste for what Docker can do, and you will see that it fits well with the techniques described in the previous chapter.

Chapter 2 from *Docker in Action* by Jeff Nickoloff.

Running software in containers

This chapter covers

- Running interactive and daemon terminal programs with containers
- Containers and the PID namespace
- Container configuration and output
- Running multiple programs in a container
- Injecting configuration into containers
- Durable containers and the container life cycle
- Cleaning up

Before the end of this chapter you'll understand all the basics for working with containers and how Docker helps solve clutter and conflict problems. You're going to work through examples that introduce Docker features as you might encounter them in daily use.

2.1 Getting help with the Docker command line

You'll use the docker command-line program throughout the rest of this book. To get you started with that, I want to show you how to get information about commands from the docker program itself. This way you'll understand how to use the exact version of Docker on your computer. Open a terminal, or command prompt, and run the following command:

```
docker help
```

Running docker help will display information about the basic syntax for using the docker command-line program as well as a complete list of commands for your version of the program. Give it a try and take a moment to admire all the neat things you can do.

docker help gives you only high-level information about what commands are available. To get detailed information about a specific command, include the command in the <COMMAND> argument. For example, you might enter the following command to find out how to copy files from a location inside a container to a location on the host machine:

```
docker help cp
```

That will display a usage pattern for docker cp, a general description of what the command does, and a detailed breakdown of its arguments. I'm confident that you'll have a great time working through the commands introduced in the rest of this book now that you know how to find help if you need it.

2.2 Controlling containers: building a website monitor

Most examples in this book will use real software. Practical examples will help introduce Docker features and illustrate how you will use them in daily activities. In this first example, you're going to install a web server called NGINX. Web servers are programs that make website files and programs accessible to web browsers over a network. You're not going to build a website, but you are going to install and start a web server with Docker. If you follow the instructions in this example, the web server will be available only to other programs on your computer.

Suppose a new client walks into your office and makes you an outrageous offer to build them a new website. They want a website that's closely monitored. This particular client wants to run their own operations, so they'll want the solution you provide to email their team when the server is down. They've also heard about this popular web server software called NGINX and have specifically requested that you use it. Having read about the merits of working with Docker, you've decided to use it for this project. Figure 2.1 shows your planned architecture for the project.

This example uses three containers. The first will run NGINX; the second will run a program called a mailer. Both of these will run as detached containers. *Detached* means that the container will run in the background, without being attached to any input or output stream. A third program, called an agent, will run in an interactive container. Both the mailer and agent are small scripts created for this example. In this section you'll learn how to do the following:

- Create detached and interactive containers
- List containers on your system

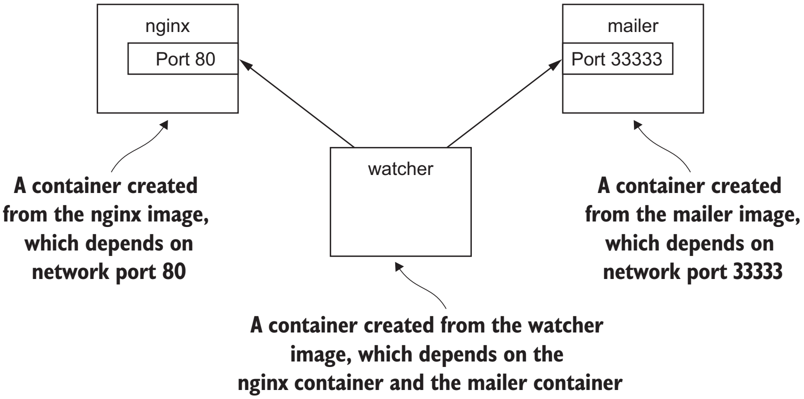

nginx
Port 80
mailer
Port 33333
watcher
A container created from the nginx image, which depends on network port 80
A container created from the mailer image, which depends on network port 33333
A container created from the watcher image, which depends on the nginx container and the mailer container

Figure 2.1 The three containers that you'll build in this example

- View container logs
- Stop and restart containers
- Reattach a terminal to a container
- Detach from an attached container

Without further delay, let's get started filling your client's order.

2.2.1 Creating and starting a new container

When installing software with Docker, we say that we're installing an *image*. There are different ways to install an image and several sources for images. Images are covered in depth in chapter 3. In this example we're going to download and install an image for NGINX from Docker Hub. Remember, Docker Hub is the public registry provided by Docker Inc. The NGINX image is from what Docker Inc. calls a trusted repository. Generally, the person or foundation that publishes the software controls the trusted repositories for that software. Running the following command will download, install, and start a container running NGINX:

```
docker run --detach \
    --name web nginx:latest
```

Note the detach flag

When you run this command, Docker will install `nginx:latest` from the NGINX repository hosted on Docker Hub (covered in chapter 3) and run the software. After Docker has installed and started running NGINX, one line of seemingly random characters will be written to the terminal. It will look something like this:

```
7cb5d2b9a7eab87f07182b5bf58936c9947890995b1b94f412912fa822a9ecb5
```

That blob of characters is the unique identifier of the container that was just created to run NGINX. Every time you run `docker run` and create a new container, that container will get a similar unique identifier. It's common for users to capture this output with a variable for use with other commands. You don't need to do so for the purposes of this example. After the identifier is displayed, it might not seem like anything has happened. That's because you used the `--detach` option and started the program in the background. This means that the program started but isn't attached to your terminal. It makes sense to start NGINX this way because we're going to run a few different programs.

Running detached containers is a perfect fit for programs that sit quietly in the background. That type of program is called a *daemon*. A daemon generally interacts with other programs or humans over a network or some other communication tool. When you launch a daemon or other program in a container that you want to run in the background, remember to use either the `--detach` flag or its short form, `-d`.

Another daemon that your client needs is a mailer. A mailer waits for connections from a caller and then sends an email. The following command will install and run a mailer that will work for this example:

```
docker run -d \
    --name mailer \
```

Start detached

This command uses the short form of the `--detach` flag to start a new container named mailer in the background. At this point you've run two commands and delivered two-thirds of the system that your client wants. The last component, called the agent, is a good fit for an interactive container.

2.2.2 Running interactive containers

Programs that interact with users tend to feel more interactive. A terminal-based text editor is a great example. The `docker` command-line tool is a perfect example of an interactive terminal program. These types of programs might take input from the user or display output on the terminal. Running interactive programs in Docker requires that you bind parts of your terminal to the input or output of a running container.

To get started working with interactive containers, run the following command:

```
docker run --interactive --tty \
    --link web:web \
    --name web_test \
    busybox:latest /bin/sh
```

Create a virtual terminal and bind stdin

The command uses two flags on the `run` command: `--interactive` (or `-i`) and `--tty` (or `-t`). First, the `--interactive` option tells Docker to keep the standard input stream (stdin) open for the container even if no terminal is attached. Second, the `--tty` option tells Docker to allocate a virtual terminal for the container, which will allow you to pass signals to the container. This is usually what you want from an interactive command-line program. You'll usually use both of these when you're running an interactive program like a shell in an interactive container.

Just as important as the interactive flags, when you started this container you specified the program to run inside the container. In this case you ran a shell program called sh. You can run any program that's available inside the container.

The command in the interactive container example creates a container, starts a UNIX shell, and is linked to the container that's running NGINX (linking is covered in chapter 5). From this shell you can run a command to verify that your web server is running correctly:

```
wget -O - http://web:80/
```

This uses a program called `wget` to make an HTTP request to the web server (the NGINX server you started earlier in a container) and then display the contents of the web page on your terminal. Among the other lines, there should be a message like "Welcome to NGINX!" If you see that message, then everything is working correctly and you can go ahead and shut down this interactive container by typing `exit`. This will terminate the shell program and stop the container.

It's possible to create an interactive container, manually start a process inside that container, and then detach your terminal. You can do so by holding down the Crtl (or Control) key and pressing P and then Q. This will work only when you've used the `--tty` option.

To finish the work for your client, you need to start an agent. This is a monitoring agent that will test the web server as you did in the last example and send a message with the mailer if the web server stops. This command will start the agent in an interactive container using the short-form flags:

```
docker run -it \
    --name agent \
    --link web:insideweb \
    --link mailer:insidemailer \
    dockerinaction/ch2_agent
```

Create a virtual terminal and bind stdin

When running, the container will test the web container every second and print a message like the following:

```
System up.
```

Now that you've seen what it does, detach your terminal from the container. Specifically, when you start the container and it begins writing "System up," hold the Ctrl (or Control) key and then press P and then Q. After doing so you'll be returned to the shell for your host computer. Do not stop the program; otherwise, the monitor will stop checking the web server.

Although you'll usually use detached or daemon containers for software that you deploy to servers on your network, interactive containers are very useful for running software on your desktop or for manual work on a server. At this point you've started all three applications in containers that your client needs. Before you can confidently claim completion, you should test the system.

2.2.3 Listing, stopping, restarting, and viewing output of containers

The first thing you should do to test your current setup is check which containers are currently running by using the `docker ps` command:

```
docker ps
```

Running the command will display the following information about each running container:

- The container ID
- The image used
- The command executed in the container
- The time since the container was created
- The duration that the container has been running
- The network ports exposed by the container
- The name of the container

At this point you should have three running containers with names: web, mailer, and agent. If any is missing but you've followed the example thus far, it may have been mistakenly stopped. This isn't a problem because Docker has a command to restart a container. The next three commands will restart each container using the container name. Choose the appropriate ones to restart the containers that were missing from the list of running containers.

```
docker restart web
docker restart mailer
docker restart agent
```

Now that all three containers are running, you need to test that the system is operating correctly. The best way to do that is to examine the logs for each container. Start with the web container:

```
docker logs web
```

That should display a long log with several lines that contain this substring:

```
"GET / HTTP/1.0" 200
```

This means that the web server is running and that the agent is testing the site. Each time the agent tests the site, one of these lines will be written to the log. The `docker logs` command can be helpful for these cases but is dangerous to rely on. Anything that the program writes to the stdout or stderr output streams will be recorded in this log. The problem with this pattern is that the log is never rotated or truncated, so the data written to the log for a container will remain and grow as long as the container exists. That long-term persistence can be a problem for long-lived processes. A better way to work with log data uses volumes and is discussed in chapter 4.

You can tell that the agent is monitoring the web server by examining the logs for web alone. For completeness you should examine the log output for mailer and agent as well:

```
docker logs mailer
docker logs agent
```

The logs for mailer should look something like this:

```
CH2 Example Mailer has started.
```

The logs for agent should contain several lines like the one you watched it write when you started the container:

```
System up.
```

TIP The `docker logs` command has a flag, `--follow` or `-f`, that will display the logs and then continue watching and updating the display with changes to the log as they occur. When you've finished, press Ctrl (or Command) and the C key to interrupt the `logs` command.

Now that you've validated that the containers are running and that the agent can reach the web server, you should test that the agent will notice when the web container stops. When that happens, the agent should trigger a call to the mailer, and the event should be recorded in the logs for both agent and mailer. The `docker stop` command tells the program with PID #1 in the container to halt. Use it in the following commands to test the system:

```
docker stop web
docker logs mailer
```

Stop the web server by stopping the container

Wait a couple seconds and check the mailer logs

Look for a line at the end of the mailer logs that reads like:

```
"Sending email: To: admin@work  Message: The service is down!"
```

That line means the agent successfully detected that the NGINX server in the container named web had stopped. Congratulations! Your client will be happy, and you've built your first real system with containers and Docker.

Learning the basic Docker features is one thing, but understanding why they're useful and how to use them in building more comprehensive systems is another task entirely. The best place to start learning that is with the process identifier namespace provided by Linux.

2.3 Solved problems and the PID namespace

Every running program—or process—on a Linux machine has a unique number called a process identifier (PID). A PID namespace is the set of possible numbers that identify processes. Linux provides facilities to create multiple PID namespaces. Each namespace has a complete set of possible PIDs. This means that each PID namespace will contain its own PID 1, 2, 3, and so on. From the perspective of a process in one namespace, PID 1 might refer to an init system process like `runit` or `supervisord`. In a different namespace, PID 1 might refer to a command shell like bash. Creating a PID namespace for each container is a critical feature of Docker. Run the following to see it in action:

```
docker run -d --name namespaceA \
    busybox:latest /bin/sh -c "sleep 30000"
docker run -d --name namespaceB \
    busybox:latest /bin/sh -c "nc -l -p 0.0.0.0:80"

docker exec namespaceA ps      ❶
docker exec namespaceB ps             ❷
```

Command ❶ above should generate a process list similar to the following:

```
PID   USER     COMMAND
  1   root     /bin/sh -c sleep 30000
  5   root     sleep 30000
  6   root     ps
```

Command ❷ above should generate a slightly different process list:

```
PID   USER     COMMAND
  1   root     /bin/sh -c nc -l -p 0.0.0.0:80
  7   root     nc -l -p 0.0.0.0:80
  8   root     ps
```

In this example you use the `docker exec` command to run additional processes in a running container. In this case the command you use is called `ps`, which shows all the running processes and their PID. From the output it's clear to see that each container has a process with PID 1.

Without a PID namespace, the processes running inside a container would share the same ID space as those in other containers or on the host. A container would be able to determine what other processes were running on the host machine. Worse, namespaces transform many authorization decisions into domain decisions. That means processes in one container might be able to control processes in other containers. Docker would be much less useful without the PID namespace. The Linux features that Docker uses, such as namespaces, help you solve whole classes of software problems.

Like most Docker isolation features, you can optionally create containers without their own PID namespace. You can try this yourself by setting the `--pid` flag on `docker create` or `docker run` and setting the value to `host`. Try it yourself with a container running BusyBox Linux and the `ps` Linux command:

```
docker run --pid host busybox:latest ps
```

Should list all processes running on the computer

Consider the previous web-monitoring example. Suppose you were not using Docker and were just running NGINX directly on your computer. Now suppose you forgot that you had already started NGINX for another project. When you start NGINX again, the second process won't be able to access the resources it needs because the first process already has them. This is a basic software conflict example. You can see it in action by trying to run two copies of NGINX in the same container:

```
docker run -d --name webConflict nginx:latest
docker logs webConflict
docker exec webConflict nginx -g 'daemon off;'
```

The output should be empty

Start a second nginx process in the same container

The last command should display output like:

```
2015/03/29 22:04:35 [emerg] 10#0: bind() to 0.0.0.0:80 failed (98:
Address already in use)
nginx: [emerg] bind() to 0.0.0.0:80 failed (98: Address already in use)
...
```

The second process fails to start properly and reports that the address it needs is already in use. This is called a port conflict, and it's a common issue in real-world systems where several processes are running on the same computer or multiple people contribute to the same environment. It's a great example of a conflict problem that Docker simplifies and solves. Run each in a different container, like this:

```
docker run -d --name webA nginx:latest

docker logs webA

docker run -d --name webB nginx:latest

docker logs webB
```

Start the first nginx instance

Verify that it is working, should be empty

Start the second instance

Verify that it is working, should be empty

To generalize ways that programs might conflict with each other, let's consider a parking lot metaphor. A paid parking lot has a few basic features: a payment system, a few reserved parking spaces, and numbered spaces.

Tying these features back to a computer system, a payment system represents some shared resource with a specific interface. A payment system might accept cash or credit cards or both. People who carry only cash won't be able to use a garage with a payment system that accepts only credit cards, and people without money to pay the fee won't be able to park in the garage at all.

Similarly, programs that have a dependency on some shared component such as a specific version of a programming language library won't be able to run on computers that either have a different version of that library or lack that library completely. Just like if two people who each use a different payment method want to park in the same garage that accepts only one method, conflict arises when you want to use two programs that require different versions of a library.

Reserved spaces in this metaphor represent scarce resources. Imagine that the parking garage attendant assigns the same reserved space to two cars. As long as only one driver wanted to use the garage at a time, there would be no issue. But if both wanted to use the space simultaneously, the first one in would win and the second wouldn't be able to park. As you'll see in the conflict example in section 2.7, this is the same type of conflict that happens when two programs try to bind to the same network port.

Lastly, consider what would happen if someone changed the space numbers in the parking lot while cars were parked. When owners return and try to locate their vehicles, they may be unable to do so. Although this is clearly a silly example, it's a great metaphor for what happens to programs when shared environment variables change. Programs often use environment variables or registry entries to locate other resources that they need. These resources might be libraries or other programs. When programs conflict with each other, they might modify these variables in incompatible ways.

Here are some common conflict problems:

- Two programs want to bind to the same network port.
- Two programs use the same temporary filename, and file locks are preventing that.
- Two programs want to use different versions of some globally installed library.
- Two copies of the same program want to use the same PID file.
- A second program you installed modified an environment variable that another program uses. Now the first program breaks.

All these conflicts arise when one or more programs have a common dependency but can't agree to share or have different needs. Like in the earlier port conflict example, Docker solves software conflicts with such tools as Linux namespaces, file system roots, and virtualized network components. All these tools are used to provide isolation to each container.

2.4 Eliminating metaconflicts: building a website farm

In the last section you saw how Docker helps you avoid software conflicts with process isolation. But if you're not careful, you can end up building systems that create *metaconflicts,* or conflicts between containers in the Docker layer.

Consider another example where a client has asked you to build a system where you can host a variable number of websites for their customers. They'd also like to employ the same monitoring technology that you built earlier in this chapter. Simply expanding the system you built earlier would be the simplest way to get this job done without customizing the configuration for NGINX. In this example you'll build a system with several containers running web servers and a monitoring agent (agent) for each web server. The system will look like the architecture described in figure 2.2.

One's first instinct might be to simply start more web containers. That's not as simple as it sounds. Identifying containers gets complicated as the number of containers increases.

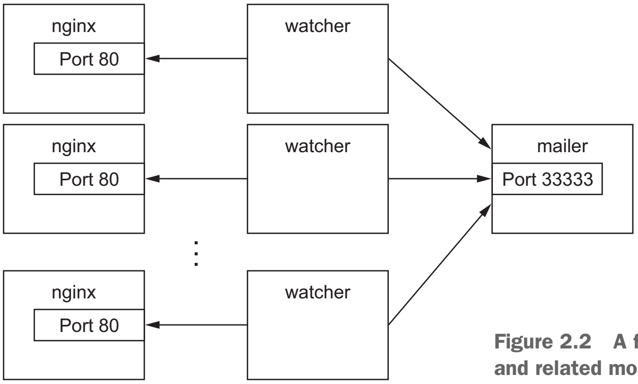


Figure 2.2 A fleet of web server containers and related monitoring agents

2.4.1 Flexible container identification

The best way to find out why simply creating more copies of the NGINX container you used in the last example is a bad idea is to try it for yourself:

```
docker run -d --name webid nginx

docker run -d --name webid nginx
```

Create a container named "webid"

Create another container named "webid"

The second command here will fail with a conflict error:

```
FATA[0000] Error response from daemon: Conflict. The name "webid" is
already in use by container 2b5958ba6a00. You have to delete (or rename)
that container to be able to reuse that name.
```

Using fixed container names like *web* is useful for experimentation and documentation, but in a system with multiple containers, using fixed names like that can create conflicts. By default Docker assigns a unique (human-friendly) name to each container it creates. The `--name` flag simply overrides that process with a known value. If a situation arises where the name of a container needs to change, you can always rename the container with the `docker rename` command:

```
docker rename webid webid-old

docker run -d --name webid nginx
```

Rename the current web container to "webid-old"

Create another container named "webid"

Renaming containers can help alleviate one-off naming conflicts but does little to help avoid the problem in the first place. In addition to the name, Docker assigns a unique identifier that was mentioned in the first example. These are hex-encoded 1024-bit numbers and look something like this:

```
7cb5d2b9a7eab87f07182b5bf58936c9947890995b1b94f412912fa822a9ecb5
```

When containers are started in detached mode, their identifier will be printed to the terminal. You can use these identifiers in place of the container name with any command that needs to identify a specific container. For example, you could use the previous ID with a `stop` or `exec` command:

```
docker exec \
    7cb5d2b9a7eab87f07182b5bf58936c9947890995b1b94f412912fa822a9ecb5 \
ps

docker stop \
    7cb5d2b9a7eab87f07182b5bf58936c9947890995b1b94f412912fa822a9ecb5
```

The high probability of uniqueness of the IDs that are generated means that it is unlikely that there will ever be a collision with this ID. To a lesser degree it is also unlikely that there would even be a collision of the first 12 characters of this ID on the same computer. So in most Docker interfaces, you'll see container IDs truncated to their first 12 characters. This makes generated IDs a bit more user friendly. You can use them wherever a container identifier is required. So the previous two commands could be written like this:

```
docker exec 7cb5d2b9a7ea ps
docker stop 7cb5d2b9a7ea
```

Neither of these IDs is particularly well suited for human use. But they work very well with scripts and automation techniques. Docker has several means of acquiring the ID of a container to make automation possible. In these cases the full or truncated numeric ID will be used.

The first way to get the numeric ID of a container is to simply start or create a new one and assign the result of the command to a shell variable. As you saw earlier, when a new container is started in detached mode, the container ID will be written to the terminal (stdout). You'd be unable to use this with interactive containers if this were the only way to get the container ID at creation time. Luckily you can use another command to create a container without starting it. The `docker create` command is very similar to `docker run`, the primary difference being that the container is created in a stopped state:

```
docker create nginx
```

The result should be a line like:

```
b26a631e536d3caae348e9fd36e7661254a11511eb2274fb55f9f7c788721b0d
```

If you're using a Linux command shell like sh or bash, you can simply assign that result to a shell variable and use it again later:

```
CID=$(docker create nginx:latest)
echo $CID
```

This will work on POSIX-compliant shells

Shell variables create a new opportunity for conflict, but the scope of that conflict is limited to the terminal session or current processing environment that the script was launched in. Those conflicts should be easily avoidable because one use or program is managing that environment. The problem with this approach is that it won't help if multiple users or automated processes need to share that information. In those cases you can use a container ID (CID) file.

Both the `docker run` and `docker create` commands provide another flag to write the ID of a new container to a known file:

```
docker create --cidfile /tmp/web.cid nginx

cat /tmp/web.cid
```

Create a new stopped container

Inspect the file

Like the use of shell variables, this feature increases the opportunity for conflict. The name of the CID file (provided after `--cidfile`) must be known or have some known structure. Just like manual container naming, this approach uses known names in a global (Docker-wide) namespace. The good news is that Docker won't create a new container using the provided CID file if that file already exists. The command will fail just as it does when you create two containers with the same name.

One reason to use CID files instead of names is that CID files can be shared with containers easily and renamed for that container. This uses a Docker feature called volumes, which is covered in chapter 4.

> **TIP** One strategy for dealing with CID file-naming collisions is to partition the namespace by using known or predictable path conventions. For example, in this scenario you might use a path that contains all web containers under a known directory and further partition that directory by the customer ID. This would result in a path like /containers/web/customer1/web.cid or /containers/web/customer8/web.cid.

In other cases, you can use other commands like `docker ps` to get the ID of a container. For example, if you want to get the truncated ID of the last created container, you can use this:

```
CID=$(docker ps --latest --quiet)
echo $CID

CID=$(docker ps -l -q)
echo $CID
```

This will work on POSIX-compliant shells

Run again with the short-form flags

> **TIP** If you want to get the full container ID, you can use the `--no-trunc` option on the `docker ps` command.

Automation cases are covered by the features you've seen so far. But even though truncation helps, these container IDs are rarely easy to read or remember. For this reason, Docker also generates human-readable names for each container.

The naming convention uses a personal adjective, an underscore, and the last name of an influential scientist, engineer, inventor, or other such thought leader. Examples of generated names are compassionate_swartz, hungry_goodall, and distracted_turing. These seem to hit a sweet spot for readability and memory. When you're working with the docker tool directly, you can always use docker ps to look up the human-friendly names.

Container identification can be tricky, but you can manage the issue by using the ID and name-generation features of Docker.

2.4.2 Container state and dependencies

With this new knowledge, the new system might looks something like this:

```
MAILER_CID=$(docker run -d dockerinaction/ch2_mailer)
WEB_CID=$(docker create nginx)

AGENT_CID=$(docker create --link $WEB_CID:insideweb \
    --link $MAILER_CID:insidemailer \
    dockerinaction/ch2_agent)
```

Make sure mailer from first example is running

This snippet could be used to seed a new script that launches a new NGINX and agent instance for each of your client's customers. You can use docker ps to see that they've been created:

```
docker ps
```

The reason neither the NGINX nor the agent was included with the output has to do with container state. Docker containers will always be in one of four states and transition via command according to the diagram in figure 2.3.

Neither of the new containers you started appears in the list of containers because docker ps shows only running containers by default. Those containers were specifically created with docker create and never started (the exited state). To see all the containers (including those in the exited state), use the -a option:

```
docker ps -a
```

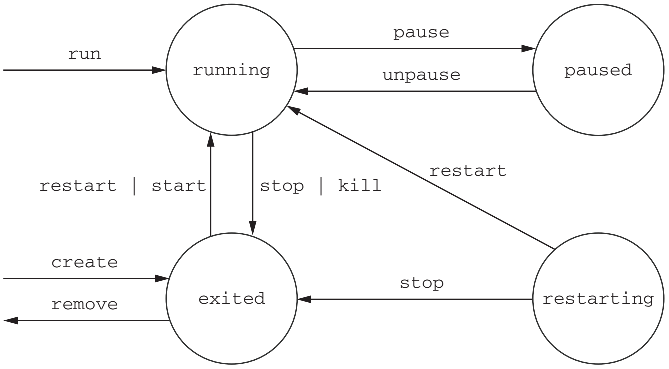


Figure 2.3 The state transition diagram for Docker containers as reported by the status column

Now that you've verified that both of the containers were created, you need to start them. For that you can use the `docker start` command:

```
docker start $AGENT_CID
docker start $WEB_CID
```

Running those commands will result in an error. The containers need to be started in reverse order of their dependency chain. Because you tried to start the agent container before the web container, Docker reported a message like this one:

```
Error response from daemon: Cannot start container
     03e65e3c6ee34e714665a8dc4e33fb19257d11402b151380ed4c0a5e38779d0a: Cannot
     link to a non running container: /clever_wright AS
     /modest_hopper/insideweb
FATA[0000] Error: failed to start one or more containers
```

In this example, the agent container has a dependency on the web container. You need to start the web container first:

```
docker start $WEB_CID
docker start $AGENT_CID
```

This makes sense when you consider the mechanics at work. The link mechanism injects IP addresses into dependent containers, and containers that aren't running don't have IP addresses. If you tried to start a container that has a dependency on a container that isn't running, Docker wouldn't have an IP address to inject. Container linking is covered in chapter 5, but it's useful to demonstrate this important point in starting containers.

Whether you're using `docker run` or `docker create`, the resulting containers need to be started in the reverse order of their dependency chain. This means that circular dependencies are impossible to build using Docker container relationships.

At this point you can put everything together into one concise script that looks like the following:

```
MAILER_CID=$(docker run -d dockerinaction/ch2_mailer)

WEB_CID=$(docker run -d nginx)

AGENT_CID=$(docker run -d \
    --link $WEB_CID:insideweb \
    --link $MAILER_CID:insidemailer \
    dockerinaction/ch2_agent)
```

Now you're confident that this script can be run without exception each time your client needs to provision a new site. Your client has come back and thanked you for the web and monitoring work you've completed so far, but things have changed.

They've decided to focus on building their websites with WordPress (a popular open source content-management and blogging program). Luckily, WordPress is published through Docker Hub in a repository named wordpress:4. All you'll need to deliver is a set of commands to provision a new WordPress website that has the same monitoring and alerting features that you've already delivered.

The interesting thing about content-management systems and other stateful systems is that the data they work with makes each running program specialized. Adam's WordPress blog is different from Betty's WordPress blog, even if they're running the same software. Only the content is different. Even if the content is the same, they're different because they're running on different sites.

If you build systems or software that know too much about their environment—like addresses or fixed locations of dependency services—it's difficult to change that environment or reuse the software. You need to deliver a system that minimizes environment dependence before the contract is complete.

2.5 Building environment-agnostic systems

Much of the work associated with installing software or maintaining a fleet of computers lies in dealing with specializations of the computing environment. These specializations come as global-scoped dependencies (like known host file system locations), hard-coded deployment architectures (environment checks in code or configuration), or data locality (data stored on a particular computer outside the deployment architecture). Knowing this, if your goal is to build low-maintenance systems, you should strive to minimize these things.

Docker has three specific features to help build environment-agnostic systems:

- Read-only file systems
- Environment variable injection
- Volumes

Working with volumes is a big subject and the topic of chapter 4. In order to learn the first two features, consider a requirements change for the example situation used in the rest of this chapter.

WordPress uses a database program called MySQL to store most of its data, so it's a good idea to start with making sure that a container running WordPress has a read-only file system.

2.5.1 Read-only file systems

Using read-only file systems accomplishes two positive things. First, you can have confidence that the container won't be specialized from changes to the files it contains. Second, you have increased confidence that an attacker can't compromise files in the container.

To get started working on your client's system, create and start a container from the WordPress image using the `--read-only` flag:

```
docker run -d --name wp --read-only wordpress:4
```

When this is finished, check that the container is running. You can do so using any of the methods introduced previously, or you can inspect the container metadata directly. The following command will print `true` if the container named wp is running and `false` otherwise.

```
docker inspect --format "{{.State.Running}}" wp
```

The `docker inspect` command will display all the metadata (a JSON document) that Docker maintains for a container. The format option transforms that metadata, and in this case it filters everything except for the field indicating the running state of the container. This command should simply output `false`.

In this case, the container isn't running. To determine why, examine the logs for the container:

```
docker logs wp
```

That should output something like:

```
error: missing WORDPRESS_DB_HOST and MYSQL_PORT_3306_TCP environment
variables
Did you forget to --link some_mysql_container:mysql or set an external db
with -e WORDPRESS_DB_HOST=hostname:port?
```

It appears that WordPress has a dependency on a MySQL database. A database is a program that stores data in such a way that it's retrievable and searchable later. The good news is that you can install MySQL using Docker just like WordPress:

```
docker run -d --name wpdb \
    -e MYSQL_ROOT_PASSWORD=ch2demo \
    mysql:5
```

Once that is started, create a different WordPress container that's linked to this new database container (linking is covered in depth in chapter 5):

```
docker run -d --name wp2 \
    --link wpdb:mysql \
    -p 80 --read-only \
    wordpress:4
```

Use a unique name

Create a link to the database

Check one more time that WordPress is running correctly:

```
docker inspect --format "{{.State.Running}}" wp2
```

You can tell that WordPress failed to start again. Examine the logs to determine the cause:

```
docker logs wp2
```

There should be a line in the logs that is similar to the following:

```
... Read-only file system: AH00023: Couldn't create the rewrite-map mutex
(file /var/lock/apache2/rewrite-map.1)
```

You can tell that WordPress failed to start again, but this time the problem is that it's trying to write a lock file to a specific location. This is a required part of the startup process and is not a specialization. It's appropriate to make an exception to the read-only file system in this case. You need to use a volume to make that exception. Use the following to start WordPress without any issues:

```
# Start the container with specific volumes for read only exceptions
docker run -d --name wp3 --link wpdb:mysql -p 80 \
    -v /run/lock/apache2/ \
    -v /run/apache2/ \
    --read-only wordpress:4
```

Create specific volumes for writeable space

An updated version of the script you've been working on should look like this:

```
SQL_CID=$(docker create -e MYSQL_ROOT_PASSWORD=ch2demo mysql:5)

docker start $SQL_CID

MAILER_CID=$(docker create dockerinaction/ch2_mailer)
docker start $MAILER_CID

WP_CID=$(docker create --link $SQL_CID:mysql -p 80 \
    -v /run/lock/apache2/ -v /run/apache2/ \
    --read-only wordpress:4)

docker start $WP_CID

AGENT_CID=$(docker create --link $WP_CID:insideweb \
    --link $MAILER_CID:insidemailer \
    dockerinaction/ch2_agent)

docker start $AGENT_CID
```

Congratulations, at this point you should have a running WordPress container! By using a read-only file system and linking WordPress to another container running a database, you can be sure that the container running the WordPress image will never change. This means that if there is ever something wrong with the computer running a client's WordPress blog, you should be able to start up another copy of that container elsewhere with no problems.

But there are two problems with this design. First, the database is running in a container on the same computer as the WordPress container. Second, WordPress is using several default values for important settings like database name, administrative user, administrative password, database salt, and so on. To deal with this problem, you could create several versions of the WordPress software, each with a special configuration for the client. Doing so would turn your simple provisioning script into a monster that creates images and writes files. A better way to inject that configuration would be through the use of environment variables.

2.5.2 Environment variable injection

Environment variables are key-value pairs that are made available to programs through their execution context. They let you change a program's configuration without modifying any files or changing the command used to start the program.

Docker uses environment variables to communicate information about dependent containers, the host name of the container, and other convenient information for programs running in containers. Docker also provides a mechanism for a user to inject environment variables into a new container. Programs that know to expect important information through environment variables can be configured at container-creation time. Luckily for you and your client, WordPress is one such program.

Before diving into WordPress specifics, try injecting and viewing environment variables on your own. The UNIX command env displays all the environment variables in the current execution context (your terminal). To see environment variable injection in action, use the following command:

```
docker run --env MY_ENVIRONMENT_VAR="this is a test" \
    busybox:latest \
    env
```

Inject an environment variable

Execute the env command inside the container

The --env flag—or -e for short—can be used to inject any environment variable. If the variable is already set by the image or Docker, then the value will be overridden. This way programs running inside containers can rely on the variables always being set. WordPress observes the following environment variables:

- WORDPRESS_DB_HOST
- WORDPRESS_DB_USER
- WORDPRESS_DB_PASSWORD
- WORDPRESS_DB_NAME
- WORDPRESS_AUTH_KEY
- WORDPRESS_SECURE_AUTH_KEY
- WORDPRESS_LOGGED_IN_KEY
- WORDPRESS_NONCE_KEY
- WORDPRESS_AUTH_SALT
- WORDPRESS_SECURE_AUTH_SALT
- WORDPRESS_LOGGED_IN_SALT
- WORDPRESS_NONCE_SALT

TIP This example neglects the KEY and SALT variables, but any real production system should absolutely set these values.

To get started, you should address the problem that the database is running in a container on the same computer as the WordPress container. Rather than using linking to satisfy WordPress's database dependency, inject a value for the WORDPRESS_DB_HOST variable:

```
docker create --env WORDPRESS_DB_HOST=<my database hostname> wordpress:4
```

This example would create (not start) a container for WordPress that will try to connect to a MySQL database at whatever you specify at <my database hostname>.

Because the remote database isn't likely using any default user name or password, you'll have to inject values for those settings as well. Suppose the database administrator is a cat lover and hates strong passwords:

```
docker create \
    --env WORDPRESS_DB_HOST=<my database hostname> \
    --env WORDPRESS_DB_USER=site_admin \
    --env WORDPRESS_DB_PASSWORD=MeowMix42 \
    wordpress:4
```

Using environment variable injection this way will help you separate the physical ties between a WordPress container and a MySQL container. Even in the case where you want to host the database and your customer WordPress sites all on the same machine, you'll still need to fix the second problem mentioned earlier. All the sites are using the same default database name. You'll need to use environment variable injection to set the database name for each independent site:

```
docker create --link wpdb:mysql \
    -e WORDPRESS_DB_NAME=client_a_wp wordpress:4

docker create --link wpdb:mysql \
    -e WORDPRESS_DB_NAME=client_b_wp wordpress:4
```

For client A

For client B

Now that you've solved these problems, you can revise the provisioning script. First, set the computer to run only a single MySQL container:

```
DB_CID=$(docker run -d -e MYSQL_ROOT_PASSWORD=ch2demo mysql:5)

MAILER_CID=$(docker run -d dockerinaction/ch2_mailer)
```

Then the site provisioning script would be this:

```
if [ ! -n "$CLIENT_ID" ]; then
    echo "Client ID not set”
    exit 1
fi

WP_CID=$(docker create \
    --link $DB_CID:mysql \
    --name wp_$CLIENT_ID \
    -p 80 \
    -v /run/lock/apache2/ -v /run/apache2/ \
    -e WORDPRESS_DB_NAME=$CLIENT_ID \
    --read-only wordpress:4)

docker start $WP_CID

AGENT_CID=$(docker create \
    --name agent_$CLIENT_ID \
    --link $WP_CID:insideweb \
    --link $MAILER_CID:insidemailer \
    dockerinaction/ch2_agent)

docker start $AGENT_CID
```

Assume $CLIENT_ID variable is set as input to script

Create link using DB_CID

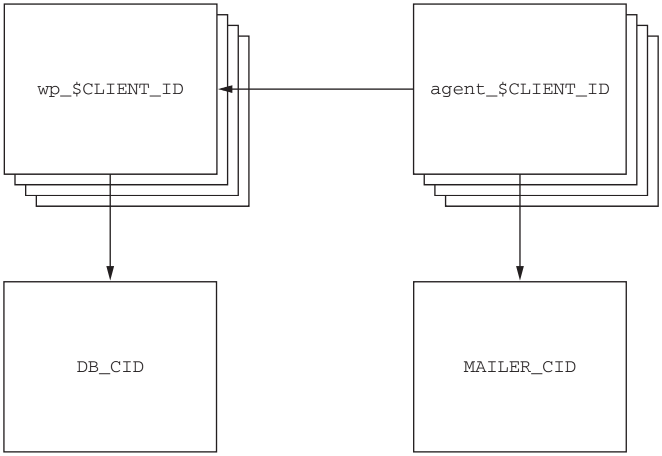


Figure 2.4 Each WordPress and agent container uses the same database and mailer.

This new script will start an instance of WordPress and the monitoring agent for each customer and connect those containers to each other as well as a single mailer program and MySQL database. The WordPress containers can be destroyed, restarted, and upgraded without any worry about loss of data. Figure 2.4 shows this architecture.

The client should be pleased with what is being delivered. But one thing might be bothering you. In earlier testing you found that the monitoring agent correctly notified the mailer when the site was unavailable, but restarting the site and agent required manual work. It would be better if the system tried to automatically recover when a failure was detected. Docker provides restart policies to help deal with that, but you might want something more robust.

2.6 Building durable containers

There are cases where software fails in rare conditions that are temporary in nature. Although it's important to be made aware when these conditions arise, it's usually at least as important to restore the service as quickly as possible. The monitoring system that you built in this chapter is a fine start for keeping system owners aware of problems with a system, but it does nothing to help restore service.

When all the processes in a container have exited, that container will enter the exited state. Remember, a Docker container can be in one of four states:

- Running
- Paused
- Restarting
- Exited (also used if the container has never been started)

A basic strategy for recovering from temporary failures is automatically restarting a process when it exits or fails. Docker provides a few options for monitoring and restarting containers.

2.6.1 Automatically restarting containers

Docker provides this functionality with a restart policy. Using the `--restart` flag at container-creation time, you can tell Docker to do any of the following:

- Never restart (default)
- Attempt to restart when a failure is detected
- Attempt for some predetermined time to restart when a failure is detected
- Always restart the container regardless of the condition

Docker doesn't always attempt to immediately restart a container. If it did, that would cause more problems than it solved. Imagine a container that does nothing but print the time and exit. If that container was configured to always restart and Docker always immediately restarted it, the system would do nothing but restart that container. Instead, Docker uses an exponential backoff strategy for timing restart attempts.

A backoff strategy determines how much time should pass between successive restart attempts. An exponential backoff strategy will do something like double the previous time spent waiting on each successive attempt. For example, if the first time the container needs to be restarted Docker waits 1 second, then on the second attempt it would wait 2 seconds, 4 seconds on the third attempt, 8 on the fourth, and so on. Exponential backoff strategies with low initial wait times are a common service-restoration technique. You can see Docker employ this strategy yourself by building a container that always restarts and simply prints the time:

```
docker run -d --name backoff-detector --restart always busybox date
```

Then after a few seconds use the trailing logs feature to watch it back off and restart:

```
docker logs -f backoff-detector
```

The logs will show all the times it has already been restarted and will wait until the next time it is restarted, print the current time, and then exit. Adding this single flag to the monitoring system and the WordPress containers you've been working on would solve the recovery issue.

The only reason you might not want to adopt this directly is that during backoff periods, the container isn't running. Containers waiting to be restarted are in the restarting state. To demonstrate, try to run another process in the backoff-detector container:

```
docker exec backoff-detector echo Just a Test
```

Running that command should result in an error message:

```
Cannot run exec command ... in container ...: No active container exists
with ID ...
```

That means you can't do anything that requires the container to be in a running state, like execute additional commands in the container. That could be a problem if you need to run diagnostic programs in a broken container. A more complete strategy is to use containers that run init or supervisor processes.

2.6.2 Keeping containers running with supervisor and startup processes

A supervisor process, or init process, is a program that's used to launch and maintain the state of other programs. On a Linux system, PID #1 is an init process. It starts all the other system processes and restarts them in the event that they fail unexpectedly. It's a common practice to use a similar pattern inside containers to start and manage processes.

Using a supervisor process inside your container will keep the container running in the event that the target process—a web server, for example—fails and is restarted. There are several programs that might be used inside a container. The most popular include `init`, `systemd`, `runit`, `upstart`, and `supervisord`. Publishing software that uses these programs is covered in chapter 8. For now, take a look at a container that uses `supervisord`.

A company named Tutum provides software that produces a full LAMP (Linux, Apache, MySQL PHP) stack inside a single container. Containers created this way use `supervisord` to make sure that all the related processes are kept running. Start an example container:

```
docker run -d -p 80:80 --name lamp-test tutum/lamp
```

You can see what processes are running inside this container by using the `docker top` command:

```
docker top lamp-test
```

The top subcommand will show the host PID for each of the processes in the container. You'll see `supervisord`, `mysql`, and `apache` included in the list of running programs. Now that the container is running, you can test the `supervisord` restart functionality by manually stopping one of the processes inside the container.

The problem is that to kill a process inside of a container from within that container, you need to know the PID in the container's PID namespace. To get that list, run the following exec subcommand:

```
docker exec lamp-test ps
```

The process list generated will have listed `apache2` in the CMD column:

```
PID TTY          TIME CMD
  1 ?        00:00:00 supervisord
433 ?        00:00:00 mysqld_safe
835 ?        00:00:00 apache2
842 ?        00:00:00 ps
```

The values in the PID column will be different when you run the command. Find the PID on the row for `apache2` and then insert that for `<PID>` in the following command:

```
docker exec lamp-test kill <PID>
```

Running this command will run the Linux `kill` program inside the lamp-test container and tell the `apache2` process to shut down. When `apache2` stops, the `supervisord` process will log the event and restart the process. The container logs will clearly show these events:

```
...
... exited: apache2 (exit status 0; expected)
... spawned: 'apache2' with pid 820
... success: apache2 entered RUNNING state, process has stayed up for >
      than 1 seconds (startsecs)
```

A common alternative to the use of init or supervisor programs is using a startup script that at least checks the preconditions for successfully starting the contained software. These are sometimes used as the default command for the container. For example, the WordPress containers that you've created start by running a script to validate and set default environment variables before starting the WordPress process. You can view this script by overriding the default command and using a command to view the contents of the startup script:

```
docker run wordpress:4 cat /entrypoint.sh
```

Running that command will result in an error messages like:

```
error: missing WORDPRESS_DB_HOST and MYSQL_PORT_3306_TCP environment
variables
...
```

This failed because even though you set the command to run as `cat /entrypoint.sh`, Docker containers run something called an entrypoint before executing the command. Entrypoints are perfect places to put code that validates the preconditions of a container. Although this is discussed in depth in part 2 of this book, you need to know how to override or specifically set the entrypoint of a container on the command line. Try running the last command again but this time using the `--entrypoint` flag to specify the program to run and using the command section to pass arguments:

```
docker run --entrypoint="cat" \
    wordpress:4 /entrypoint.sh
```

Use "cat" as the entrypoint

Pass /entrypoint.sh as the argument to cat

If you run through the displayed script, you'll see how it validates the environment variables against the dependencies of the software and sets default values. Once the script has validated that WordPress can execute, it will start the requested or default command.

Startup scripts are an important part of building durable containers and can always be combined with Docker restart policies to take advantage of the strengths of each. Because both the MySQL and WordPress containers already use startup scripts, it's appropriate to simply set the restart policy for each in an updated version of the example script.

With that final modification, you've built a complete WordPress site-provisioning system and learned the basics of container management with Docker. It has taken considerable experimentation. Your computer is likely littered with several containers that you no longer need. To reclaim the resources that those containers are using, you need to stop them and remove them from your system.

2.7 Cleaning up

Ease of cleanup is one of the strongest reasons to use containers and Docker. The isolation that containers provide simplifies any steps that you'd have to take to stop processes and remove files. With Docker, the whole cleanup process is reduced to one of a few simple commands. In any cleanup task, you must first identify the container that you want to stop and/or remove. Remember, to list all of the containers on your computer, use the `docker ps` command:

```
docker ps -a
```

Because the containers you created for the examples in this chapter won't be used again, you should be able to safely stop and remove all the listed containers. Make sure you pay attention to the containers you're cleaning up if there are any that you created for your own activities.

All containers use hard drive space to store logs, container metadata, and files that have been written to the container file system. All containers also consume resources in the global namespace like container names and host port mappings. In most cases, containers that will no longer be used should be removed.

To remove a container from your computer, use the `docker rm` command. For example, to delete the stopped container named wp you'd run:

```
docker rm wp
```

You should go through all the containers in the list you generated by running `docker ps -a` and remove all containers that are in the exited state. If you try to remove a container that's running, paused, or restarting, Docker will display a message like the following:

```
Error response from daemon: Conflict, You cannot remove a running container.
     Stop the container before attempting removal or use -f
FATA[0000] Error: failed to remove one or more containers
```

The processes running in a container should be stopped before the files in the container are removed. You can do this with the `docker stop` command or by using the `-f` flag on `docker rm`. The key difference is that when you stop a process using the `-f` flag, Docker sends a `SIG_KILL` signal, which immediately terminates the receiving process. In contrast, using `docker stop` will send a `SIG_HUP` signal. Recipients of `SIG_HUP` have time to perform finalization and cleanup tasks. The `SIG_KILL` signal makes for no such allowances and can result in file corruption or poor network experiences. You can issue a `SIG_KILL` directly to a container using the `docker kill` command. But you should use `docker kill` or `docker rm -f` only if you must stop the container in less than the standard 30-second maximum stop time.

In the future, if you're experimenting with short-lived containers, you can avoid the cleanup burden by specifying --rm on the command. Doing so will automatically remove the container as soon as it enters the exited state. For example, the following command will write a message to the screen in a new BusyBox container, and the container will be removed as soon as it exits:

```
docker run --rm --name auto-exit-test busybox:latest echo Hello World
docker ps -a
```

In this case, you could use either `docker stop` or `docker rm` to properly clean up, or it would be appropriate to use the single-step `docker rm -f` command. You should also use the `-v` flag for reasons that will be covered in chapter 4. The docker CLI makes it is easy to compose a quick cleanup command:

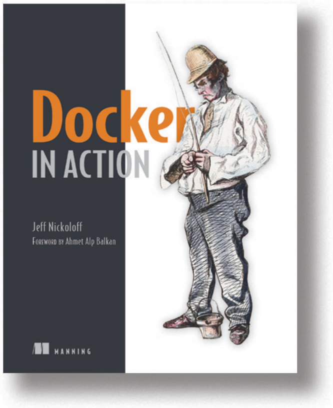
```
docker rm -vf $(docker ps -a -q)
```

This concludes the basics of running software in containers. Each chapter in the remainder of part 1 will focus on a specific aspect of working with containers. The next chapter focuses on installing and uninstalling images, how images relate to containers, and working with container file systems.

2.8 Summary

The primary focus of the Docker project is to enable users to run software in containers. This chapter shows how you can use Docker for that purpose. The ideas and features covered include the following:

- Containers can be run with virtual terminals attached to the user's shell or in detached mode.
- By default, every Docker container has its own PID namespace, isolating process information for each container.
- Docker identifies every container by its generated container ID, abbreviated container ID, or its human-friendly name.
- All containers are in any one of four distinct states: running, paused, restarting, or exited.
- The `docker exec` command can be used to run additional processes inside a running container.
- A user can pass input or provide additional configuration to a process in a container by specifying environment variables at container-creation time.
- Using the `--read-only` flag at container-creation time will mount the container file system as read-only and prevent specialization of the container.
- A container restart policy, set with the `--restart` flag at container-creation time, will help your systems automatically recover in the event of a failure.
- Docker makes cleaning up containers with the `docker rm` command as simple as creating them.



The idea behind Docker is simple. Create a tiny virtual environment, called a container, that holds just your application and its dependencies. The Docker engine uses the host operating system to build and account for these containers. They are easy to install, manage, and remove. Applications running inside containers share resources, making their footprints small.

Docker in Action teaches readers how to create, deploy, and manage applications hosted in Docker containers. After starting with a clear explanation of the Docker model, you will learn how to package applications in containers, including techniques for testing and distributing applications. You will also learn how to run programs securely and how to manage shared resources. Using carefully designed examples, the book teaches you how to orchestrate containers and applications from installation to removal. Along the way, you'll discover techniques for using Docker on systems ranging from dev-and-test machines to full-scale cloud deployments.

What's inside:

- Packaging containers for deployment
- Installing, managing, and removing containers
- Working with Docker images
- Distributing with DockerHub

Readers need only have a working knowledge of the Linux OS. No prior knowledge of Docker is assumed.

index