Author Picks

FREE

Agile Development for Serverless Platforms

by Danilo Poccia

manning

Agile Development for Serverless Platforms

Selected by Danilo Poccia

Manning Author Picks

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 9781617294983
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

Releasing great software requires more than great dev tools; it also demands an efficient operations pipeline that takes advantage of modern Agile practices. Serverless platforms like AWS Lambda implement the basic building blocks you need to run code, store data, or process streaming information so developers can focus on the features they want to provide, not on the underlying infrastructure. Combined with an agile process, serverless architectures facilitate a quick feedback loop between developers, end users, and business stakeholders, allowing the rapid prototyping and easy production rollout required for innovation.

Because serverless architectures start with hosted, self-managed services instead of custom-built servers, they provide a lot of ops advantages when it comes to reliability, scalability, and availability. Web APIs are also key to the serverless mindset, because they enable simple, consistent integrations within and between applications.

This collection of chapters from several Manning books will introduce you to serverless application design using AWS Lambda. You'll also learn about how adopting an agile mindset will give you a leg up when you build and deploy serverless systems.

We hope you'll enjoy it!

Danilo Poccia
Author of *AWS Lambda in Action*

Are You Ready for Agile?

This selection introduces the core ideas of the Agile mindset and helps you assess whether you and your team are ready to go Agile. You'll enjoy Greg and Ahmed's practicality-over-purity approach, which encourages you to adopt the parts of the Agile process that you're ready for and grow into the rest over time.

Are you ready for agile?

Yes, you're ready for agile. The real questions are as follows:

- How much agility are you ready for *today*?
- How much agility can you add *tomorrow*?
- How can you continuously adapt to your ever-changing business climate?

We're confident you can improve your current development process and obtain a level of agility. If your environment is conducive to it, you may be able to reach the level of agility that Archway Software reached in our discussion in chapter 2.

We'll start this chapter by providing information that helps you understand the goals of an agile process and how these goals relate to packaged agile methods such as Extreme Programming (XP) and Scrum. The chapter will conclude by discussing our approach for bringing agile into your workplace. We'll start your migration by

providing a tool that will let you assess your potential for bringing in agile practices and cultural changes.

3.1 What areas will you become more agile in?

When people think of becoming agile, they often envision the practices and not the goals of an agile process. We often hear people say that they can't become agile because their developers don't want to do pair programming, or they have limitations with co-locating their project team members. Although these types of practices may help you become agile, they aren't the only practices that support the *goals* of an agile process. Let's take a moment to look at some of the key agile goals you'll be able to accomplish on some level.

3.1.1 Increasing customer involvement

A traditional process has the customer involved mainly at the beginning and the end of the project. In agile, you seek customer feedback and input throughout the project. The customer or product owner is involved in planning, tradeoff decisions, prioritization, and demonstrations. Increased customer involvement leads to several benefits such as quicker feedback, accurate delivery, increased customer satisfaction, and rapid decisions. A great indirect benefit of customer involvement is the customer's newfound appreciation for the work needed to deliver on requests.

3.1.2 Improving prioritization of features

Agile processes improve prioritization and deliver higher-value features first. This is accomplished by creating feature cards or user stories and evaluating features before requirements are detailed. You'll evaluate features for their customer value, level of risk, frequency of use, and dependencies. This allows you to do the following:

- Estimate work and evaluate risks early in the process.
- Prioritize features in terms of customer value early in the process.
- Deliver features in usable subsets.

In effect, the agile prioritization process lets your team run leaner and create deep requirements only for work that passes the prioritization test.

3.1.3 Increasing team buy-in and involvement

The majority of people on an agile project team are involved in planning, estimating, and sequencing. The team is also involved in adapting to discoveries between iterations. Over time, the team begins suggesting features for the product or platform. Increasing team involvement ensures that everyone understands the value of the project before work begins and also increases team satisfaction.

3.1.4 Clarifying priorities and reminding everyone of the consequences of changing them

An agile team works with the customer and/or sponsor to determine the most critical category for the project. Is schedule the number-one priority, or is staying within

budget? Additional categories may include quality, feature richness, and compliance. The project team learns the priorities and uses this knowledge to make tradeoff decisions along the way.

Many projects wait for a fire before identifying their priorities. An agile team knows the project priorities in advance of an emergency and can react quickly to keep the focus on the main objective.

3.1.5 Adapting to change during development

A more agile and iterative methodology provides an opportunity to reassess and redirect the project while it's in motion. You perform development in iterations and offer demonstrations at the end of each. The customer has an opportunity to request changes based on the demonstrations, even though this may affect other features or potentially the project timeline. Team members learn to expect and embrace change.

3.1.6 Better understanding the project's status

Agile development is time-boxed. You evaluate status by demonstrating functioning code. Supporting tasks are also measured in binary terms (done or not done) to eliminate possible confusion related to expressing status as "percent complete." An agile process also involves team members reporting their status themselves versus through a manager or other intermediary. This improves tracking accuracy and personal accountability.

3.1.7 More efficient planning and estimating

Many companies try to plan all of a project's details at the start. The planning may be at a detailed level even though the amount of uncertainty at this point is extremely high. An agile team performs a level of planning that correlates to the current level of uncertainty in the project.

As you learn more about desired features you'll do more detailed planning, but you won't waste time trying to guess intricate details early in the project. Figure 3.1 illustrates this point.

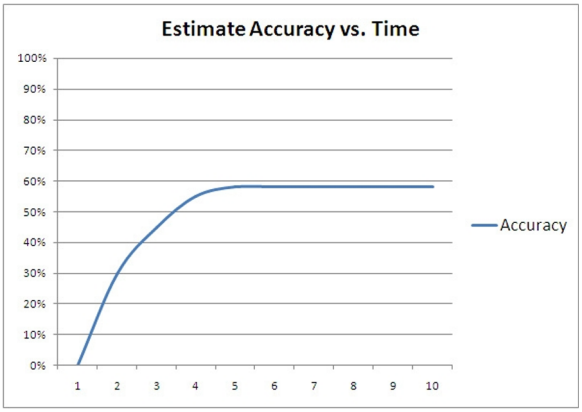


Figure 3.1 The accuracy of initial feature estimates improves dramatically during the first few hours of estimation but levels out over time. In this example, the effort and time spent after five hours of estimating doesn't improve accuracy and is wasted project time.

3.1.8 Continuous risk management

A secondary definition of *agile* could be *continuous risk management*. The processes are all intended to make the team alert and responsive to new information and changes as the project progresses. The following are a few examples of how agile manages risk:

- Features are evaluated for requirements uncertainty and technical uncertainty. These attributes help determine whether a feature goes into an iteration and what iteration it should go into, to mitigate risk. For example, a feature with high business value and high technical risk, such as an interface, would go into an early iteration to allow more time for uncertainty. On the other hand, a feature with low business value and high technical uncertainty might be moved to the last iteration or removed from the project all together.
- Risk is managed via demonstrations throughout the project. The customer gets a feel for how requirements are translated into an application before the project is complete. This provides a window for adapting and hitting the final target.
- Risk is managed on a daily basis by building and integrating the latest code. This process allows the team and the customer to validate the status of the latest build.
- Deployment risk is also managed by gathering maintenance and deployment concerns as early as possible. This starts early in the planning phase and continues throughout development.
- Risk is managed via team review of potential features. During the feature-card exercise, representatives from all areas can raise risks and concerns with proposed features. These concerns are noted with the feature information and sometimes can lead to a feature not being pursued.

3.1.9 Delivering the project needed at the end

Jim Highsmith, one of the founding members of the Agile Alliance, taught Adaptive Software Development a few years before the Agile Manifesto was created. One of Jim's adaptive principles is, "Deliver the project needed at the end, not the one requested at the beginning."

This idea is a foundational piece of agile software development. Jim knew the world wasn't static during the project lifecycle; therefore the lifecycle should support changes that happen during the project. This includes identifying new requirements, discovering technical risks, and identifying potential changes in the business environment.

3.1.10 Achieving the right level of project structure

Many companies have created a formal Project Management or Software Development Lifecycle (PMLC/SDLC) to support their projects. These lifecycles are collections of processes that every project must follow. By establishing required processes, companies eliminate variation between projects and provided a safety net for inexperienced

project teams and project managers. If you don't know what to do next, you just look at the lifecycle documentation to determine your next step.

This approach is beneficial when you have inexperienced employees. A standardized process defines roles, provides common tools, and offers gateways to evaluate status.

If your employees are more experienced, this formal methodology has drawbacks. The team will notice that every step or process isn't needed for their specific project. They will frequently find themselves doing compliance work that adds no value, except to be in compliance.

The agile process described in this book approaches the issue differently. We suggest a standardized methodology, but the required processes are minimal and are of value to every project. Your team chooses the majority of the processes to use at the start of the project. The team also revisits their process and documentation options as the project proceeds, to see if they need to add or remove a process or document.

To illustrate this idea, let's look at an example from Acme Media *after* the company has outlined a new, more agile process (see table 3.1).

Acme Media has projects that last from 1 week to 6 months. The company doesn't require the teams for one-week projects to create iteration plans or to do a cost-benefit analysis every time.

Table 3.1 Required and optional processes and documentation

Required for all projects	Optional processes and documents
Project worksheet	Elevator statement
Operational worksheet	Documented answers to feasibility discussion guide questions
Feature-card exercise (cards optional)	Feature-card document (possibly created using only index cards)
Retrospective discussion	User scenarios
	Prototypes and/or mockups
	Iteration plan
	Maintenance plan
	Evolutionary requirements
	Additional documentation as required by the team/project
	Test plan
	Detailed schedule
	Launch plan
	Action items from project retrospective
	Test Driven Development (TDD)
	Agile estimating
	Daily stand-up meeting
	Demonstrations

These one-week projects are frequently driven by a need to increase readership or to provide support in the aftermath of a major news event such as an election. Executive approval is almost immediate, and the projects use team members already assigned to the website. These teams only need the processes and documents outlined in the first column of table 3.1.

Conversely, Acme Media pursues some major projects that require funding, synchronization with third parties, and identification of milestones. In these instances, the project teams review the items in the second column of table 3.1 and decide which ones to use in addition to the required ones in the first column.

In this way, agile provides the correct amount of structure for the project. Time isn't wasted on processes that don't add value, and teams can scale their processes mid-project if needed.

Now that you understand the goals of an agile process, you need to know the best way to obtain them. You can do this by selecting a prepackaged agile process, creating a process from scratch, or a combination of the two. Let's evaluate each option.

3.2 The different flavors of agile

Many packaged methods are available for agile. For our purposes, *packaged* will mean a framework with a common set of practices. In this section, we'll discuss two of the most popular packages in use today: Scrum and XP. According to VersionOne's 2008 "State of Agile Development" survey, 77 percent of the respondents said they use Scrum, XP, or a Scrum/XP hybrid. Each of these packages has its own unique characteristics, strengths, and weaknesses. Let's examine each package.

3.2.1 Scrum

The Scrum process begins by reviewing a product backlog with the product owner. You identify the highest-priority features and then estimate how many will fit into a sprint. These features then compose the sprint backlog. A *sprint* is a predefined period of time, usually 2 to 4 weeks, during which the team analyzes, designs, constructs, tests, and documents the selected features. Figure 3.2 shows an overview of the process.

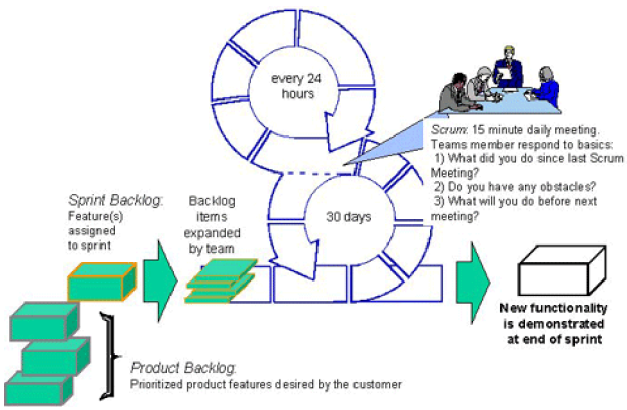


Figure 3.2 A high-level overview of the Scrum process (graphic provided courtesy of Ken Schwaber and Control Chaos)

The team holds a daily status meeting, referred to as the daily *Scrum*, to review feature status. Individual team members answer these three questions:

- What have you accomplished since our last meeting?
- What will you work on today?
- Are you encountering any impediments or roadblocks in completing your work?

When a sprint is completed, the features are demonstrated to the customer, and the team and the customer decide whether additional work is needed or if the sprint work is approved to be released to a beta or production environment. Each sprint is followed by a retrospective during which the team lists items that went well or poorly; action plans are documented to keep the successes going and to improve the areas that performed poorly.

Some of the characteristics of Scrum are as follows:

- *Discipline*—Scrum is strict about time-boxing activities, compiling code daily, and team members being punctual and responsible.
- *Three major roles*—Scrum teams have a ScrumMaster, a product owner, and team members.
- *Quality*—Features are expected to be totally complete and deployable at the end of a sprint.

Scrum has a number of strengths:

- *Prioritized delivery*—Features are delivered in a sequence that ties to business value.
- *Non-prescriptive on practices performed during a sprint*—This is demonstrated by the fact that a Scrum/XP hybrid is the second most popular agile methodology in use. Many teams pull their detailed practices from XP while using the Scrum framework.
- *Demonstrated success across the software industry*—Scrum has been successful in multiple environments.
- *Status transparency*—The daily meetings expose the project status.
- *Team accountability*—Everyone signs off on the work that will be pursued during the sprint.
- *Continuous delivery*—Scrum delivers product features (commercial software or web portals) continuously.

Scrum also has some weaknesses:

- Scrum doesn't want specialists. It may be difficult to quickly convert an existing team from a group of specialists to a group where anyone can perform any task.
- A Scrum team can't be successful without a strong ScrumMaster, which makes the process highly dependent on one individual.
- Because Scrum is mainly a framework, the team still needs to identify the practices and methods to use within the framework.

Scrum is incredibly popular today—it's almost become synonymous with the term *agile development*. Scrum provides a great, repeatable process that is well suited for product development and steady-state release management. In addition, a plethora of books, consultants, and other resources are available for those who pursue Scrum.

Scrum may be more difficult to use with teams that do one-off projects versus steady-state releases, or if a team has highly specialized resources and skill sets. In addition, the Scrum framework still needs agile practices inserted to support a complete development lifecycle.

3.2.2 Extreme Programming

Similar to Scrum, XP starts the process by creating a backlog of work to perform during a sprint/iteration. XP creates the backlog by working with customers and creating user stories. In parallel with this work, the team performs an architectural *spike*, during which they experiment with the features to envision the initial architecture. XP classifies this work as the *exploration phase*.

The *planning phase* follows exploration. This phase focuses on identifying the most critical user stories and estimating when they can be implemented. Tasks are defined for each feature, to aid with estimating complexity. The team outlines an overall release schedule, with an understanding that a high level of uncertainty exists until the work begins. A release will have one to many iterations, which are typically 2- to 4-week construction windows.

When an iteration begins, the specific plan for the iteration is revisited. The team adds any new user stories and tasks that have been discovered since the overall release was outlined.

XP integrates customer testing into the development iteration. The customer is asked to identify the acceptance tests, and the team works to automate these tests so they can be run throughout the iteration.

The planning phase is followed by the *productionizing phase*, during which the code is certified for release. *Certified* means the code passes all customer tests plus nonfunctional requirements such as load testing, service-level requirements, and response-time requirements. You can see an overview of XP in figure 3.3.

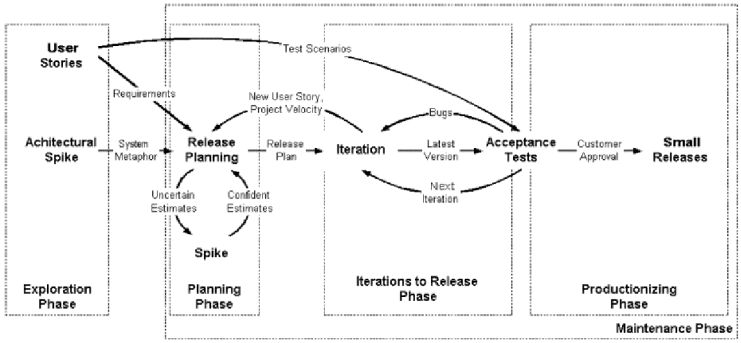


Figure 3.3 The Extreme Programming (XP) lifecycle (graphic provided with permission from Scott Ambler, based on the writings of Don Wells and the first edition of Kent Beck's XP Explained)

Some of the characteristics of XP are as follows:

- *Specific practice*—Unlike Scrum, XP is specific about the practices that should be used during a software project. These practices include pair programming, TDD, continuous integration, refactoring, and collective code ownership.
- *Modeling*—XP teams frequently use modeling to better understand the tasks and architecture needed to support a user story.
- *Simplicity*—Teams perform the minimum work needed to meet requirements.
- *Automation*—Unit and functional tests are automated.
- *Quality through testing*—Features are tested constantly, and developers check each other's code via pair programming.

These are some of XP's strengths :

- Customer-focused (it's all about user stories)
- Quality via frequent testing
- Constant focus on identifying and delivering the critical user stories
- High visibility on project status
- Great support for volatile requirements

It also has weaknesses:

- *Need for team maturity*—Practices such as pair programming and TDD require responsible developers, and they aren't always easy to obtain.
- *Dependency on testing*—If developers know that significant testing will take place downstream, they may be less than diligent when they're creating designs.
- *Scalability*—XP may not work well for large projects.
- *Dependency on team member co-location*—The team usually has a team room.

XP supports many of the critical goals of an agile process, such as dealing with volatile requirements and delivering prioritized, working software as soon as possible. XP also supports the principle of *just enough*, keeping with the lean philosophy of minimizing waste.

XP has sometimes been criticized for its lack of formality in system documentation and system design. In recent years this has changed, and XP teams now create the documentation needed to support a project's customers.

3.3 Create your own flavor to become agile within your constraints

As we discussed in chapter 1, VersionOne's 2007 "State of Agile Development" survey validated the benefits of using agile. If the survey is accurate, then should every company migrate to agile methods tomorrow?

We're huge proponents of agile, but we need to tell you a few things that the surveys don't reveal. Here are some questions that would bring additional perspective to VersionOne's findings.

- How difficult was it to convert to an agile development process?
- How was your conversion initiated? Did the idea originate with executive management or from within the development team?
- Have your employees bought into the process, or was it forced on them?
- What are you doing to ensure that your development process is viable for the future?
- What did you do to make agile work within the realities of your environment?

We believe 100 percent of the survey respondents would say that moving to agile was a lot of work. We think they would tell you that to be successful, you need your project team to buy into the process; and that management requires time to learn how to provide value in an agile environment.

This discussion reminds us of a popular commercial from our childhood. When we were kids, we ate Jiffy Pop popcorn. Jiffy Pop ran a commercial for many years that stated, "Jiffy Pop: it's as much fun to make as it is to eat!"

After you establish an agile culture and lifecycle, it's "fun to eat" (as illustrated in figure 3.4), and you'll do a better job of delivering projects. But creating an agile environment is work. Many companies implement an agile methodology and then fade back into their previous process because they didn't cover all the delicate areas needed to ensure long-term support for agile.

We've spent a lot of time with companies that have made it to the other side and stayed there. As this book continues, we'll show you how companies got to be agile with the least amount of pain and sustainable benefits.

Now let's take a moment to look at the importance of creating an agile process that supports the unique characteristics of your environment.

Figure 3.4 Is agile development like Jiffy Pop popcorn—as much fun to make as it is to eat? Not during the migration phase. Managers need to learn when to manage (or not), and team members need to experiment with their new freedoms. These cultural changes take work and time.

3.3.1 Your goal: reach the right level of agility for your organization

Many companies try to "shotgun" agile into their organization. They think, "Let's get through the migration pain quickly and start obtaining the benefits as soon as possible." We've seen a few cases where this approach makes sense: for example, a project team that has become so dysfunctional that they're delivering practically no functionality or business value. This approach also works well for a start-up company that hasn't yet established its development process. But for most companies, you should allow time for the process to "bake."

This is why we suggest an iterative approach for bringing agile into an organization. An iterative approach allows you to see how well your employees are adapting to

the change. It also lets you learn what works and what doesn't in your environment. In effect, it allows you to reach the right level of agility for your organization.

Part of your iterative approach will include a process for maintaining the methodology. We suggest establishing a core team to support this maintenance. A core team is composed of employees from all aspects of the development process. They play a huge part in establishing your custom methodology and then settle into a maintenance mode with the goal of constantly adapting to your environment. The core team is covered in detail in chapter 6.

Next, you need to choose the best way to iteratively create a methodology at your company. Should you select a packaged method, such as Scrum or XP? Or should you create a custom or hybrid process?

CUSTOM PROCESS OR PACKAGED METHOD?

In order to be successful, you should customize your agile process. For many years, consultants and others have said that you must embrace agile completely or not at all.

In 2006, we witnessed a shift in this attitude. Highly respected folks such as Kent Beck (the founder of XP) and Steve McConnell (the writer of *Code Complete*) now endorse customization. Kent Beck noted the following in an interview with InfoQ (InfoQ.com is an independent online community focused on change and innovation in enterprise software development) in 2006:

> *Failure at an organizational level seems to come from the inability to customize processes and make them their own. Trying to apply someone else's template to your organization directly doesn't work well. It leaves out too many important details of the previous successes and ignores your company's specific situation. Rubber-stamping agile processes isn't agile. The value of having a principle-based process is that you can apply the principles for an individualized process for your situation and, as an extra bonus, one that has been designed to adapt from your learning as you adopt changes into your organization. It's always "custom."*

Kent's quote is comforting to us because it supports our personal experiences. *Custom* means picking and choosing the agile practices that best support your environment. *Custom* means you shouldn't use a pure packaged methodology off the shelf, such as Scrum or XP. You can start with one of these methods as a basis for your process, but you should modify it to obtain the best results for your company.

If we revisit VersionOne's 2008 survey, we see that 14 percent of the people who responded are using a hybrid process based on Scrum and XP. The hybrid model is closer to what we'll suggest for you. To be specific, here are the steps we'll walk you through as the book continues:

1. Assess your organization to determine where you should begin adding agility.
2. Obtain executive support for the move to a more agile process. You can use the readiness assessment in chapter 4 to quantify the value of bringing in agile and identify the risks you must manage during migration.
3. Get the development team involved in the migration process to ensure buy-in. You do this by establishing a core team.

4. Identify a coach or consultant to help you with your migration. They will train the core team on agile and help you with other adoption aspects.
5. Develop a clear understanding of your current processes by documenting them.
6. Review your current process, and look for areas that can be shifted to more agile methods. Focus on areas with the most potential for improvement and the most value to the customer and your organization. The readiness assessment will also help with this task.
7. Outline a custom process based on the findings in step 6.
8. Try the new process on a pilot project.
9. Review the findings after the pilot, make changes, and continue to scale out your new methodology.

As this book continues, our case study, Acme Media, will represent your company. We'll take Acme through these nine steps and show you how the company iteratively creates and tests a custom process. We'll also show you how Acme Media takes its own constraints into account with the new methodology.

Before we jump into the case study, let's spend a moment looking at the characteristics that make it easier to adopt agile and the characteristics that make agile adoption more challenging.

3.3.2 Characteristics that make agile easier to adopt

As we stated earlier in this chapter, agile principles can be applied in any environment, but some environmental characteristics influence how easy the principles are to adopt. Let's look at these characteristics.

Urgency to deliver

Agile works best in an urgent environment. It provides tools to prioritize features quickly and determine how much scope to pursue within the constraints of a critical timeline. If you have urgency due to a competitive market, compliance deadlines, or a large backlog of project requests, agile provides methods for quicker delivery.

Evolving or volatile requirements

One descriptor of agile could be *just enough*. "Give me just enough requirements to start a design." "Give me just enough design to start my code." "Give me just enough code to demonstrate some level of value to the customer." If you don't have all the requirements, you can still get started with an agile project. If you complete an iteration and the customer wants to change the requirements, you can adapt and still meet the objectives. Managing changing requirements still takes effort in an agile environment, but you don't have to fight the project framework. The framework is designed to support uncertainty.

Customer availability

One Agile Manifesto principle states, "Business people and developers must work together daily throughout the project." In our experience, these groups don't have to

work together every day throughout a project cycle, but there are definite times when the customer must be available. In theory, a project must not be urgent if the customer can't make time to clarify requirements or review functionality. The customer can have a proxy, such as a product manager; but someone needs to be available every day to represent the customer's vision.

CONSISTENT RESOURCES

Part of the power of agile is a level of familiarity within the team and a consistent understanding of the processes they use. Agile teams and processes get better over time. If project team members are new to each other, they must learn processes together while at the same time trying to complete the project. Agile works best with a core group of people who work together on continuous projects. Agile isn't a good methodology to use with a team that has never worked together before, unless you have long-term plans to keep them together.

CO-LOCATED RESOURCES

Agile promotes face-to-face communication and common understanding. One of the best ways to support this principle is to put your team members face to face. Co-location is an amazing tool. Your team can get out of *email hell,* and their mutual understanding of the project will increase.

One of the best setups we have seen is at a Fortune 500 company we visited. All 10 of the project team members are in an area approximately 25 feet by 25 feet. The cubicles have half-walls that provided a level of privacy when people are sitting but let them easily see the rest of the team and communicate when they stand up. This setup provides the privacy the developers enjoy when they're deep into a coding session but also lets team members stand up to converse with each other at any time without having to go to each others' cubicles. Team members can also walk a few feet and reach common areas where they can whiteboard a design or have a quick caucus.

THE TEAM IS A TEAM

In larger companies, a project team may be constructed of team members from a shared resource pool. For example, the QA (Quality Assurance) lead for a project may be from the QA shared resources pool. If such team members view themselves as resources on loan, and not as team members dedicated to the project, the result can be functional silos.

When silos exist, team members are more concerned about the welfare of their team or area than they are with the livelihood of the project. This mentality doesn't bode well for agile development and leads to customer neglect. The team needs to bond as a unified group toward the goals of the project. Roles are assigned, but one of the objectives of agile is for the team to working collectively.

Working collectively can also be applied to team member roles. A tester can point out a possible code improvement. A developer can suggest a feature enhancement. In general, team members *speak out*—they don't limit their roles to their titles.

Management should ensure that individual goals include how well employees support the common good of the project.

3.3.3 Roadblocks that others have overcome

Now that you know the characteristics that make agile easier to implement, let's look at a few that make agile more difficult to move to.

Lack of agile knowledge

Your first challenge will be finding expertise to help you with your migration. If you're fortunate, you'll have some level of agile experience within your company; but this probably won't be true to the point that you can coach yourself through an agile migration.

We'll help you with this issue by showing you how often Acme Media requested assistance, from initial training to issues encountered along the way.

Large project teams

Agile is compromised as team size increases. Major principles such as face-to-face communication and common understanding require additional effort to maintain their effectiveness as a team grows.

Larger teams require additional overhead to ensure that information is shared consistently across all groups. Scrum teams frequently use the term *scrum of scrums*, meaning a representative from each team Scrum attends a master Scrum meeting to share information with other groups.

Jeff Bezos of Amazon.com believes that the most productive and innovative teams can be "fed with two pizzas." Jeff shared this thought with his senior managers at an offsite retreat. He envisioned a company culture of small teams that could work independently, which would lead to more innovative products. Since that time, the Amazon "pizza teams" have created some of the most popular features on the site (Fast Company, 2004).

If your team has an average appetite, you can convert Jeff's concept into a team of five to seven people. This is a nice-size group for communication and agility. If five to seven is perfect, then what is the maximum size for a team to remain agile? On the high side, we believe you can have a team of 15 people without major impact on your agility. When you have more than 15, communication needs to become more formal, which slows the team.

There are ways to make agile work with larger or distributed teams, but you'll sacrifice some level of agility.

Distributed development

Related to large teams, many companies use distributed development. Frequently, the distributed development is performed by offshore resources.

Distributed development implies that the team is large in size and that communication methods must be scaled to get information to all involved. In addition, you may have issues with time zone differences, language, and code integration into a common environment. Some offshore companies support and advertise the use of agile methodologies, but their location may make it challenging to support the core principles.

We've seen agile teams successfully use offshore resources for commodity or repeatable-type work, such as regression testing, smoke testing, and cookie-cutter

development (for example, providing an offshore group with standardized tools to create automated workflows).

FIXED-BID CONTRACT WORK

Fixed-bid contract work goes against most of the agile principles. The customer isn't a partner, evolving requirements are a no-no, and adapting is usually called *scope creep*. We used to believe that fixed-bid work couldn't be performed using an agile process, but recently we've met several managers who have customized their process to allow the inner workings to be agile while customer interaction remained contract oriented.

AN IMMATURE OR ONE-TIME TEAM

If you have a team that will work together for only one project, they're usually better served by using a plan-driven methodology unless they have previous exposure to agile. If the team will work through multiple projects or releases, you can introduce agile techniques, and the team can migrate to a full agile methodology as their knowledge matures.

GOING TOO FAST

"Hey, it's agile. We don't need to do any planning to convert to it, just start thinking agile!" A lot of folks take this approach when migrating to agile. But if you go too fast, you don't give your company enough time to digest the concepts. When this happens, you may experience issues with common understanding and terminology.

Don't let this happen to you. You need to plan before migrating to agile, and this book will show you how to do it with an *awareness, buy-in, ownership* approach. If you take your time, the methodology will stick, and you'll minimize the risk of failure. You'll learn more about ownership in chapter 5.

TEAM WITH SPECIALIZED SKILL SETS

An organization's structure can create artificial barriers between teams, and so can skill sets. If your team has specialized skill sets, it's hard to be agile when the work mix doesn't correlate well to the available resource types. Some tasks always have to be done by certain individuals, which doesn't help the team bond or unite when pursuing the completion of a feature.

Specialized skill sets also place an additional constraint on team capacity. Imagine that your team has only one person who can perform user-interface design, and the work assigned to an iteration is 80 percent user-interface work. Other team members can look for work to do outside of the iteration, but delivery will be slow due to the one-person constraint.

Teams that are just becoming agile usually have members with specialized roles. You can overcome this constraint by cross training over time and rewarding employees for obtaining and using additional skills.

AVOIDING CUSTOMIZATION

Many people get hung up on the questions, "Are we doing it right? Are we doing it in an agile fashion? Are we following a pure agile process?"

When teams ask us these questions, we tell them the answers aren't important. All we want to know is this: Have you created a development process that provides the most benefit to your company?

This same mentality has managers trying to find a perfect agile methodology and insert it directly into their company. As we discussed earlier, you can start with a packaged agile process, but you need to look at the realities of your company and adjust accordingly. Acme Media will look at a generic agile process and see how it applies to their realities; then, they'll modify the process to fit their environment.

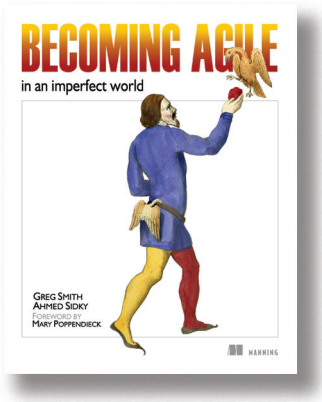
3.4 Key points to remember

The key points to remember from this chapter are as follows:

- Moving to agile isn't a one-time event. You can and will add agility over time.
- The goals of an agile process tie directly to company success.
- You can start with a prepackaged agile process such as Scrum and then modify and enrich the process to support the realities of your environment.
- Some of your existing company characteristics will make it easier to move to agile. This is especially true if you have volatile requirements or urgency to deliver frequently.
- Every migration to agile encounters roadblocks. We'll identify the most common roadblocks and show you how others have addressed them.
- Every migration to agile is unique, but we believe our nine-step framework will work for most companies and provide the best chance of moving to and sustaining an agile process.

3.5 Looking ahead

In this chapter, you've learned that the question isn't whether you're ready to become agile, but rather what level of agility you're ready for today. In chapter 4, we'll help you answer this question by discussing the use of assessment tools to determine which agile practices you can initially adopt with minimum risk. Assessing your current potential is also important for gaining executive support, which we'll cover in chapter 5.



Many books discuss Agile from a theoretical or academic perspective. *Becoming Agile* takes a different approach and focuses on explaining Agile from a ground-level point-of-view. Author Greg Smith, a certified ScrumMaster with dozens of Agile projects under his belt, presents Agile principles in the context of a case study that flows throughout the book.

Becoming Agile focuses on the importance of adapting Agile principles to the realities of your environment. While Agile purists have often discouraged a "partial-Agile" approach, the reality is that in many shops a "purist" approach simply isn't a viable option. Over the last few years, Agile authorities have begun to discover that the best deployments of Agile are often customized to the specific situation of a given company.

As well, *Becoming Agile* addresses the cultural realities of deploying Agile and how to deal with the needs of executives, managers, and the development team during migration. The author discusses employee motivation and establishing incentives that reward support of Agile techniques.

Becoming Agile will show you how to create a custom Agile process that supports the realities of your environment. The process will minimize risk as you transition to Agile iteratively, allowing time for your culture and processes to acclimate to Agile principles.

What's inside

- How to migrate to Agile
- How to get your team to buy into the change
- How to scale and sustain your new Agile process
- How to create an Agile process that works for your company
- How to use Agile in special situations
- How to iteratively build up your Agile process and culture

Working with Web APIs

In this chapter, you'll get an overview of how Web APIs work, along with some guidance on how to design APIs that your developers and external API consumers will find refreshingly clear and easy to use.

Chapter 2 from *Irresistible APIs* by Kirsten Hunter

Working with web APIs

This chapter covers

- Structure of a simple API
- Ways to inspect calls to an API
- Interaction between an API and a client application
- Deployment of the sample API and application on your system

The next few chapters cover the server-client interaction in detail, but this chapter helps you understand the concepts with a simple example of an API and sample application. Most basic API examples use a to-do list, but that's kind of overused. I decided to go a different way: I've selected a list application with pizza toppings. Note that this particular application is simple by design; the goal is to show you how to interact with the API, and how an application interacts with an API. If this were a production application it would have a full pizza, or pizzas, and the database wouldn't be shared, but for the goals here I've taken out as much complexity as possible to make the basic principles clear.

Looking at an API is interesting, but it doesn't necessarily help you to understand how it can drive an application. Additionally, performing actions such as create and delete in a browser is challenging, so in addition to the API I've included a simple application using this API with JavaScript. This application exercises all the functionality in the API so you can see how an application interacts with a web API.

To get an idea of how this works in practice, I've created a basic API using Node.js, a JavaScript-based web server framework. (You can learn more about this framework at www.nodejs.org.) The API supports all the needed actions to represent a complete system: create, read, update, and delete. The first task will be to explore the API in a browser using the read functionality.

This application runs on a web host at www.irresistibleapis.com/demo. You can check out the application there and follow along with the concepts in this chapter. If you're a developer and want to explore the code more intimately, use the exercises at the end of the chapter to get the example running on your own system, including both the Node.js application and the HTML/JavaScript web application. Section 2.6 also describes the various moving parts to this API and application so you can play with it as you like.

2.1 HTTP basics

To understand the transactions between the client and the server in API interactions, you'll need a basic grasp of how HTTP works. Chapter 4 covers this topic in more detail, but for now I'll give you some high-level information about the protocol.

You're probably most familiar with HTTP as the way web browsers get information from web servers. An HTTP transaction is composed of a request from the client to the server (like a browser asking for a web page), and a response from the server back to the client (the web page from the server, for a browser). First, I'll describe the elements in an HTTP request. You're familiar with the URL, the address that you type into the address box on a browser, but that address is only one portion of the information sent from your browser to the server in order to process a web request.

2.1.1 HTTP request

Figure 2.1 illustrates the elements that make up an HTTP request, along with examples of how these sections are used. The HTTP request is usually sent with headers, setting the context for the transaction. An HTTP request always has a method; methods are the verbs of the HTTP protocol. To understand what your browser does, imagine that you're visiting my main website. Here are the pieces of the request that are sent by your browser:

- *Headers: Accept: text/html*—This tells the server that the browser wants to get an HTML-formatted page back. It's the most readable format for humans, so it makes sense that your browser would request it.
- *Method:* `GET`—This is the read method in HTTP and is generally the method used by browsers when reading web pages.
- *URL: http://irresistibleapis.com*—This is the only piece you indicated for the browser.
- *Body: none*—A GET request doesn't need a body, because you're not changing anything on the server—you're reading what's there.

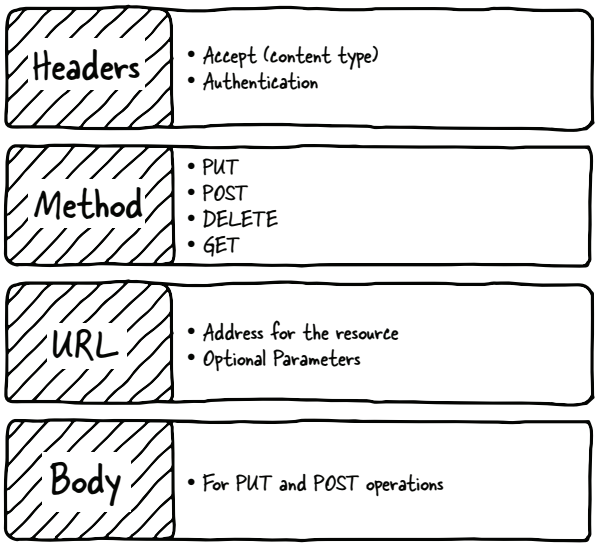


Figure 2.1 An HTTP request will always have a method and will be sent to a specific URL, or resource. Depending on the specific call, headers may be sent to specify information about the request. If the call is designed to write new information to the system, a body will be sent to convey that information.

All the actions of CRUD (create, read, update, and delete) are represented by methods within HTTP:

- Create: `POST`
- Read: `GET`
- Update: `PUT`
- Delete: `DELETE`

The URL is the unique identifier for the resource. It's like any other URL on the internet, except in this case it's used to describe the resource in an application system. If parameters are needed for the request, such as a keyword for search, they're included in the parameters of the request. To see how parameters would look, here's an example search request:

```
http://www.example.com/api/v1.0/search?keyword=flintstone&sort=alphabetical
```

In this example, the resource being called is http://www.example.com/api/v1.0/search. The question mark and everything following it are parameters giving more information about what the client wants in the response. A body section is only sent for create (`POST`) and update (`PUT`) transactions.

Next, I'll describe the sections of an HTTP response.

2.1.2 HTTP response

Figure 2.2 shows the elements of a typical HTTP server response. The server is likely to send back several headers giving information about the system and the response. All requests have a method, and all responses have a status code. These status codes are described in more detail in chapter 4, but for now it's sufficient to know that 2XX means that the request was successful, 3XX is a redirect to another location, 4XX is an

error in the request from the client, and 5XX means the server had a problem. In the earlier example, calling my website, the server would've responded with the following:

- *Status code: 200*—Everything worked correctly.
- *Headers:*
  - *Content-Type: text/html*—as requested by the client
  - Date: <date of response>
  - Content-Length: <size of response>
- *Body*—The content of the page. This is what you see if you "view source" within the browser—the HTML page that tells the browser how to render the page and what content to display.

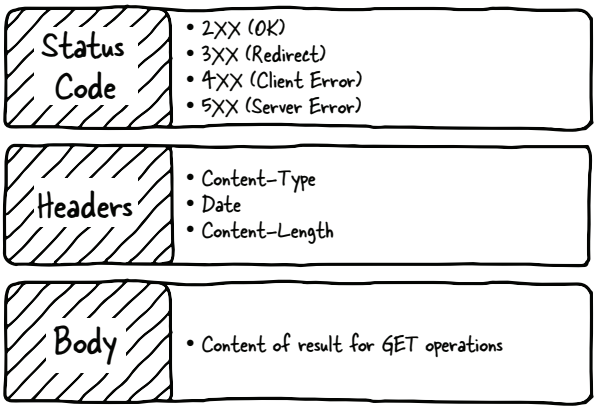


Figure 2.2 A response will always have a status code, and a well-designed platform will send headers to provide information about the response (such as size or the content type). For most requests, a body will be sent back from the server to provide information about the current status of the resource.

2.1.3 HTTP interactions

Every HTTP transaction between a client and server is composed of a request, sent from the client to the server, and a response, sent from the server back to the client. There's no higher level interaction; each request/response is stateless and starts again from scratch. To help you understand this better, I'll move on to a discussion of a specific API.

2.2 The Toppings API

Many different styles of API are available, but the one I'm going to be using and talking most about here is a Representational State Transfer (REST)-style API, the most common type of web API.

As discussed in chapter 1, REST APIs are designed to work with each resource as a noun. A specific resource within a system has a unique identifier, which is a URL, like the ones you visit in the browser. This URL identifies the resource in the system and is designed to be understandable when viewed. For example, with a REST API you could view the list of existing toppings with the following request:

```
http://irresistibleapis.com/api/v1.0/toppings
```

These are the actual URLs, retrieved with a `GET` (read) operation. If you put the preceding URL in a browser, you'll see the results displayed in figure 2.3.

You can visit this URL in your browser right now and get the information about a single topping or a list of toppings. Figure 2.3 shows what this call will look like in a web browser. Go ahead and try both of these calls in your own web browser to see how easy it is to retrieve information from this kind of service. Again, this is like any other web request, only formatted for a computer to work with.

Now, to view a single topping, you'd take the `id` field from the list you retrieved and append it to the URL. Basically, you're saying, "Give me the toppings list" and then, "but just the one with the ID of 1." Almost all APIs work this way. The parent level is a list of items, and adding an ID will retrieve a single member of the list.

```
http://irresistibleapis.com/api/v1.0/toppings/1
```

The same resource is accessed to update, view, or delete a particular item, using different HTTP methods (as described in section 2.1) to tell the server what you want to do. You can add new items by sending a `POST` to the list itself (so in the earlier case, the `/toppings` endpoint would be used to add a new topping). This type of API encourages engagement and innovation by the developers, and consistency across multiple API providers makes it easier to get up and going writing clients.

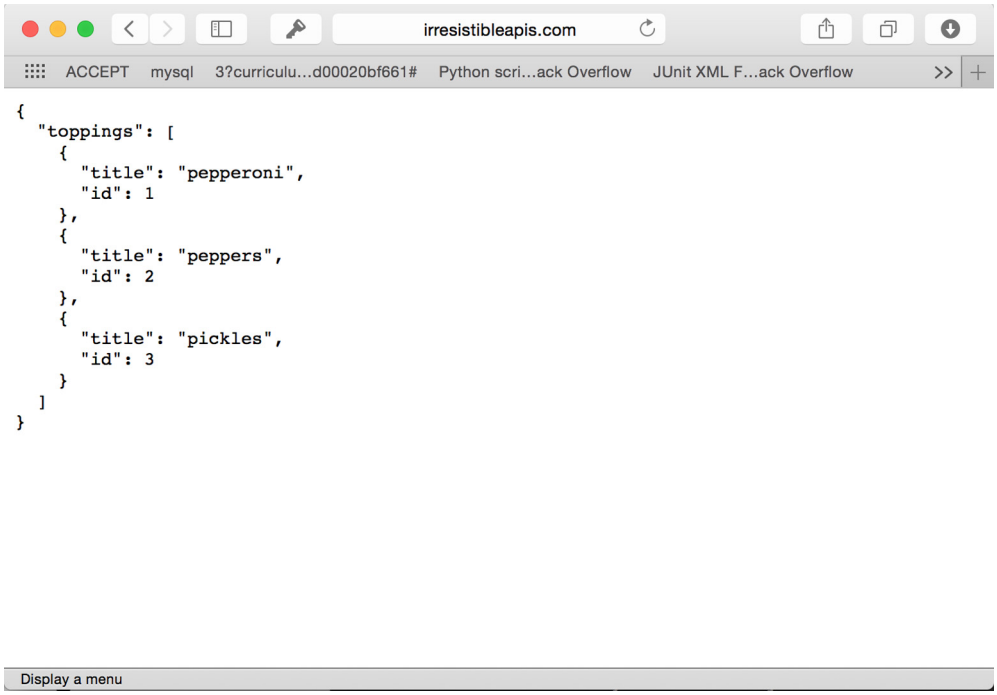


Figure 2.3 Example result of a web call in a browser. The response is JSON, a common markup language for web APIs. As you can see, the formatting makes it easy to understand the content of the response.

2.3 Designing the API

To go through the steps, imagine an online website for a pizza parlor. Users are having trouble interfacing with the pizza ordering system and want to be able to customize their pizzas. The company wants to increase customer satisfaction. This represents the business value for this platform. Figure 2.4 illustrates each call to the system and how it would be formatted.

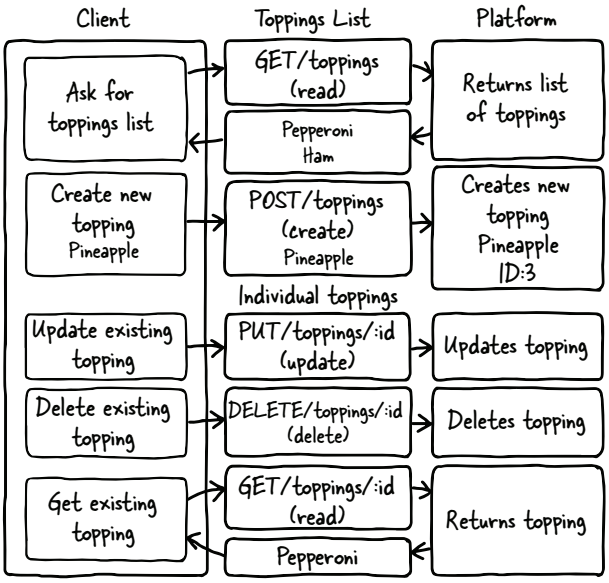


Figure 2.4 This diagram represents the complete set of interactions with the API system. The `GET` request reads the current value of the resource, whether it's a list or an individual item. `POST` is only allowed at the list level, and creates a new resource in the system. `PUT` updates and `DELETE` deletes an existing resource. All four of the needed methods, `Create`, `Read`, `Update`, and `Delete`, are represented in this diagram.

To provide this, they need to create a system that consistently allows users to pick different pizza toppings and keep them in a list (use case). The company decides to measure success by determining the increase in people finishing up started orders (measurements). Fortunately for this example, it's relatively easy to figure out how an API can meet these needs.

Because I'm creating a resource-based API, each request will be a unique URL describing one piece of the back-end structure with a method describing what the client wants to do with that resource. In this case, I have only two different types of resources: individual toppings and lists of toppings. Individual topping resources such as `/api/v1.0/toppings/1` are used for viewing, editing, and deleting a single topping. The list resource `/api/v1.0/toppings` is used for viewing all toppings or for adding a new topping to the list. Table 2.1 shows each call to the API and a description of what it does.

Table 2.1 API calls

API call	Description
`GET /api/v1.0/toppings`	List current toppings
`GET /api/v1.0/toppings/1`	View a single topping
`POST /api/v1.0/toppings`	Create a new topping
`PUT /api/v1.0/toppings/1`	Update an existing topping
`DELETE /api/v1.0/toppings/1`	Delete an existing topping

And that's it. The platform features create, read, update, and delete operations available to you by combining the HTTP methods with the URLs for your resources. But what do you get when you make these calls? When you GET the resource for a single topping, you get information about that topping. Try this now in your browser: http://irresistibleapis.com/api/v1.0/toppings/1.

Listing 2.1 Retrieving a single topping

```
GET /api/v1.0/toppings/1
{                          <-------- Curly braces indicate an object.
  "topping": {
    "id": 1,
    "title": "Pepperoni"
  }
}
```

This response is represented in JavaScript Object Notation (JSON), a formatting syntax first described in chapter 1. JSON is covered in more detail in chapter 4, but for now you can see how the data is structured. (If you want more information about JSON, you can find it at http://json.org.) The curly braces indicate an object, which is a group of pairings of names and values. What's represented here is a JSON structure describing a single object—a "topping," which has an ID of 1 and a title of Pepperoni. This is the same resource address a client can access to view, delete, or update an existing topping. This means that the URL for the single topping is the toppings list of http://irresistibleapis.com/api/v1.0/toppings followed by the ID of the topping from within this structure—so it's http://irresistibleapis.com/api/v1.0/toppings/1.

If you GET the resource for the list of toppings directly, the returned information includes a list instead of a single object. Call this URL in your browser to see the list: http://irresistibleapis.com/api/v1.0/toppings.

Listing 2.2 Retrieving a list of all toppings

```
GET /api/v1.0/toppings
{                          <-------- Curly braces indicate dictionaries.
  "toppings": [            <-------- Square braces indicate lists.
    {
```

```
      "id": 1,
      "title": "Pepperoni"
    },
    {
      "id": 2,
      "title": "Pineapple"
    }
  ]
}
```

In this case, because the request was for a list of objects, square brackets demonstrate that the returned object contains a *list* of toppings. Each individual topping looks the same as listing 2.1. Again, this is how information is represented in JSON. To understand these calls and responses, remember that an *object* (with keys and values) is represented by curly braces, and a *list* (an unnamed collection of items) is represented with square brackets. In some programming languages these are referred to as *hashes* and *arrays*.

Both of these calls can be made from a standard web browser. If other people have added items to the list, you'll see those included in the list view as well; this is a live call into the API system and returns the appropriate information. In this case, the API is generated by node. If you're a developer who's interested in learning more about the back end of the system, Exercise 3 at the end of the chapter will give you information about how to run this system on your own, as well as the application running on top of the API.

This simple API interaction gives you the opportunity to start understanding some of the topics covered in chapter 4.

2.4 Using a web API

You can interact with this API in various ways, as you'll learn in this section. Feel free to try any or all of these approaches to see how the interaction works.

2.4.1 Browser

A browser can make `GET` calls to specific resources easily. Note that this is easy in the case of my demo API because there's no authentication to worry about. The challenge is that the browser doesn't have any way to easily update, delete, or create new items. Using the developer tools or web inspector in your browser can give you more information about the call as well.

For instance, the Chrome web browser has developer tools that allow you to inspect the traffic it's processing. Figure 2.5 shows what these tools look like in the browser. I'll break down what you're seeing here in terms of what I described earlier. Note that the Chrome tools are showing the request and response combined together in the tab.

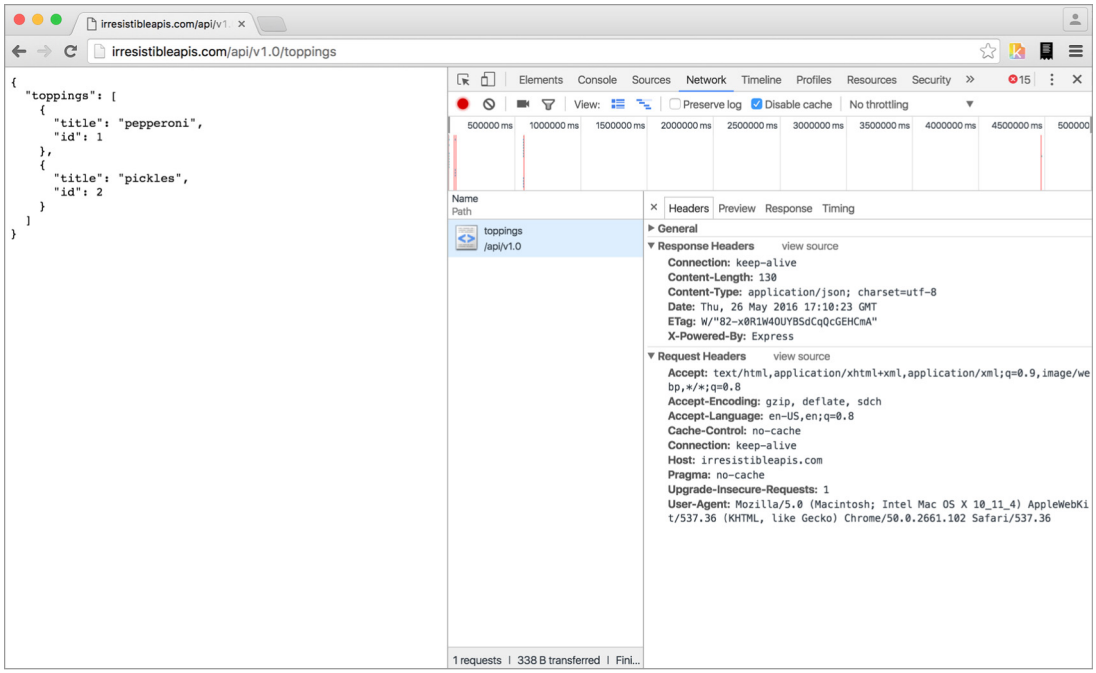


Figure 2.5 The Chrome browser makes it possible to see information about the request and response headers, the body of the request or response, and other useful information about the transaction. Although browsers aren't designed to send `PUT` or `DELETE` responses, the information provided here can go a long way in helping you to understand the interactions with the platform.

For the request:

- *Headers*—`Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp;*/*;q=0.8`—This is the list of accepted formats for this browser request, in order of preference. Because it includes `*/*` (meaning "any content type") late in the list, the browser will accept any type of response and do the best it can with it. Many other headers are shown in figure 2.5. Take a look at them and run the same request on your system to see how they change and what stays the same in each request/response transaction.
- *Method*—`GET`
- *URL*—http://irresistibleapis.com/api/v1.0/toppings
- *Request body*—none
- *Status code*—200 OK

2.4.2 Command line (curl)

If you're comfortable with the command line, you can use the `curl` command to make calls to the API as well. This tool is fairly straightforward and makes it possible to interact with the API more completely, using all the available methods rather than limiting transactions to read operations as the browser does. `curl` is native on UNIX-based systems such as Linux and Macintosh, and you can install it easily for Windows from http://curl.haxx.se/download.html.

Let's take a quick tour through the API using `curl`. By default, `curl` uses `GET` (read), but you can specify other methods on the command line, as shown in the following examples. Remember that your responses may be different if other people have been changing things; go ahead and work with what you get. Don't be shy—this API is for this book, and you can't break anything important. The best way to understand this type of system is to work with it yourself.

First, let's use `curl` to look at a single topping. Lines beginning with a dollar sign indicate a command-line call. The other information is the information returned by the server itself.

Listing 2.3 GET /api/v1.0/toppings/1

```
$ curl http://irresistibleapis.com/api/v1.0/toppings/1
{
  "topping": {
    "id": 1,
    "title": "Pepperoni"
  }
}
```

That seems pretty reasonable. I'd eat a pizza with pepperoni on it. Let's list all the toppings and see what else is on the pizza. Remember that the list for the toppings is at the parent level, or `/api/v1.0/toppings`.

Listing 2.4 GET /api/v1.0/toppings

```
$ curl http://irresistibleapis.com/api/v1.0/toppings
{
  "toppings": [
    {
      "id": 1,
      "title": "Pepperoni"
    },
    {
      "id": 2,
      "title": "Pineapple"
    },
    {
      "id": 3,
      "title": "Pickles"
    }
  ]
}
```

Wait, what? Pickles? That's kind of gross. Let's delete that one. The `id` for it is 3, so the correct path to operate on is `/api/v1.0/toppings/3`.

Listing 2.5 DELETE /api/v1.0/toppings/3

```
curl -i -X DELETE http://irresistibleapis.com/api/v1.0/toppings/3
{
  "result": true
}
```

The response here says we succeeded. To be sure, let's pull a list of toppings again.

Listing 2.6 GET /api/v1.0/toppings

```
$ curl http://irresistibleapis.com/api/v1.0/toppings

{
  "toppings": [
    {
      "id": 1,
      "title": "Pepperoni"
    },
    {
      "id": 2,
      "title": "Pineapple"
    }
  ]
}
```

Okay, that's much better. But our pizza has pepperoni and pineapple, and I'd much prefer ham with my pineapple. Let's go ahead and change that first one to make the pizza how I want it. To update an existing item, the command needs to send a `PUT` to the resource with the new information required.

Listing 2.7 PUT /api/v1.0/toppings/1

```
$ curl -i -H "Content-Type: application/json" -X PUT -d '{"title":"Ham"}'
http://irresistibleapis.com/api/v1.0/toppings/1

{
  "topping": {
    "id": 1,
    "title": "Ham"
  }
}
```

Nice, now the pizza is looking pretty good. But as far as I'm concerned the pizza is merely a vehicle to get cheese in my mouth, so I'll add some extra cheese to go with the Hawaiian pizza I've built.

Listing 2.8 POST /api/v1.0/toppings/1

```
$ curl -H "Content-Type: application/json" -X POST -d '{"title":"Extra extra
 cheese"}' http://irresistibleapis.com/api/v1.0/toppings
{
  "topping": {
    "id": 3,
    "title": "Extra extra cheese"
  }
}
```

Let's do one final check to make sure that the pizza looks good.

Listing 2.9 GET /api/v1.0/toppings

```
$ curl http://irresistibleapis.com/api/v1.0/toppings

{
  "toppings": [
    {
      "id": 1,
      "title": "Ham"
    },
    {
      "id": 2,
      "title": "Pineapple"
    },
    {
      "id": 3,
      "title": "Extra extra cheese"
    }
  ]
}
```

Awesome! Now the pizza is just right.

Note that with `curl` you can also pass `-i` for slightly more chatty information, or `-v` for a much larger dose of verbose output. If you're having fun and you'd like to try those now, feel free. The extra details you'll see are HTTP transaction details, which are described in chapter 4.

2.4.3 HTTP sniffers

Browsers have become good at showing information about the calls they're making, but this is of limited use for a couple of reasons. As I mentioned earlier, a browser is only capable of sending a read request, which restricts the actions you're able to explore. When you submit a form, it creates a create (`POST`) request, but you can't arbitrarily call these operations in your browser.

HTTP sniffers are tools that allow you to explore all the HTTP traffic your system processes. HTTP sniffers watch and report on the network traffic your system is generating, whether it comes from a browser, an application, or a raw command-line call.

With these tools, you can see the entirety of the HTTP request and response, and this allows you to debug what's happening if you're running into issues.

If you're using a Mac, HTTPScoop (www.tuffcode.com) is a friendly choice. It's easy to set up and use, and the output is clear and complete. The downside to this tool is that it can't monitor secure transactions (HTTPS calls), and so it's not going to work with any API requiring secure calls. For the purposes of this book, though, you'll only be accessing a nonsecure API (the demo API), so HTTPScoop is a fine choice—it would be my first choice for any Mac users wanting a reasonably intuitive experience. The license cost is $15, but you can try it for two weeks for free.

Figure 2.6 shows an example of the windows in HTTPScoop. For this chapter, I'll focus on the main screen listing all calls and the Request/Response tab. Later in the book you'll learn about headers, status codes, and other HTTP details so you can understand how they all interact together. For now, though, consider the request to be a simple request and response, and don't worry about particular details if you're not already familiar with HTTP.

For Windows users, the best choice out there is Fiddler, which you can find at www.telerik.com/fiddler. For Windows, Mac, and Linux, there's a slightly more complicated choice in Charles (www.charlesproxy.com). If you're quite advanced in your network administration skills, you can try out Wireshark from www.wireshark.org. Wireshark is available and free for every major platform and sniffs all kinds of traffic, not only web/HTTP traffic, but the interface is complex, and it can be difficult to understand what you're seeing.

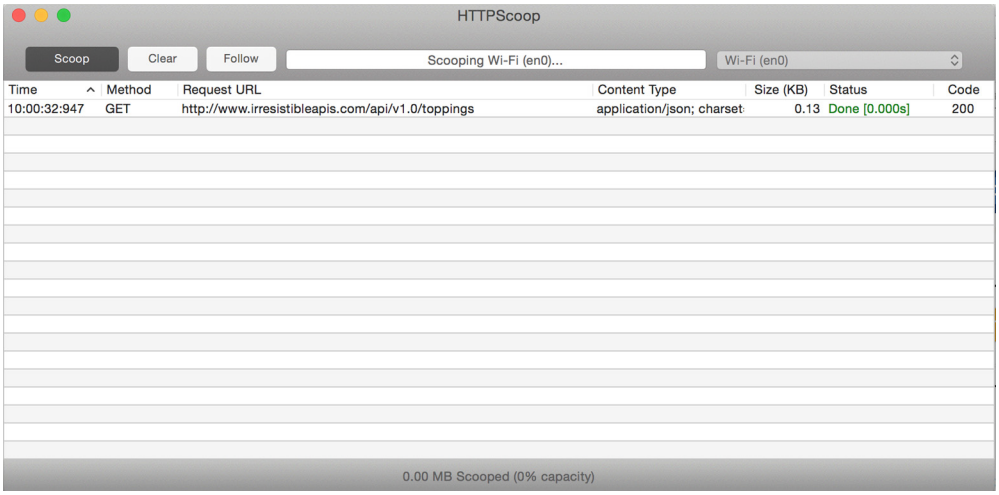


Figure 2.6 This is an example of a call being inspected by HTTPScoop. On this basic landing page, you can see the Request URL, representing the resource. The content type of the response, status code, and response size are also provided.

EXERCISE 1 Watch the traffic in an HTTP sniffer as you go through the exercises from this chapter. Use the `curl` calls to access the API directly and see what the calls look like. For more verbosity with `curl`, you can use `-v` in your command and see more information about the call from the client side. Compare the information in the sniffer to what `curl` sends and see if you can find patterns. Which debugging method gives the best information? Which one is easier for you to use?

EXERCISE 2 Make a deliberately incorrect call. Call `/api/v1.0/toppings/100`—there's not likely 100 toppings on the pizza, so this is a bad call. What kind of output did you get from `curl -v`? What did the HTTP sniffer show? The status code tells you how the system responded, which should give you the information you need to figure out what the issue is.

2.5 Interaction between the API and client

Seeing these `GET` calls to the API is somewhat interesting, but unfortunately you can't see the `POST`, `PUT`, or `DELETE` calls using a browser. `curl` isn't intuitive for exploring a system. Without some kind of application using the API, it's difficult to explore and visualize the elegance and simplicity of this kind of interface.

Keeping in line with the simple API, I've created a simple application to exercise the API, creating a list of toppings for your virtual pizza. Again, for a real application there would be a full pizza and a method to place the order, but this application is deliberately as simple as possible so it's easy to understand how it works.

I'll go through the same sequence I did in the last section. Here's our starting pizza, with pepperoni, pineapple, and pickles. Loading the initial page causes an API call to be generated, and we get the current list of toppings from the system.

First, take a look at the JSON representation returned when the API is called directly at `/api/v1.0/toppings`, shown in figure 2.7. Figure 2.8 shows how the application looks when this API call is made on the back end.

```
{
  "toppings": [
    {
      "title": "pepperoni",
      "id": 1
    },
    {
      "title": "peppers",
      "id": 2
    },
    {
      "title": "pickles",
      "id": 3
    }
  ]
}
```

Figure 2.7 Here you see a representation of the API toppings list in JSON, the markup language used by the platform. As described, the curly braces indicate an object, or dictionary, and the square brackets represent an array, or list of objects.

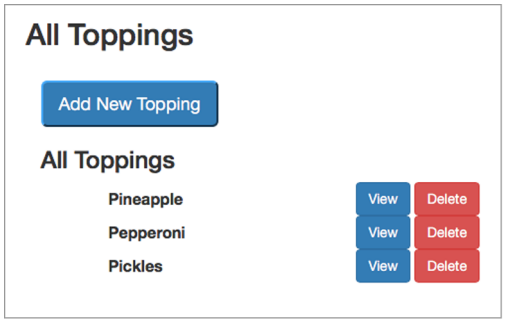


Figure 2.8 The application view for the toppings list shows the same information, as shown in figure 2.4. This screen is created by calling the toppings list and creating the HTML based on the returned information. If the list changes on the server, both figure 2.4 and figure 2.5 would change, with both showing the same information in different ways.

Now take a look at the main application at http://irresistibleapis.com/demo. With the JSON data, the simple application can build the front page. Some of the items are static—they don't change. The top half of the page, for instance, is always the same, with the title of the display and a button to add new toppings. The bottom half, though, is created based on the information retrieved from the API. Each topping is listed, and the ID of the topping is used to create an appropriate button to act on that specific item. The user has no need to understand the relationship between the ID and the name of the topping, but the IDs are used programmatically to set up the page to be functionally correct. Note how the information in the API in figure 2.4 directly maps to what's shown in the application in figure 2.5. The buttons on this page map directly to the other API calls, as shown in table 2.2.

Table 2.2 The mapping between the API calls and application functions

API call	Application function
`GET /api/v1.0/toppings`	Main application page
`GET /api/v1.0/toppings/1`	View button on main page
`POST /api/v1.0/toppings`	"Add new topping"
`DELETE /api/v1.0/toppings/1`	Delete button on either page

As we walk through the API actions, use the HTTP sniffer of your choice to watch the traffic as the interactions happen. Note that because this system is live, other people may have added, deleted, or edited the toppings, and they may not match. Feel free to use the buttons to adjust the toppings to match or follow along with your own favorite toppings (Jalapeños? Sun dried tomatoes? Legos?).

The first action in the previous example was removing the pickles from the pizza, and clicking Delete on this page for the Pickles entry will do that. This button knows which ID to operate on because it was embedded in the page when the listing was rendered.

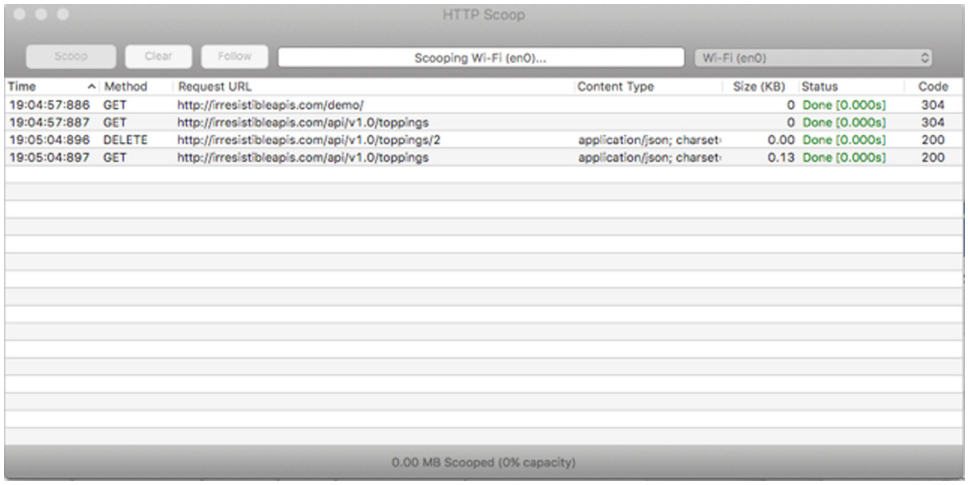


Figure 2.9 This HTTPScoop screen shows a list of all the calls made by the system. In this case, you can see the `DELETE` method is called to remove the `/toppings/2` resource from the system, and it was successful, as indicated by the 2XX response in the code column.

Clicking the Delete button will make the `DELETE` call and then make a call to the API to re-render the list of toppings with the deleted topping gone. If you're using an HTTP sniffer or have configured your browser to show you web traffic, you can see this call happening from your system. Figure 2.9 shows what it looks like in HTTPScoop.

As you can see, the application pulled a few different framework files and then got the full listing for the main page. When I clicked Delete, the application sent a `DELETE` request to the API server and then requested a new list of toppings. All the requests were successful, so the main page refreshed to show the new list. Figure 2.10 shows the list after I deleted the offending pickles from the toppings list.

To edit an existing topping, in this case to change Pepperoni to Ham, click the View button. Doing so makes the read call for the specific item and allows you to edit the title. Using this technique to edit the Pepperoni to Ham and then clicking Save causes a `PUT` to happen exactly as in the original example. Watch your HTTP sniffer or browser traffic to see how this progression works. Figure 2.11 shows what the Edit page looks like for a particular topping—in this case I changed the title from Pepperoni to Ham.

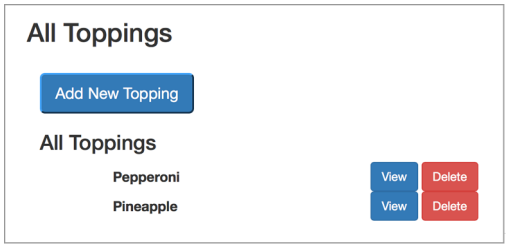


Figure 2.10 Once the topping has been deleted from the system, the HTML representation of the toppings list no longer shows the deleted topping. If the platform call is made (to `/toppings`) you'll see that the change is reflected in the JSON representation as well.

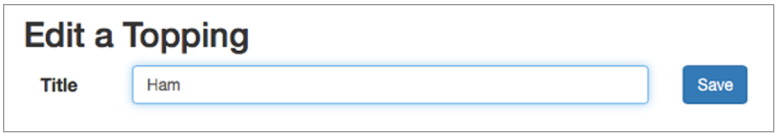


Figure 2.11 The Edit a Topping screen allows you to change the title of an existing resource.

When this change is `PUT` to the API, it will change the item's title from Pepperoni to Ham, updating the database to reflect the change permanently.

The `PUT` request, viewed in HTTPScoop, shows the request and response (see figure 2.12).

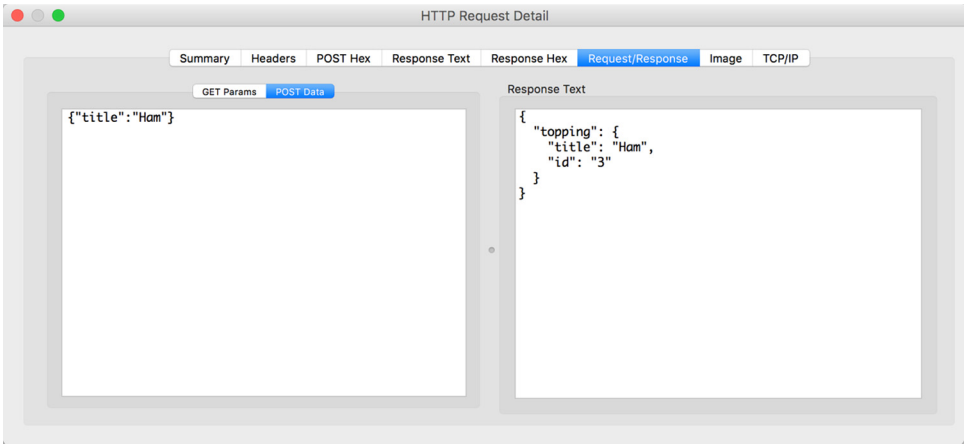


Figure 2.12 When you change the title of an existing resource, the information is sent to the server, and it sends back the new representation of that item. In this case, the object is quite simple; the title is the only field that can be changed. This is a simple demonstration of how an update works on an API platform.

As with the associated `curl` request earlier, the debugging demonstrates that the client sends a request including the new information for the requested item. A `PUT` request replaces information for an existing item in the system. In the response, the server returns a response showing the new values for the resource. This returned object matches the object that was `PUT` to the system. Without HTTPScoop, this seems a little magical, but you should be seeing a pattern by this point; these common operations are direct mappings to system calls on the back end of the application.

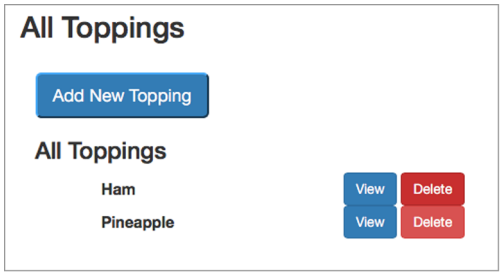


Figure 2.13 The list of toppings now includes Ham and Pineapple; the Pickles have been deleted (thank heavens), and the Pepperoni has been changed to Ham using an update. Again, if you made a call to the `/toppings` resource you'd see the changes shown in the JSON representation as well.

Again, once the topping is edited, the application redisplays the main page, now with Ham and Pineapple (figure 2.13).

What's left then? Now I need to add my extra cheese to the pizza, because it's my favorite sort of thing. Clicking the Add New Topping button on the main page gives me a page for adding a new topping, as shown in figure 2.14. Remember, adding a new item to the list is a `POST` action, and that's what will happen on the back end. Figure 2.15 shows what the API transaction looks like when this `POST` is sent.

Figure 2.14 The Add a Topping screen is designed to add new toppings to the system. As mentioned earlier, a create action is generally represented by a `POST` operation, and that's what the system will do in this case.

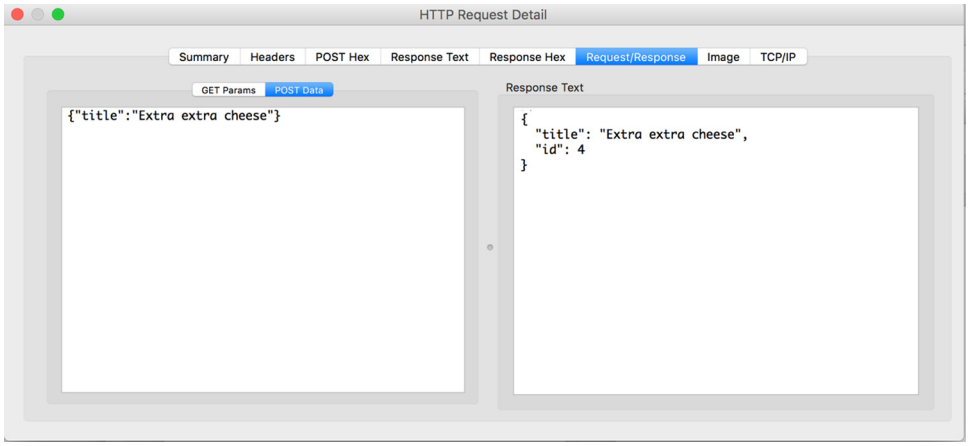


Figure 2.15 HTTPScoop `POST` request/response. The only field needed to create a new topping is the title, and it's set to `Extra extra cheese` (yum!). The response shows the ID and title—the entire representation—of the newly added item.

This example demonstrates again the difference between PUT, which updates a specific existing item, and POST, which creates a new item by adding it to the specified list. After adding this new topping to the system, the application again requests the list of toppings, which brings the web page back once again to the main page. This completes the circuit using an application to exercise the back-end API. The single page running this application is quite straightforward, because all the logic and actions are happening on the back end using the API.

Now that you've had the opportunity to view some specific traffic, take time to play with the example application with the various HTTP inspection methods. Because this sample application runs in your browser, you have the option of using developer tools in your browser to watch the traffic or an HTTP sniffer for this exploration. For the exercises in this book, you'll want to use an HTTP sniffer, so pick the one you're most comfortable with and start familiarizing yourself with its use.

Advanced Example Note

If you're a developer and want to install your own copy of this system, follow the instructions in section 2.6 to do so. Otherwise, skip to section 2.7 for a summary of this chapter.

2.6 Install your own API and front end

This optional section is designed specifically for developers who want to understand more completely the back-end functionality of the API and sample application. You can use a Docker container to run the system quickly on your own system or download the code from my GitHub repository. I'll walk through the steps to install and use the Docker container first and then give more general instructions for grabbing the code from GitHub to run on your own system.

2.6.1 Installing the system via Docker

Docker is extremely simple to install on Linux systems and quite easy on Mac OS X and Windows systems as well. Installing the Docker container is simple once you've got the Docker system set up. Using this container allows you to separate the code and processes from the main processes on your system while avoiding the memory and space-management issues of more heavyweight virtual machine systems. The Docker installers for installation on Windows and Macintosh are at www.docker.com/toolbox.

If you're an advanced user running Windows and already have virtualization working via VBox or another virtualization system, you need to be aware that Docker relies on VirtualBox, which may conflict with your existing setup. Additionally, boot2docker requires that virtualization be available on your system, which infrequently requires changes to the BIOS. Also, virtualization is only available on 64-bit systems. If your system is a 32-bit system, you'll need to install the code directly from GitHub.

Once you've installed Docker using the instructions at the Docker website, you're ready to pull and run the container.

On Linux, issue this command (on one line):

```
% sudo docker run -p 80:3000 synedra/irresistible
```

That binds your system's port 80 to the Docker container on port 3000.

On systems using boot2docker (Windows or Mac OS X), the command is as follows (root access isn't needed because of the nature of docker-machine):

```
% docker run -p 80:3000 synedra/irresistible
```

The application automatically runs in the Docker container. When using boot2docker, the Docker engine assigns a separate IP address for Docker containers. In order to determine the IP address of your Docker container, issue the command `docker-machine ip default`. Once you've done that, you can access the system at http://<docker-ip/. Because the server is running on port 80, the default web port, the browser will find the web server on that port.

If you'd like to start the container and explore the code, you can do so with the following command, which won't start the node server:

```
% docker run -i -t synedra/irresistible /bin/bash
```

You'll now be root in a shell within the container. Accessing the system in this way allows you to look at the code and figure out how all the pieces are working together. The application itself is composed of the toppings.js file, and the front-end web server is run from the static/index.html file. The previous command will allow you to access the application directly without cross-domain issues. You can read more about Docker port forwarding at https://docs.docker.com/userguide/dockerlinks/.

If you're running Docker directly on Linux, you can access the system directly at http://localhost. If you already have a web service running on the default port, you can assign a different port in the `docker run` command.

2.6.2 Installing the system via Git

If you prefer to run the applications on your own system rather than using the Docker container, you need to have Git and Node.js installed on your system. The commands needed to pull the repository to your system and install and run node are as follows:

```
% git clone https://github.com/synedra/irresistible
% cd irresistible/
% curl -sL https://deb.nodesource.com/setup | bash - && apt-get
 install -yq nodejs build-essential
% npm install -g npm
% npm config set registry http://registry.npmjs.org/
% npm install -g express@2.5.1
% npm install express
% npm install
% node toppings.js
```

From there you can access the system at http://localhost:3000 (or port 3000 on whichever server you're using). Node.js runs on port 3000 by default, so if you want to expose the system on the standard port (80), you'll want to run a separate server on the front end—something like Nginx or Apache—and then create a reverse proxy back to the node server. For security reasons it's best not to use root to run a bare web service, and you can't access the standard ports as a regular user. This is one of the advantages to using the Docker system—because it's isolated from the rest of your system at its own IP address, it's safe to run the front-end server on port 80.

2.6.3 Exploring the code

As you're running the system and exploring it, you'll see the logs for the system show up in the terminal window where you started up the web server. Using an HTTP sniffer, you can watch the API traffic your system is generating as described in section 2.3. Once you've started a web browser at http://docker_ip_address/, not only will you be able to see the traffic in an HTTP sniffer, but you'll start seeing server entries in the terminal window that you started.

The logs show you all the traffic—both front-end calls to / and the back-end requests to the API. This combined log data makes it easy to see how the systems are interacting.

If you used the Docker setup, you were placed directly into the /opt/webapp directory. The Git instructions will put you in the same directory: the webapp subdirectory of the repository. Table 2.3 shows a listing of the files in the program directory along with a description of what each one does.

Table 2.3 Files included in the program directory

Filename	Description
Procfile	Used if you want to deploy this to Heroku
Toppings.js	The main program for the system
static/index.html	A simple single-page application that exercises the API

The toppings.js file is used to run the node web server. When you type node toppings.js, the application looks for the index.html file in the static directory and serves it up.

The application uses Bootstrap, a single-page application framework that makes your simple applications look pretty. The formatting pieces are mostly contained within the Bootstrap framework, and overrides are made within the index.html file. This is all to explain what the `id` and `style` attributes are for each `<div>` on the page. In this case, it's using the main-single-template for the outside wrapper, and the inside is a main-single container. This function will present the table of items for the page to render.

The `$.get` function makes the call to `/api/v1.0/toppings`, at which point the back end returns a list of toppings, and this function is called to render the page.

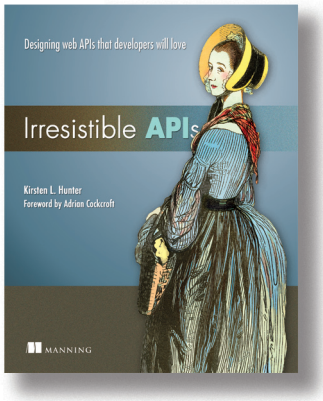
EXERCISE 3 Play around with the page, see how each piece works, and try to see if you can make the application go directly to the Edit page from the toppings list instead of the View page.

2.7 Summary

At this point you've either played directly with my hosted service or set up your own. This chapter covered the following concepts:

- The structure of a simple web API system includes the required actions for a complete platform: create, read, update, and delete.
- A basic HTTP transaction includes a clearly defined request and response, creating a foundation for web APIs.
- From HTTP sniffers to Chrome Developer Tools, the ability to monitor the traffic makes it much easier to understand what's happening between the systems.
- RESTful API ideals define the endpoints as nouns, and not verbs. Between these ideas and the HTTP transactions they work with, the web API system is complete.

Now that you have an understanding of the various moving pieces in a simple API, you can begin thinking about your own API at a higher level: how to architect the entire system to use the simple pieces I discussed here to build a fantastic API system. This chapter was about the bottom up, and how the cogs and wheels work together to make things work. The next chapter will help you to learn how to think top down: what are the goals for your API system and how can you meet them most efficiently?



It takes a village to deliver an irresistible web API. Business stakeholders look for an API that works side-by-side with the main product to enhance the experience for customers. Project managers require easy integration with other products or ways for customers to interact with your system. And, developers need APIs to consistently interoperate with external systems. The trick is getting the whole village together. This book shows you how.

Irresistible APIs presents a process to create APIs that succeed for all members of the team. In it, you'll learn how to capture an application's core business value and extend it with an API that will delight the developers who use it. Thinking about APIs from the business point of view, while also considering the end-user experience, encourages you to explore both sides of the design process and learn some successful biz-to-dev communication patterns. Along the way, you'll start to view your APIs as part of your product's core value instead of just an add-on.

What's inside

- Design-driven development
- Developing meaningful use cases
- API guiding principles
- How to recognize successful APIs

Written for all members of an API design team, regardless of technical level.

Architectures and Patterns

Like most ideas in software development, serverless applications can take many forms based on the required use cases. This chapter will give you an overview of how serverless applications work along with a glimpse into several architectures and patterns that you can apply while building them.

Chapter 2 from *Serverless Architectures on AWS*
by Peter Sbarski

Architectures and patterns

This chapter covers

- Use cases for serverless architectures
- Examples of patterns and architectures

What are the use cases for serverless architectures, and what kinds of architectures and patterns are useful? We're often asked about use cases as people learn about a serverless approach to the design of systems. We find that it's helpful to look at how others have applied technology and what kinds of use cases, designs, and architectures they've produced. Our discussion will center on these use cases and sample architectures. This chapter will give you a solid understanding of where serverless architectures are a good fit and how to think about design of serverless systems.

2.1 Use cases

Serverless technologies and architectures can be used to build entire systems, create isolated components, or implement specific, granular tasks. The scope for use of serverless design is large, and one of its advantages is that it's possible to use it for small and large tasks alike. We've designed serverless systems that power web

and mobile applications for tens of thousands of users, and we've built simple systems to solve specific, minute problems. It's worth remembering that serverless is not just about running code in a compute service such as Lambda. It's also about using third-party services and APIs to cut down on the amount of work you must do.

2.1.1 Application back end

In this book you're going to build a back end for a media-sharing, YouTube-like application. It will allow users to upload video files, transcode these files to different playable formats, and then allow other users to view them. You'll construct an entirely serverless back end for a fully featured web application with a database and a RESTful API. And we're going to show that serverless technologies are appropriate for building scalable back ends for all kinds of web, mobile, and desktop applications.

Technologies such as AWS Lambda are relatively new, but we've already seen large serverless back ends that power entire businesses. Our serverless platform, called A Cloud Guru (http://acloud.guru), supports many thousands of users collaborating in real time and streaming hundreds of gigabytes of video. Another example is Instant (http://instant.cm), which is a serverless content management system for static websites. And yet another example is a hybrid-serverless system built by EPX Labs. We'll discuss all of these systems later in the chapter.

Apart from web and mobile applications, serverless is a great fit for IoT applications. Amazon Web Services (AWS) has an IoT platform (https://aws.amazon.com/iot-platform/how-it-works/) that combines the following:

- Authentication and authorization
- Communications gateway
- Registry (a way to assign a unique identity to each device)
- Device shadowing (persistent device state)
- A rules engine (a service to transform and route device messages to AWS services)

The rules engine, for example, can save files to Amazon's Simple Storage Service (S3), push data to an Amazon Simple Queue Service (SQS) queue, and invoke AWS Lambda functions. Amazon's IoT platform makes it easy to build scalable IoT back ends for devices without having to run a server.

A serverless application back end is appealing because it removes a lot of infrastructure management, has granular and predictable billing (especially when a serverless compute service such as Lambda is used), and can scale well to meet uneven demand.

2.1.2 Data processing and manipulation

A common use for serverless technologies is data processing, conversion, manipulation, and transcoding. We've seen Lambda functions built by other developers for processing of CSV, JSON, and XML files; collation and aggregation of data; image resizing; and format conversion. Lambda and AWS services are well suited for building event-driven pipelines for data-processing tasks.

In chapter 3, you'll build the first part of your application, which is a powerful pipeline for converting videos from one format to another. This pipeline will set file permissions and generate metadata files. It will run only when a new video file is added to a designated S3 bucket, meaning that you'll pay only for execution of Lambda when there's something to do and not while the system is idling. More broadly, however, we find data processing to be an excellent use case for serverless technologies, especially when we use a Lambda function in concert with other services.

2.1.3 Real-time analytics

Ingestion of data—such as logs, system events, transactions, or user clicks—can be accomplished using services such as Amazon Kinesis Streams (see appendix A for more information on Kinesis). Lambda functions can react to new records in a stream, and can process, save, or discard data quickly. A Lambda function can be configured to run when a specific number (batch size) of records is available for processing, so that it doesn't have to execute for every individual record added to the stream.

Kinesis streams and Lambda functions are a good fit for applications that generate a lot of data that need to be analyzed, aggregated, and stored. When it comes to Kinesis, the number of functions spawned to process messages off a stream is the same as the number of shards (therefore, there's one Lambda function per shard). Furthermore, if a Lambda function fails to process a batch, it will retry. This can keep going for up to 24 hours (which is how long Kinesis will keep data around before it expires) if processing fails each time. But even with these little gotchas (which you now know), the combination of Kinesis streams and Lambda is really powerful if you want to do real-time processing and analytics.

2.1.4 Legacy API proxy

One innovative use case of the Amazon API Gateway and Lambda (which we've seen a few times) is what we refer to as the legacy API proxy. Here, developers use API Gateway and Lambda to create a new API layer over legacy APIs and services to make them easier to use. The API Gateway is used to create a RESTful interface, and Lambda functions are used to transpose request/response and marshal data to formats that legacy services understand. This approach makes legacy services easier to consume for modern clients that may not support older protocols and data formats.

2.1.5 Scheduled services

Lambda functions can run on a schedule, which makes them effective for repetitive tasks like data backups, imports and exports, reminders, and alerts. We've seen developers use Lambda functions on a schedule to periodically ping their websites to see if they're online and send an email or a text message if they're not. There are Lambda blueprints available for this (a *blueprint* is a template with sample code that can be selected when creating a new Lambda function). And we've seen developers write Lambda functions to perform nightly downloads of files off their servers and send

daily account statements to users. Repetitive tasks such as file backup and file validation can also be done easily with Lambda thanks to the scheduling capability that you can set and forget.

2.1.6 Bots and skills

Another popular use of Lambda functions and serverless technologies is to build bots (a *bot* is an app or a script that runs automated tasks) for services such as Slack (a popular chat system—https://slack.com). A bot made for Slack can respond to commands, carry out small tasks, and send reports and notifications. We, for example, built a Slack bot in Lambda to report on the number of online sales made each day via our education platform. And we've seen developers build bots for Telegram, Skype, and Facebook's messenger platform.

Similarly, developers write Lambda functions to power Amazon Echo skills. Amazon Echo is a hands-free speaker that responds to voice commands. Developers can implement *skills* to extend Echo's capabilities even further (a skill is essentially an app that can respond to a person's voice; for more information, see http://amzn.to/2b5NMFj). You can write a skill to order a pizza or quiz yourself on geography. Amazon Echo is driven entirely by voice, and skills are powered by Lambda.

> **A note on writing a bot for Slack**
>
> There are two Lambda blueprints (you'll see these when you create a Lambda function in AWS) that can help you build a Slack bot quickly (look for "slack-echo-command" and "cloudwatch-alarm-to-slack"). Slack bots need to respond within 3,000 milliseconds; otherwise, you may get a timeout error message. If you hit timeouts, think about creating two bots: one to receive a command and another to post a notification to your Slack channel when the result is available.

2.2 Architectures

The two overarching architectures that we'll discuss in this book are *compute as back end* (that is, back ends for web and mobile applications) and *compute as glue* (pipelines built to carry out workflows). These two architectures are complementary. It's highly likely that you'll build and combine these architectures if you end up working on any kind of real-world serverless system. Most of the architectures and patterns described in this chapter are specializations and variations of these two to some extent.

2.2.1 Compute as back end

The compute-as-back-end architecture describes an approach where a serverless compute service such as Lambda and third-party services are used to build a back end for web, mobile, and desktop applications. You may note in figure 2.1 that the front end links directly to the database and an authentication service. This is because there's no need to put every service behind an API Gateway if the front end can communicate with them in a secure manner (for example, using delegation tokens; chapters 5 and 9

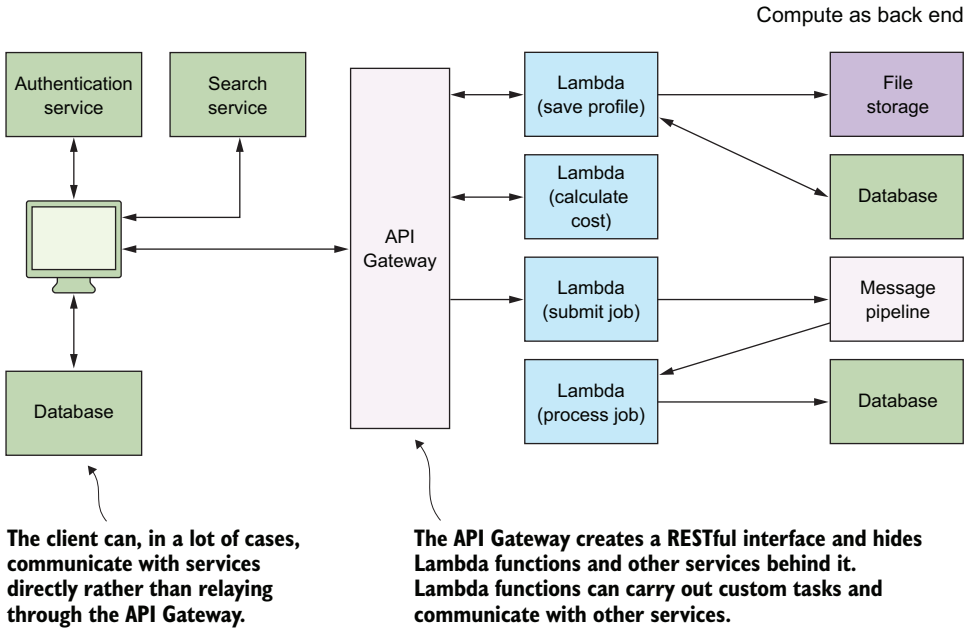


Figure 2.1 This is a rather simple back end architecture for storing, calculating, and retrieving data. The front end can read directly from the database and securely communicate with different services. It can also invoke Lambda functions through the API Gateway.

discuss this in more detail). One of the aims of this architecture is to allow the front end to communicate with services, encompass custom logic in Lambda functions, and provide uniform access to functions via a RESTful interface.

In chapter 1, we described our principles of serverless architectures. Among them we mentioned thicker front ends (principle 4) and encouraged the use of third-party services (principle 5). These two principles are particularly relevant if you're building a serverless back end rather than event-driven pipelines. We find that good serverless systems try to minimize the scope and the footprint of Lambda functions so that these functions do only the bare minimum (call them *nano functions*, if you will) and primarily focus on the tasks that must not be done in the front end because of privacy or security concerns. Nevertheless, finding the right level of granularity for a function can be a challenging task. Make functions too granular and you'll end up with a sprawling back end, which can be painful to debug and maintain after a long time. Ignore granularity and you'll risk building mini-monoliths that nobody wants (one helpful lesson we've learned is to try to minimize the number of data transformations in a Lambda function to keep complexity under control).

A CLOUD GURU

A Cloud Guru (https://acloud.guru) is an online education platform for solution architects, system administrators, and developers wanting to learn Amazon Web Services. The core features of the platform include (streaming) video courses, practice

exams and quizzes, and real-time discussion forums. A Cloud Guru is also an e-commerce platform that allows students to buy courses and watch them at their leisure. Instructors who create courses for A Cloud Guru can upload videos directly to an S3 bucket, which are immediately transcoded to a number of different formats (1080p, 720p, HLS, WebM, and so on) and are made available for students to view. The Cloud Guru platform uses Firebase as its primary client-facing database, which allows clients to receive updates in near real time without refreshing or polling (Firebase uses web sockets to push updates to all connected devices at the same time). Figure 2.2 shows a cut down version of the architecture used by A Cloud Guru.

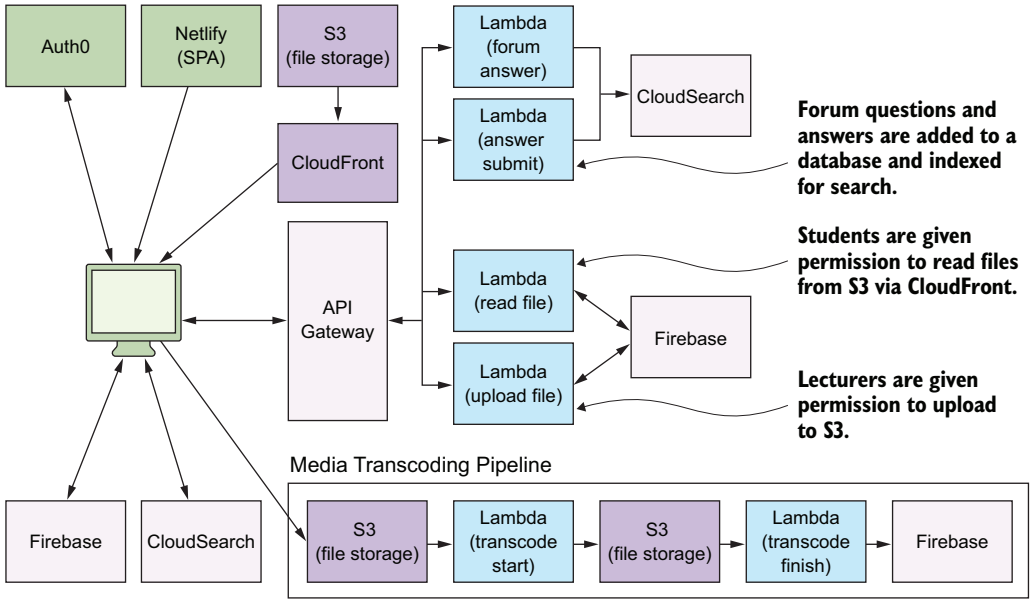


Figure 2.2 This is a simplified version of the Cloud Guru architecture. Current production architecture has additional Lambda functions and services for performing payments, managing administration, gamification, reporting, and analytics.

Note the following about the Cloud Guru architecture given in figure 2.2:

- The front end is built using AngularJS and is hosted by Netlify (https://netlify.com). You could use S3 and CloudFront (CloudFront is a global content delivery network provided by AWS) instead of Netlify if you wanted to.
- Auth0 is used to provide registration and authentication facilities. It creates delegation tokens that allow the front end to directly and securely communicate with other services such as Firebase.
- Firebase is the real-time database used by A Cloud Guru. Every client creates a connection to Firebase using web sockets and receives updates from it in near real time. This means that clients receive updates as they happen without having to poll.

- Lecturers who create content for the platform can upload files (usually videos, but they could be other types) straight to S3 buckets via their browser. For this to work, the web application invokes a Lambda function (via the API Gateway) to request the necessary upload credentials first. As soon as credentials are retrieved, the client web application begins a file upload to S3 via HTTP. All of this happens behind the scenes and is opaque to the user.
- Once a file is uploaded to S3, it automatically kicks off a chain of events (our event-driven pipeline) that transcodes the video, saves new files in another bucket, updates the database, and immediately makes transcoded videos available to other users. Throughout this book you'll write a similar system and see how it works in detail.
- To view videos, users are given permission by another Lambda function. Permissions are valid for 24 hours, after which they must be renewed. Files are accessed via CloudFront.
- Users can submit questions and answers to the forums. Questions, answers, and comments are recorded in the database. This data is then sent to for indexing to AWS CloudSearch, which is a managed searching and indexing service from AWS. This allows users to search and view questions, answers, and comments that other people have written.

INSTANT

Instant (http://instant.cm) is a startup that helps website owners add content management facilities—including inline text editing and localization—to their static websites. The founders, Marcel Panse and Sander Nagtegaal, describe it as instant content management system. Instant works by adding a small JavaScript library to a website and making a minor change to HTML. This allows developers and administrators to edit text elements directly via the website's user interface. Draft edits made to the text are stored in DynamoDB (see appendix A on DynamoDB). The final, production version of the text (that the end user sees) is served as a JSON file from an S3 bucket via Amazon CloudFront (figure 2.3).

A simplified version of the Instant architecture is shown in figure 2.4.

Note the following about the Instant architecture in figure 2.4:

- (This is not shown in the diagram.) A JavaScript library must be added to a website that wants to use Instant. Authentication is done via Google (with the user's own Google account) by clicking a widget that appears in the website at a special URL (for example, yourwebsite.com/#edit). After successful authentication with Google, the Instant JavaScript widget authenticates with AWS Cognito, which provisions temporary AWS IAM credentials (see appendix A for information on AWS Cognito).
- Route 53, Amazon's Domain Name System (DNS) web service, is used to route requests either to CloudFront or to the API Gateway. (See appendix A for more information on Route 53.)

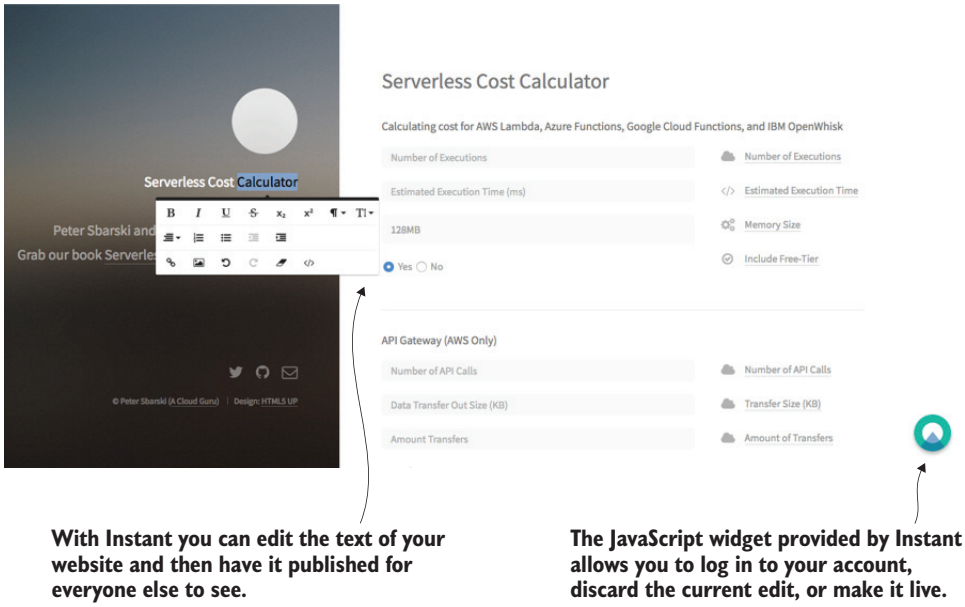


Figure 2.3 You can use Instant to add support for multiple languages, which makes it a powerful service if you need to localize your website and don't have a content management system.

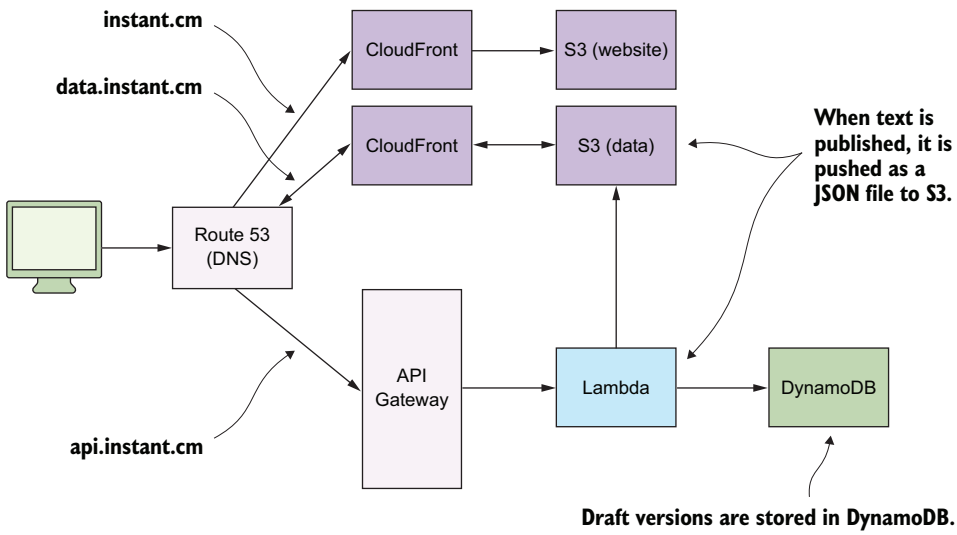


Figure 2.4 The Instant system uses AWS Lambda, API Gateway, DynamoDB, S3, CloudFront, and Amazon Route 53 as its main components. The system scales to support many clients.

- As a user edits text on their website, the Instant widget sends changes to the API Gateway, which invokes a Lambda function. This Lambda function saves drafts to DynamoDB, along with relevant metadata.
- When the user decides to publish their edit (by selecting an option in the Instant widget), data from DynamoDB is read and saved in S3 as a static JSON file. This file is served from S3 via CloudFront. The Instant widget parses the JSON file received from CloudFront and updates the text on the website for the end user to see.

Marcel and Sander make a few points about their system:

> *The use of Lambda functions leads to an architecture of microservices quite naturally. Every function is completely shielded from the rest of the code. It gets better: the same Lambda function can fire in parallel in almost infinite numbers—and this is all done completely automated.*

In terms of cost, Marcel and Sander share the following:

> *With our serverless setup, we primarily pay for data transfer through CloudFront, a tiny bit for storage and for each millisecond that our Lambda functions run. Since we know on average what a new customer uses, we can calculate the costs per customer exactly. That's something we couldn't do in the past, when multiple users were shared across the same infrastructure.*

Overall, Marcel and Sander find that adopting an entirely serverless approach has been a winner for them primarily from the perspectives of operations, performance, and cost.

2.2.2 Legacy API proxy

The legacy API proxy architecture is an innovative example of how serverless technologies can solve problems. As we mentioned in section 2.1.4, systems with outdated services and APIs can be difficult to use in modern environments. They might not conform to modern protocols or standards, which might make interoperability with current systems harder. One way to alleviate this problem is to use the API Gateway and Lambda in front of those legacy services. The API Gateway and Lambda functions can transform requests made by clients and invoke legacy services directly, as shown in figure 2.5.

The API Gateway can transform requests (to an extent) and issue requests against other HTTP endpoints (see chapter 7). But it works only in a number of fairly basic (and limited) use cases where only JSON transformation is needed. In more complex scenarios, however, a Lambda function is needed to convert data, issue requests, and process responses. Take a Simple Object Access Protocol (SOAP) service as an example. You'd need to write a Lambda function to connect to a SOAP service and then map responses to JSON. Thankfully, there are libraries that can take care of much of the heavy lifting in a Lambda function (for example, there are SOAP clients that can be downloaded from the npm registry for this purpose; see https://www.npmjs.com/package/soap).

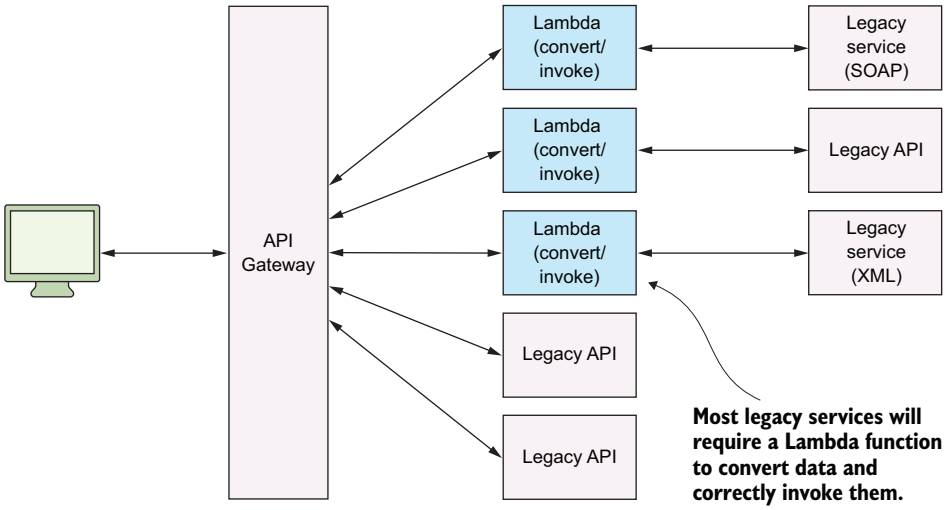


Figure 2.5 The API proxy architecture is used to build a modern API interface over old services and APIs.

2.2.3 Hybrid

As we mentioned in chapter 1, serverless technologies and architectures are not an all-or-nothing proposition. They can be adopted and used alongside traditional systems. The hybrid approach may work especially well if a part of the existing infrastructure is already in AWS. We've also seen adoption of serverless technologies and architectures in organizations with developers initially creating standalone components (often to do additional data processing, database backups, and basic alerting) and over time integrating these components into their main systems; see figure 2.6.

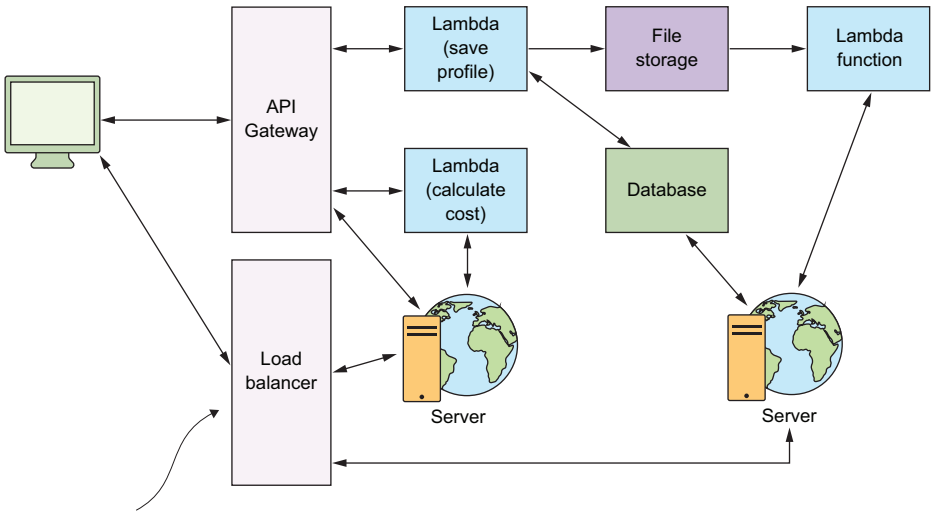


Figure 2.6 The hybrid approach is useful if you have a legacy system that uses servers.

EFFICIENT HYBRID-SERVERLESS JOB-PROCESSING SYSTEM

EPX Labs (http://epxlabs.com) proudly state that the "future of IT Operations and Application Development is less about servers and more about services." They specialize in serverless architectures, with one of their recent solutions being a hybrid serverless system designed to carry out maintenance and management jobs on a distributed server-based infrastructure running on Amazon's Elastic Compute Cloud (EC2) (figure 2.7).

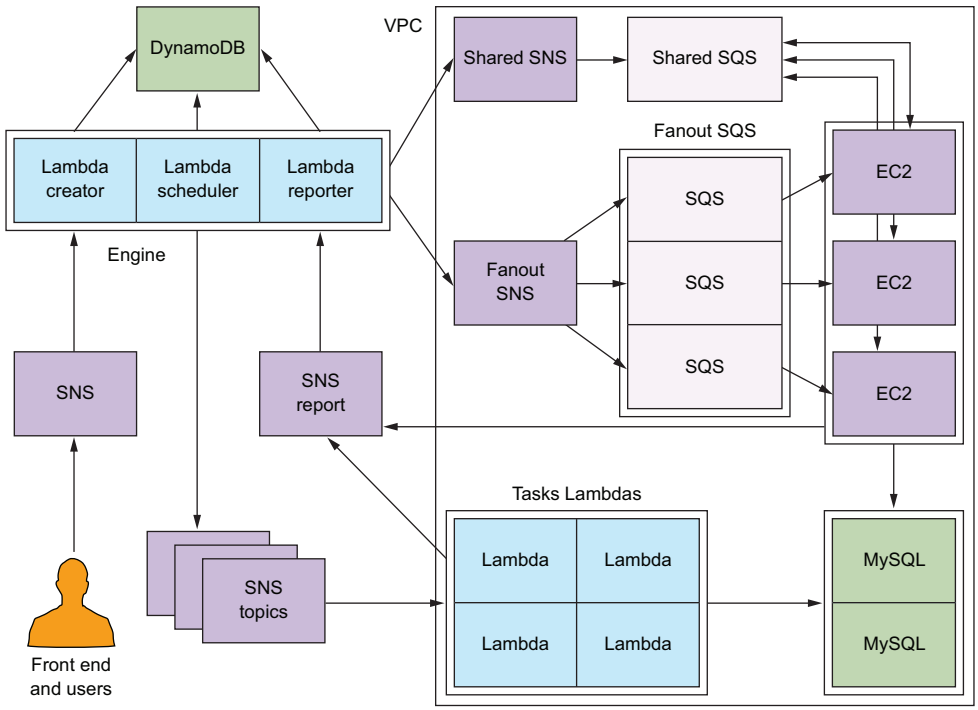


Figure 2.7 The Hybrid-Serverless Job-Processing System designed by EPX Labs

Evan Sinicin and Prachetas Prabhu of EPX Labs describe the system they had to work with as a "multi-tenant Magento (https://magento.com) application running on multiple frontend servers. Magento requires certain processes to run on the servers such as cache clearing and maintenance operations. Additionally, all site management operations such as build, delete, and modify require a mix of on-server operations (building out directory structures, modifying configuration files, etc.) as well as database operations (creating new database, modifying data in database, and so on)." Evan and Prachetas created a scalable serverless system to assist with these tasks. Here's how they describe how the system is built and the way it works:

- The system is broken into two parts: the engine, which is responsible for creating, dispatching, and managing jobs, and the task processors.

- The engine consists of several Lambda functions fronted by the Simple Notification Service (SNS—see appendix A for more information). Task processors are a mix of Lambda and Python processes.
- A job is created by sending JSON data to the creator (part of the engine) via an SNS topic. Each job is broken down into a set of discrete tasks. Tasks fall into three categories:
  - Individual server tasks—must be executed on all servers.
  - Shared server tasks—must be executed by one server.
  - Lambda tasks—executed by a Lambda function.
- Once created in DynamoDB, the job is sent to the scheduler, which identifies the next task to be run and dispatches it. The scheduler dispatches the task based on the type of task, either pinging a task Lambda via SNS or placing messages onto the shared or fan-out Simple Queue Service (SQS) queues (see section 2.3 for more information on these patterns).
- Task execution on the servers is handled by custom-written Python services. Two services run on each server; one polls the shared SQS queue for shared server tasks and the other polls the individual server queue (specific to an EC2 instance). These services continually poll the SQS queues for incoming task messages and execute them based on the contained information. To keep this service stateless, all data required for processing is encapsulated in the encrypted message.
- Each Lambda task corresponds to a discrete Lambda function fronted by an SNS topic. Typically, Lambda tasks operate on the MySQL databases backing Magento; therefore, they run in the virtual private cloud (VPC). To keep these Lambda functions stateless, all data required for processing is encapsulated in the encrypted message itself.
- Upon completion or failure, the task processors will report the success or failure to the engine by invoking the reporter Lambda via SNS. The reporter Lambda will update the job in DynamoDB and invoke the scheduler to do any cleanup (in the case of a failure) or dispatch the next task.

2.2.4 GraphQL

GraphQL (http://graphql.org) is a popular data query language developed by Facebook in 2012 and released publicly in 2015. It was designed as an alternative to REST (Representational State Transfer) because of REST's perceived weaknesses (multiple round-trips, over-fetching, and problems with versioning). GraphQL attempts to solve these problems by providing a hierarchical, declarative way of performing queries from a single end point (for example, api/graphql); see figure 2.8.

GraphQL gives power to the client. Instead of specifying the structure of the response on the server, it's defined on the client (http://bit.ly/2aTjlh5). The client can specify what properties and relationships to return. GraphQL aggregates data from multiple sources and returns it to the client in a single round trip, which makes

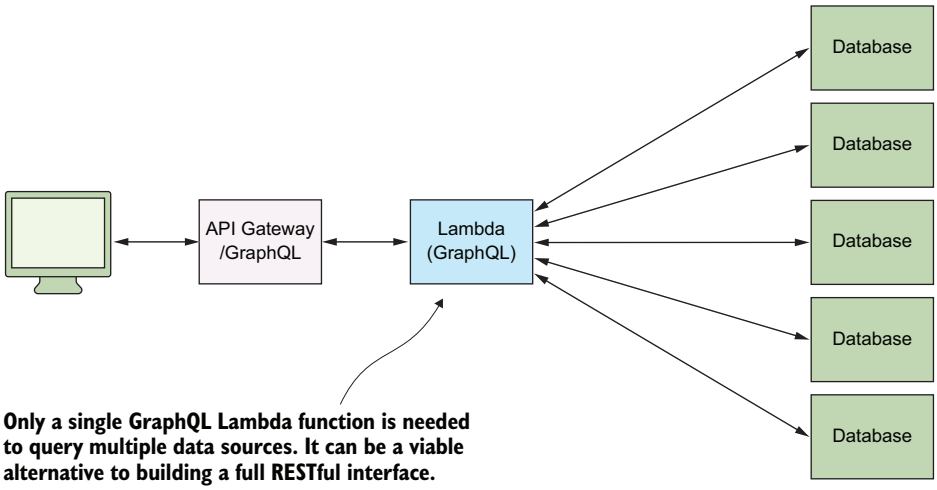


Figure 2.8 The GraphQL and Lambda architecture has become popular in the serverless community.

it an efficient system for retrieving data. According to Facebook, GraphQL serves millions of requests per second from nearly 1,000 different versions of its application.

In serverless architectures, GraphQL is usually hosted and run from a single Lambda function, which can be connected to an API Gateway (there are also hosted solutions of GraphQL like scaphold.io). GraphQL can query and write to multiple data sources, such as DynamoDB tables, and assemble a response that matches the request. A serverless GraphQL is a rather interesting approach you might want to look at next time you need to design an interface for your API and query data. Check out the following articles if you want to implement GraphQL in a serverless architecture:

- "A Serverless Blog leveraging GraphQL to offer a REST API with only 1 endpoint" (https://github.com/serverless/serverless-graphql-blog)
- "Serverless GraphQL" (http://bit.ly/2aN7Pc2)
- "Pokémon Go and GraphQL with AWS Lambda" (http://bit.ly/2aIhCud)

2.2.5 Compute as glue

The compute-as-glue architecture shown in figure 2.9 describes the idea that we can use Lambda functions to create powerful execution pipelines and workflows. This often involves using Lambda as *glue* between different services, coordinating and invoking them. With this style of architecture, the focus of the developer is on the design of their pipeline, coordination, and flow of data. The parallelism of serverless compute services like Lambda helps to make these architectures appealing. The example you're going to build in this book uses this pattern to create an event-driven pipeline that transcodes videos (chapter 3, in particular, focuses on creating pipelines and applying this pattern to solve a complex task rather easily).

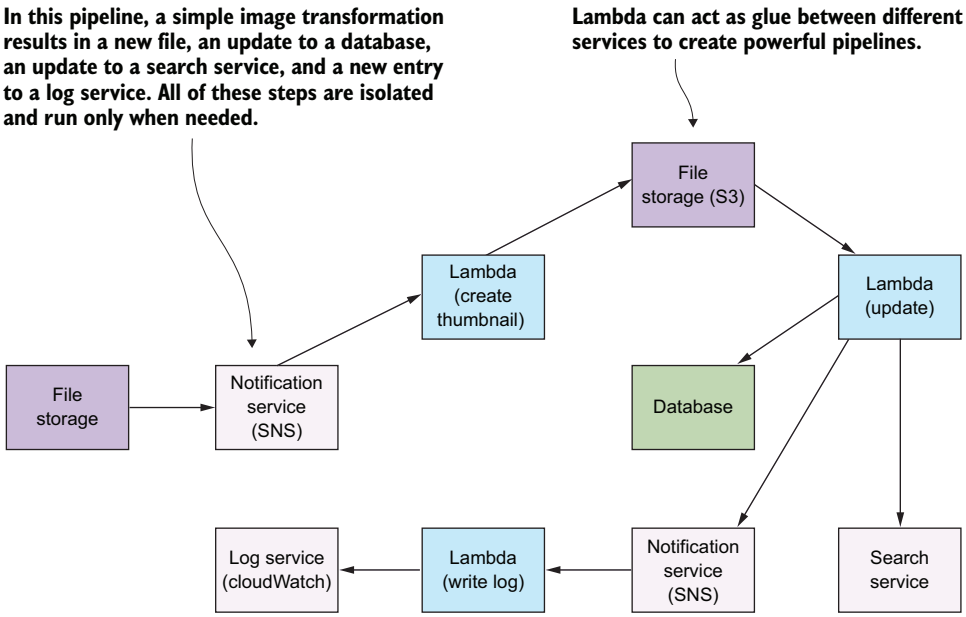


Figure 2.9 The compute-as-glue architecture uses Lambda functions to connect different services and APIs to achieve a task.

ListHub processing engine

EPX Labs has built a system to process large real estate XML feeds (figure 2.10). Evan Sinicin and Prachetas Prabhu say that the goal of their system is "to pull the feed, separate the large file into single XML documents, and process them in parallel. Processing includes parsing, validation, hydration, and storing."

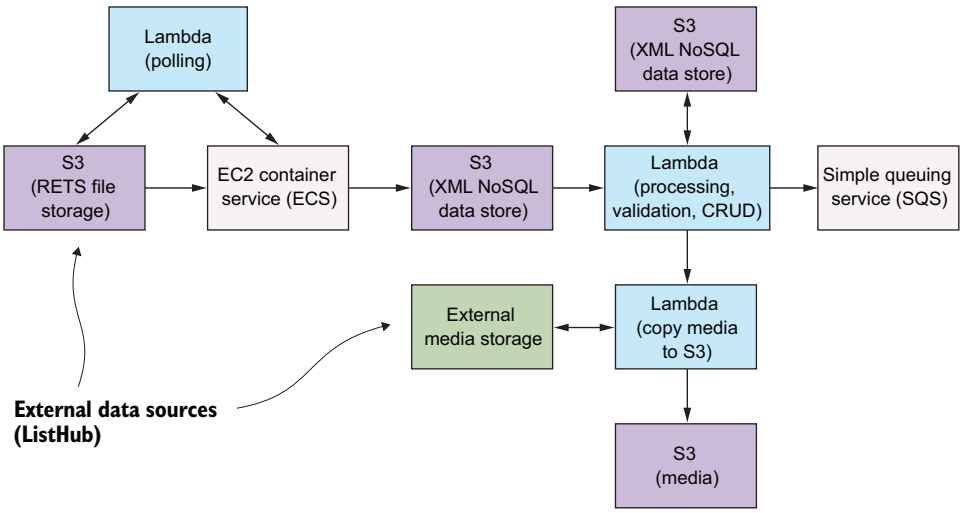


Figure 2.10 EPX Labs has built a system to effortlessly process large (10 GB+) XML documents.

They go on to describe how the system works in more detail:

- The system was designed to process a real estate listing XML feed. The feed is provided by ListHub as a massive (10 GB+) XML document with millions of nested listings. This file is provided via S3 for direct download and processing. The listings conform to the Real Estate Standards Organization (RETS) standard.
- ListHub does not have any sort of push capabilities, so the polling Lambda checks the last-modified metadata of the S3 object to see if a new feed has been posted. This usually occurs every 12 hours or so.
- Once a new feed has been published, the polling Lambda spins up an EC2 Container Service (ECS) container to carry out the parsing of the massive file. ECS is used because this process can take a long time (Lambda can run for a maximum of 5 minutes). The ECS container has a Clojure program that asynchronously processes the feed file and places the parsed information into S3.
- EPX Labs uses S3 as a NoSQL store. Using an S3 PutObject event trigger, each new XML listing placed into S3 triggers a Lambda that carries out the validation and hydration processes. Another S3 bucket stores processed listing IDs (as object keys). The validation Lambda can quickly verify that the listing hasn't been processed on a previous run by checking whether the ID/key already exists.
- The validation Lambda also triggers the hydration Lambda ("Copy Media to S3 Lambda"). This Lambda copies assets such as pictures and videos to an S3 bucket so they can be displayed on the front end.
- The final step is to save the relevant, normalized listing data into the final data store that serves the front end and other systems. To avoid overwhelming the data store with writes, the listing data is put onto an SQS queue so it can be processed at a rate the final data store can handle.

Evan and Prachetas say that their approach yields a number of benefits, including that they can use S3 as a cheap, high-performance, and scalable NoSQL data store and that they can use Lambda to undertake massively concurrent processing.

2.2.6 Real-time processing

As discussed in section 2.1.3, Amazon Kinesis Streams is a technology that can help process and analyze large amounts of streaming data. This data can include logs, events, transactions, social media feeds—virtually anything you can think of—as shown in figure 2.11. It's a good way to continuously collect data that may change over time. Lambda is a perfect tool for Kinesis Streams because it scales automatically in response to how much data there is to process.

With Kinesis Streams you can accomplish the following:

- Control how much data is passed into a Kinesis stream before a Lambda function is invoked and how data gets to Kinesis in the first place
- Put a Kinesis stream behind an API Gateway
- Push data to the stream directly from a client or have a Lambda function add records to it

Kinesis Streams can ingest a lot of messages that can be processed with Lambda functions. Data-intensive applications that perform real-time reporting and analytics can benefit from this architecture.

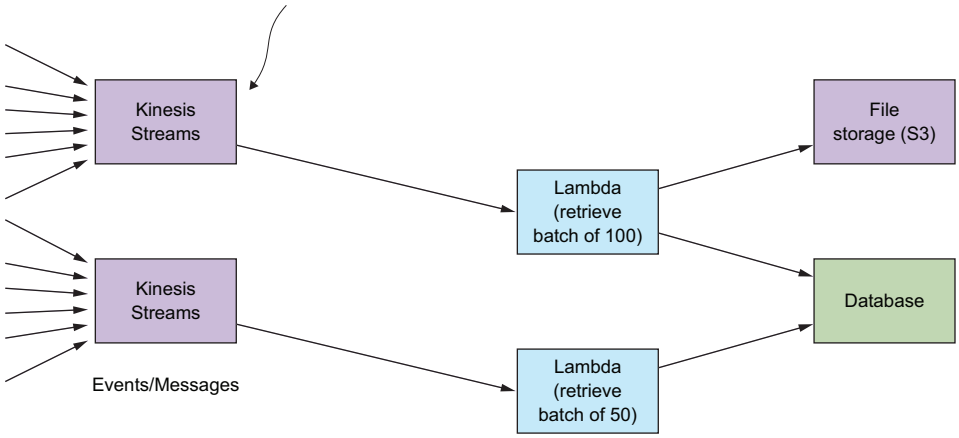


Figure 2.11 Lambda is a perfect tool to process data in near real time.

2.3 Patterns

Patterns are architectural solutions to problems in software design. They're designed to address common problems found in software development. They're also an excellent communications tool for developers working together on a solution. It's far easier to find an answer to a problem if everyone in the room understands which patterns are applicable, how they work, their advantages, and their disadvantages. The patterns presented in this section are useful for solving design problems in serverless architectures. But these patterns aren't exclusive to serverless. They were used in distributed systems long before serverless technologies became viable. Apart from the patterns presented in this chapter, we recommend that you become familiar with patterns relating to authentication (see chapter 4 for a discussion of the federated identity pattern), data management (CQRS, event sourcing, materialized views, sharding), and error handling (retry pattern). Learning and applying these patterns will make you a better software engineer, regardless of the platform you choose to use.

2.3.1 Command pattern

With the GraphQL architecture (section 2.2.4), we discussed the fact that a single end point can be used to cater to different requests with different data (a single GraphQL endpoint can accept any combination of fields from a client and create a response that matches the request). The same idea can be applied more generally. You can design a system in which a specific Lambda function controls and invokes other functions. You can connect it to an API Gateway or invoke it manually and pass messages to it to invoke other Lambda functions.

In software engineering, the command pattern (figure 2.12) is used to "encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations" because of the "need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request" (http://bit.ly/29ZaoWt). The command pattern allows you to decouple the caller of the operation from the entity that carries out the required processing.

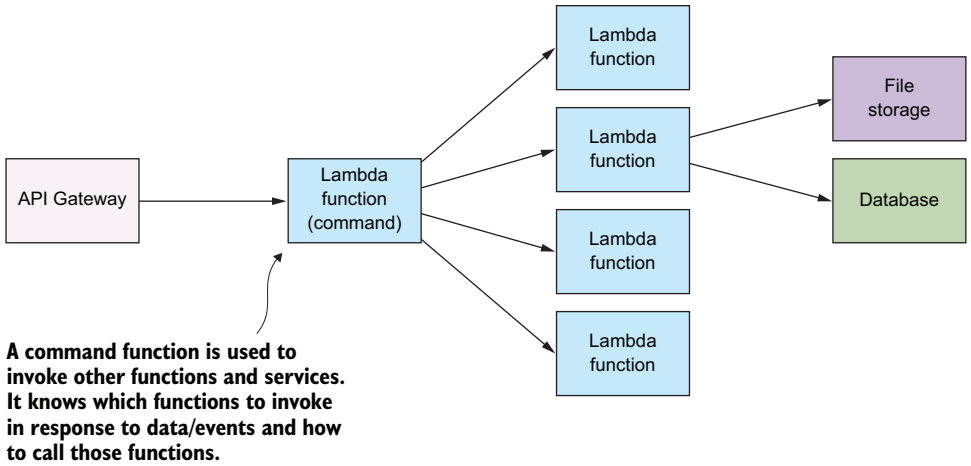


Figure 2.12 The command pattern is used to invoke and control functions and services from a single function.

In practice, this pattern can simplify the API Gateway implementation, because you may not want or need to create a RESTful URI for every type of request. It can also make versioning simpler. The command Lambda function could work with different versions of your clients and invoke the right Lambda function that's needed by the client.

WHEN TO USE THIS

This pattern is useful if you want to decouple the caller and the receiver. Having a way to pass arguments as an object, and allowing clients to be parametrized with different requests, can reduce coupling between components and help make the system more extensible. Be aware of using this approach if you need to return a response to the API Gateway. Adding another function will increase latency.

2.3.2 Messaging pattern

Messaging patterns, shown in figure 2.13, are popular in distributed systems because they allow developers to build scalable and robust systems by decoupling functions and services from direct dependence on one another and allowing storage of events/records/requests in a queue. The reliability comes from the fact that if the consuming service goes offline, messages are retained in the queue and can still be processed at a later time.

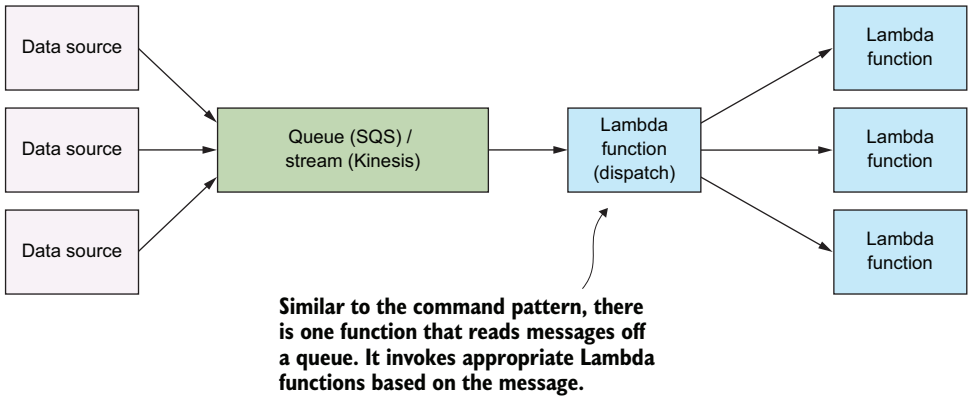


Figure 2.13 The messaging pattern, and its many variations, are popular in distributed environments.

This pattern features a message queue with a sender that can post to the queue and a receiver that can retrieve messages from the queue. In terms of implementation in AWS, you can build this pattern on top of the Simple Queue Service. Unfortunately, at the moment Lambda doesn't integrate directly with SQS, so one approach to addressing this problem is to run a Lambda function on a schedule and let it check the queue every so often.

Depending on how the system is designed, a message queue can have a single sender/receiver or multiple senders/receivers. SQS queues typically have one receiver per queue. If you needed to have multiple consumers, a straightforward way to do it is to introduce multiple queues into the system (figure 2.14). A strategy you could apply is to combine SQS with Amazon SNS. SQS queues could subscribe to an SNS topic; pushing a message to the topic would automatically push the message to all of the subscribed queues.

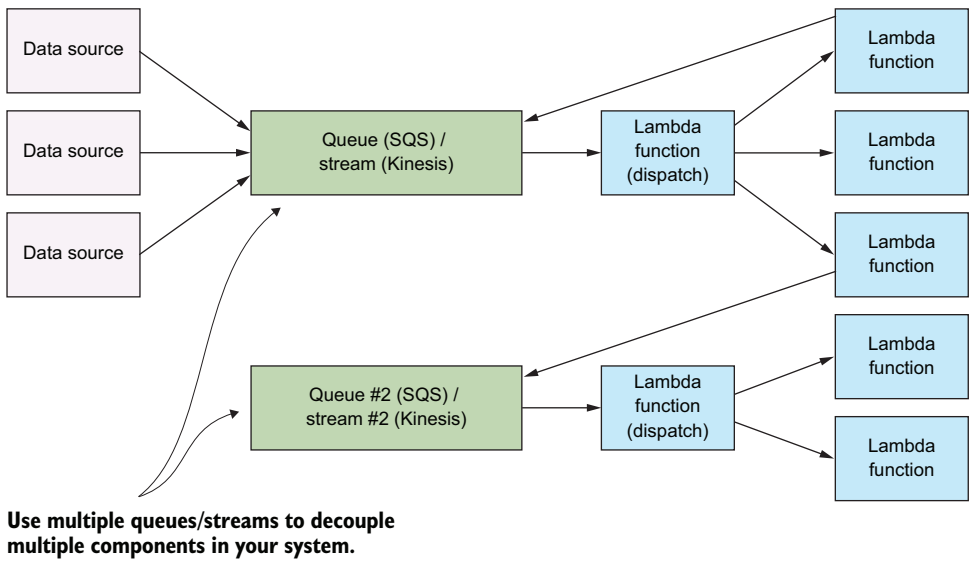


Figure 2.14 Your system may have multiple queues/streams and Lambda functions to process all incoming data.

Kinesis Streams is an alternative to SQS, although it doesn't have some features, such as dead lettering of messages (http://amzn.to/2a3HJzH). Kinesis Streams integrates with Lambda, provides an ordered sequence of records, and supports multiple consumers.

WHEN TO USE THIS

This is a popular pattern used to handle workloads and data processing. The queue serves as a buffer, so if the consuming service crashes, data isn't lost. It remains in the queue until the service can restart and begin processing it again. A message queue can make future changes easier, too, because there's less coupling between functions. In an environment that has a lot of data processing, messages, and requests, try to minimize the number of functions that are directly dependent on other functions and use the messaging pattern instead.

2.3.3 *Priority queue pattern*

A great benefit of using a platform such as AWS and serverless architectures is that capacity planning and scalability are more of a concern for Amazon's engineers than for you. But in some cases, you may want to control how and when messages get dealt with by your system. This is where you might need to have different queues, topics, or streams to feed messages to your functions. Your system might go one step further and have entirely different workflows for messages of different priority. Messages that need immediate attention might go through a flow that expedites the process by using more expensive services and APIs with more capacity. Messages that don't need to be processed quickly can go through a different workflow, as shown in figure 2.15.

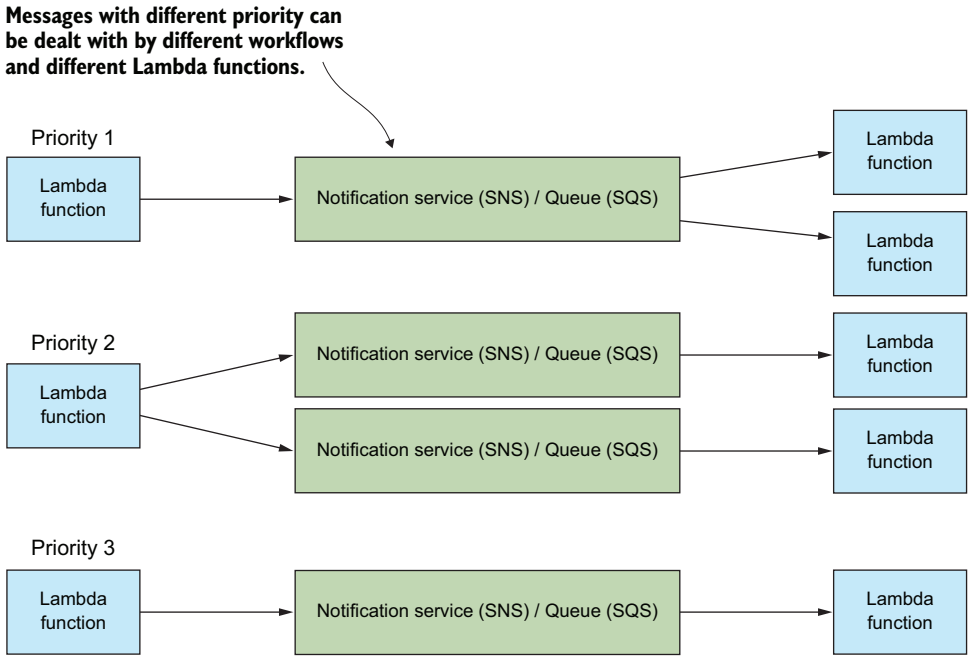


Figure 2.15 The priority queue pattern is an evolution of the messaging pattern.

This pattern might involve the creation and use of entirely different SNS topics, Kinesis Streams, SQS queues, Lambda functions, and even third-party services. Try to use this pattern sparingly, because additional components, dependencies, and workflows will result in more complexity.

WHEN TO USE THIS

This pattern works when you need to have a different priority on processing of messages. Your system can implement workflows and use different services and APIs to cater to many types of needs and users (for example, paying versus nonpaying users).

2.3.4 Fan-out pattern

Fan-out is a type of messaging pattern that's familiar to many users of AWS. Generally, the fan-out pattern is used to push a message out to all listening/subscribed clients of a particular queue or a message pipeline. In AWS, this pattern is usually implemented using SNS topics that allow multiple subscribers to be invoked when a new message is added to a topic. Take S3 as an example. When a new file is added to a bucket, S3 can invoke a single Lambda function with information about the file. But what if you need to invoke two, three, or more Lambda functions at the same time? The original function could be modified to invoke other functions (like the command pattern), but that's a lot of work if all you need is to run functions in parallel. The answer is to use the fan-out pattern using SNS; see figure 2.16.

A message added to an SNS topic can force invocation of multiple Lambda functions in parallel.

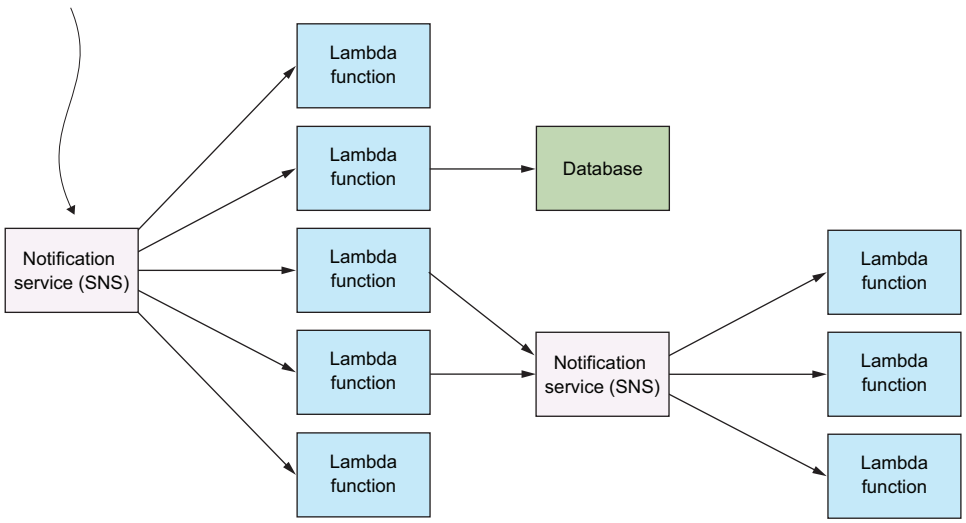


Figure 2.16 The fan-out pattern is useful because many AWS services (such as S3) can't invoke more than one Lambda function when an event takes place.

SNS topics are communications/messaging channels that can have multiple publishers and subscribers (including Lambda functions). When a new message is added to a topic, it forces invocation of all subscribers in parallel, thus causing the event to *fan out*. Going back to the S3 example discussed earlier, instead of invoking a single-message Lambda function, you can configure S3 to push a message onto an SNS topic to invoke all subscribed functions at the same time. It's an effective way to create event-driven architectures and perform operations in parallel. You'll implement this yourself in chapter 3.

WHEN TO USE THIS

This pattern is useful if you need to invoke multiple Lambda functions at the same time. An SNS topic will try and retry to invoke your Lambda functions if it fails to deliver the message or if the function fails to execute. Furthermore, the fan-out pattern can be used for more than just invocation of multiple Lambda functions. SNS topics support other subscribers such as email and SQS queues. Adding a new message to a topic can invoke Lambda functions, send an email, or push a message on to an SQS queue, all at the same time.

2.3.5 Pipes and filters pattern

The purpose of the pipes and filters pattern is to decompose a complex processing task into a series of manageable, discrete services organized in a pipeline (figure 2.17). Components designed to transform data are traditionally referred to as *filters*, whereas connectors that pass data from one component to the next component are referred to as *pipes*. Serverless architecture lends itself well to this kind of pattern. This is useful for all kinds of tasks where multiple steps are required to achieve a result.

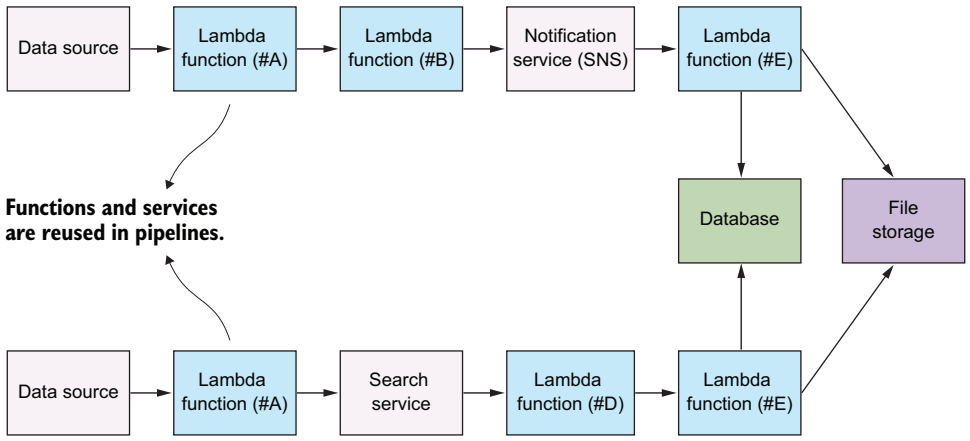


Figure 2.17 This pattern encourages the construction of pipelines to pass and transform data from its origin (pump) to its destination (sink).

We recommend that every Lambda function be written as a granular service or a task with the single-responsibility principle in mind. Inputs and outputs should be clearly defined (that is, there should be a clear interface) and any side effects minimized. Following this advice will allow you to create functions that can be reused in pipelines and more broadly within your serverless system. You might notice that this pattern is similar to the compute-as-glue architecture we described previously. The compute-as-glue architecture is closely inspired by this pattern.

When to use this

Whenever you have a complex task, try to break it down into a series of functions (a pipeline) and apply the following rules:

- Make sure your function follows the single-responsibility principle.
- Make the function idempotent; that is, your function should always produce the same output for given input.
- Clearly define an interface for the function. Make sure inputs and outputs are clearly stated.
- Create a black box. The consumer of the function shouldn't have to know how it works, but it must know to use it and what kind of output to expect every time.

2.4 Summary

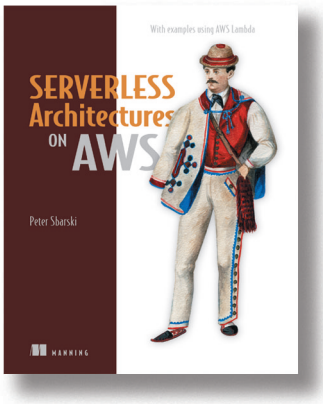
This chapter focused on use cases, architectures, and patterns. These are critical to understand and consider before embarking on a journey to build your system. The architectures we discussing include the following:

- Compute as back end
- Compute as glue
- Legacy API wrapper
- Hybrid
- GraphQL
- Real-time processing

In terms of patterns, we covered these:

- Command pattern
- Messaging pattern
- Priority queue pattern
- Fan-out pattern
- Pipes and filters pattern

Throughout the rest of this book, we're going to apply elements we explored in this chapter, with a particular focus on creating compute-as-back-end and compute-as-glue architectures. In the next chapter, you'll begin building your serverless applications by implementing the compute-as-glue architecture and trying the fan-out pattern.



There's a shift underway toward serverless cloud architectures. With the release of serverless compute technologies, such as AWS Lambda, developers are now building entirely serverless platforms at scale. In these new architectures, traditional back-end servers are replaced with cloud functions acting as discrete single-purpose services. By composing and combining these serverless cloud functions together in a loose orchestration, and adopting useful third-party services, you can build powerful yet easy to understand applications. Serverless architecture is about building rich, scalable, high-performing, and cost-effective systems without having to worry about traditional compute infrastructure, having more time to focus on code, and moving quickly.

Serverless Architectures on AWS teaches you how to build, secure and manage serverless architectures that can power the most demanding web and mobile apps. You'll get going quickly with this book's ready-made and real-world examples, code snippets, diagrams, and descriptions of architectures that can be readily applied. This book describes a traditional application and its back end concerns and then shows how to solve these same problems with a serverless approach. You'll begin with a high-level overview of what serverless is all about, start creating your own media transcoding system, and learn more about AWS. Next, you'll go in depth and learn about Lambda, API Gateway and other important serverless technologies. This section will teach you how to compose Lambda functions and discuss important considerations when it comes to building serverless systems. The third part of the book focuses on more advanced topics as your architecture grows. By the end, you'll be able to reason about serverless systems and be able to compose your own systems by applying these ideas and examples.

What's inside

- Creating a serverless back end
- Using Lambda and the API Gateway
- Connecting multiple services
- Authorization and authentication in a serverless environment
- Securely communicating with third-party services
- Interacting with a database from the front end
- Setting up continuous integration and deployment
- Building high-performance systems using messaging and eventing
- Using AWS to your advantage

This book is for all software developers interested in back end technologies. Experience with JavaScript (node.js) and AWS is useful but not required.

Designing an Authentication Service

There's no better way to understand serverless than seeing it in action. In this chapter, you'll learn to add an authentication service to a serverless application, illustrating both the flexibility of the serverless approach and the advantages of using existing hosted services to extend it.

Chapter 8 from *AWS Lambda in Action* by Danilo Poccia.

Designing an authentication service

This chapter covers

- Designing a sample event-driven application
- Interacting with your users via JavaScript
- Sending emails from Lambda functions
- Storing data in Amazon DynamoDB
- Managing encrypted data

In the previous chapter you learned how to use standalone Lambda functions from different client applications:

- A web page, using JavaScript
- A native Mobile App, with the help of the AWS Mobile Hub to generate your starting code
- An Amazon API Gateway to generate server-side dynamic content for web browsers

Now it's time to build your first event-driven serverless application, using multiple functions together to achieve your purpose. Your goal is to implement a sample

authentication service that can be used by itself or together with Amazon Cognito with developer-authenticated identities.

> **NOTE** The authentication service you're going to build is an example of an event-driven serverless application and hasn't been validated by a security audit. If you need such a service, my advice is to use an already built and production-ready implementation, such as Amazon Cognito User Pools.

You'll define the architecture of your serverless back end built with AWS Lambda. In the chapter after this one, you'll implement all the required components. The first step is to define how your users interact with the application.

8.1 The interaction model

To make your application easy to use for a broad range of use cases, the main interface for your users is the web browser. Via a web browser, users can access static HTML pages that include JavaScript code, which can call one or more Lambda functions to execute code in the back end. At the end of the chapter, you'll see how it is easy to reuse the same flow and architecture with a mobile app.

Using the Amazon API Gateway

Another option, instead of calling Lambda functions directly from the client application, is to model a RESTful API with the Amazon API Gateway, using features similar to what you learned in chapter 3. The advantage of this approach is the decoupling of the client application from the actual back-end implementation:

- You call a Web API from the client application and not a Lambda function.
- You can easily change the back end implementation to (or from) AWS Lambda at any time, without affecting the development of the client application (for example, a web or mobile app).
- You can potentially open your back end to other services, publishing a public API that can further extend the reach of your application.

The Amazon API Gateway provides other interesting features, such as

- SDK generation
- Caching of function results
- Throttling to withstand traffic spikes

However, for the purpose of this book, I decided to use AWS Lambda directly in the authentication service. This makes the overall implementation simpler to build and more understandable for a first-time learner.

If you're building a new application, I advise you to evaluate the pros and cons of using the Amazon API Gateway as I did and make an informed decision.

The HTML pages, JavaScript code, and any other file required to render the page correctly on the web browser (such as CSS style sheets) can be stored on Amazon S3 as publicly readable objects. To store structured data, such as user profiles and passwords,

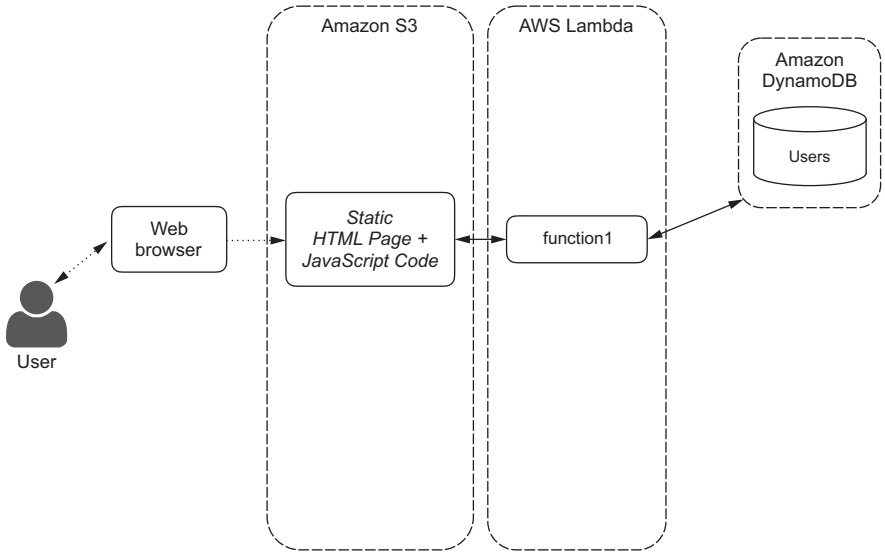


Figure 8.1 The first step in implementing the interaction model for your application: using a web browser to execute back end logic via Lambda functions that can store data on DynamoDB tables

Lambda functions can use DynamoDB tables. A summary of this interaction model is shown in figure 8.1.

> **TIP** Because the client side of the application is built using HTML pages and JavaScript code, it's relatively easy to repackage it as a hybrid mobile app, using frameworks such as Apache Cordova (formerly PhoneGap). Hybrid apps are popular because you can develop a mobile client once and use it in multiple environments, such as iOS, Android, and Windows Mobile. For more information on using Apache Cordova to implement mobile apps, please look at: https://cordova.apache.org.

It's important for an authentication service to verify contact data provided by users. A common use case is to verify that the email address given by a user is valid. To do that, the Lambda functions in the back end need to send emails to the users. To avoid the complexity of configuring and managing an email server, you can use Amazon Simple Email Services (SES) to send emails. This allows you to extend your interaction model adding this capability (figure 8.2).

> **NOTE** Amazon SES is a fully managed email service that you can use to send any volume of email, and receive emails that can be automatically stored on Amazon S3 or processed by AWS Lambda. When you receive an email with Amazon SES, you can also send a notification using Amazon Simple Notification Service (SNS). For more information on Amazon SES, see https://aws.amazon.com/ses/.

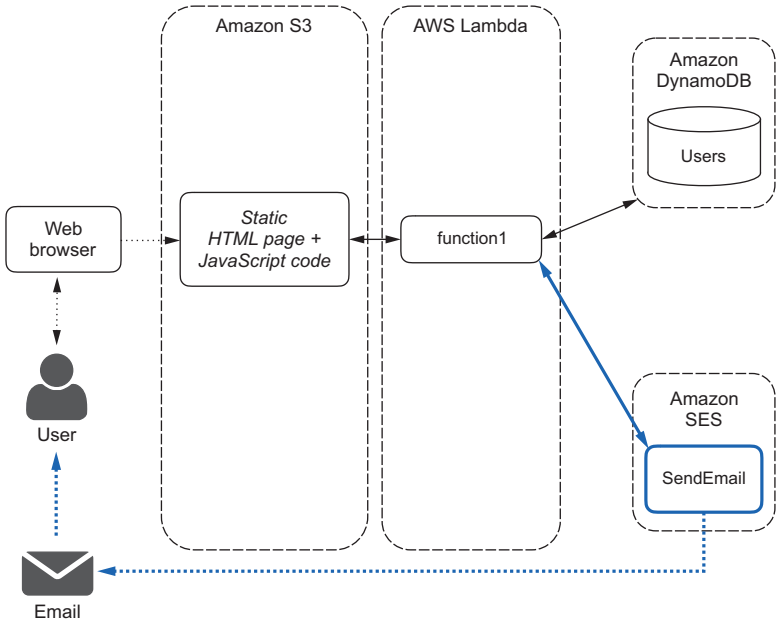


Figure 8.2 Adding the capability for Lambda functions to send emails to the users, via Amazon SES. In this way you can verify the validity of email addresses provided by users.

When a user receives an email sent by Amazon SES, you need a way of interacting with your back end to complete the verification process. To do that, you can include in the body of the email a link to the URL of another static HTML page on Amazon S3. When the user clicks the link, the web browser will open that page and execute the JavaScript code that's embedded in the page. The execution includes the invocation of another Lambda function that can interact with the data stored in Amazon DynamoDB (figure 8.3).

Now that you know how to interact with your users using a web browser or by sending emails, you can design the overall architecture of the authentication service.

8.2 The event-driven architecture

Every static HTML page you put on Amazon S3 can potentially be used as an *interactive step* to engage the user. If you compare this with a native mobile app, each of those pages can behave similarly to an *activity* in Android or a *scene* in iOS.

As the first step, you'll implement a menu of all possible actions users can perform (such as sign-up, login, or change password) and put that in an `index.html` page (figure 8.4). For now, this page doesn't require any client logic, so you have no JavaScript code to execute; it's a list of actions linking to other HTML pages.

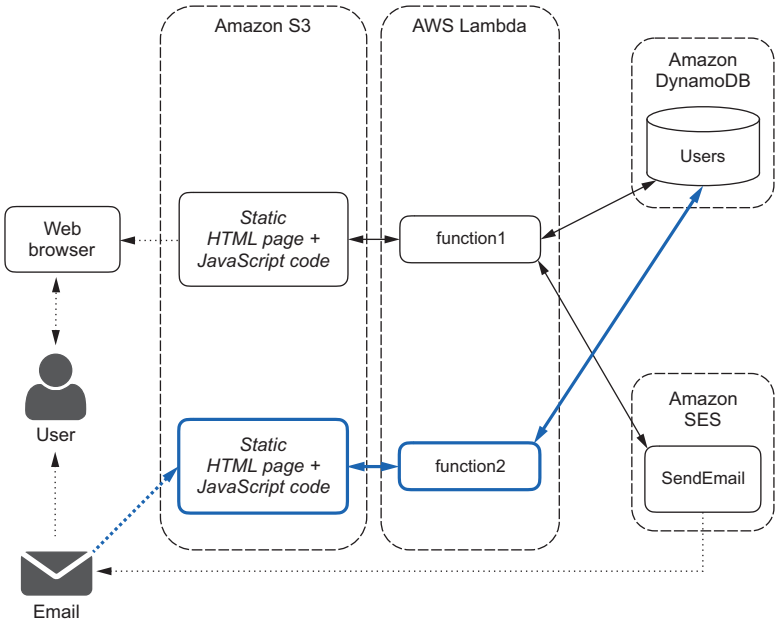


Figure 8.3 Emails received by users can include links to other HTML pages that can execute JavaScript code and invoke other Lambda function to interact with back-end data repositories such as DynamoDB tables.

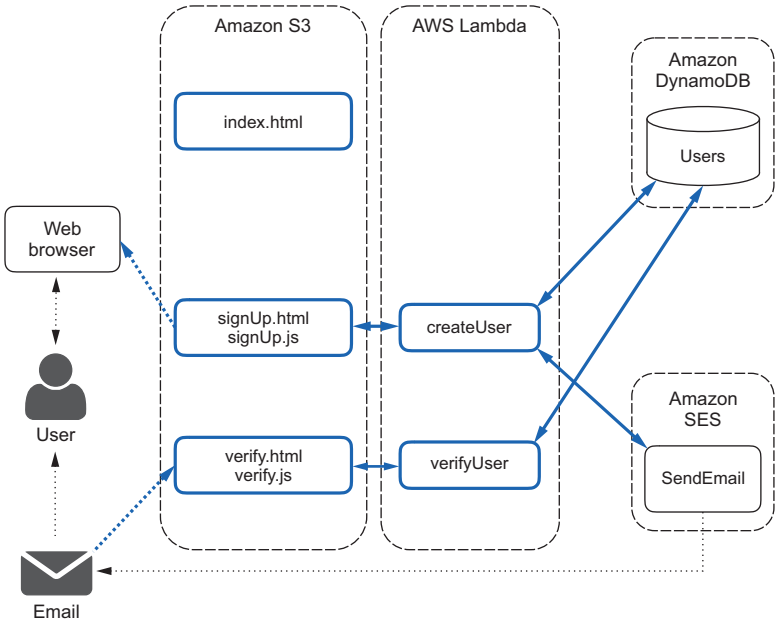


Figure 8.4 The first HTML pages, JavaScript files, and Lambda functions required to sign up new users and verify their email addresses

Next, you'll want users to sign up and create a new account using a `signUp.html` page. This page needs JavaScript code to invoke the `createUser` Lambda function (see figure 8.4).

> **TIP** To simplify separate management of the user interface (in the HTML page) and the client-side login (in the JavaScript code), put the JavaScript code in a separate file, with the same name as the HMTL page, but with the .js extension (for example, signUp.js in this case).

The `createUser` Lambda function takes as input all the information provided by a new user (such as the email and the password) and stores it in the `Users` DynamoDB table. A new user is flagged as unverified on the table because you don't know if the provided email address is correct. To verify that the email address given by the user is valid and that the user can receive emails at that address, the `createUser` function sends an email to the user (via Amazon SES).

The email sent to the user has a link to the `verify.html` page that includes a query parameter with a unique identifier (for example, a `token`) that's randomly generated for that specific user and stored in the `Users` DynamoDB table. For example, the link in the HTML page would be similar to the following:

```
http://some.domain/verify.html?token=<some unique identifier>
```

The JavaScript code in the `verify.html` page can read the unique identifier (`token`) from the URL and send it as input (as part of the event) to the `verifyUser` Lambda function. The function can check the validity of the `token` and change the status of the user to "verified" on the DynamoDB table.

A verified user can log in using the provided credentials (email, password). You can use a `login.html` page and a `login` Lambda function to check in the User table that the user is verified and the credentials are correct (figure 8.5). At first, this function can return the login status as a Boolean value (`true` or `false`). You'll learn later in this chapter how to federate the authentication service you're building with Amazon Cognito as a developer-authenticated identity.

Another important capability is for your users to change their passwords. Changing passwords periodically (for example, every few months) is a good practice to reduce the risk associated with compromised credentials.

You can add a `changePassword.html` page that can use a `changePassword` Lambda function to update credentials in the `Users` DynamoDB table (figure 8.6). But this page is different from others: only an authenticated user can change their own password.

There are two possible implementations that you can use for secure access to the `changePassword` function:

1. Add the current password to the input event of the function, to check the authentication of the user before changing the password.
2. Use Amazon Cognito, via the `login` function, to provide an authenticated status to the user.

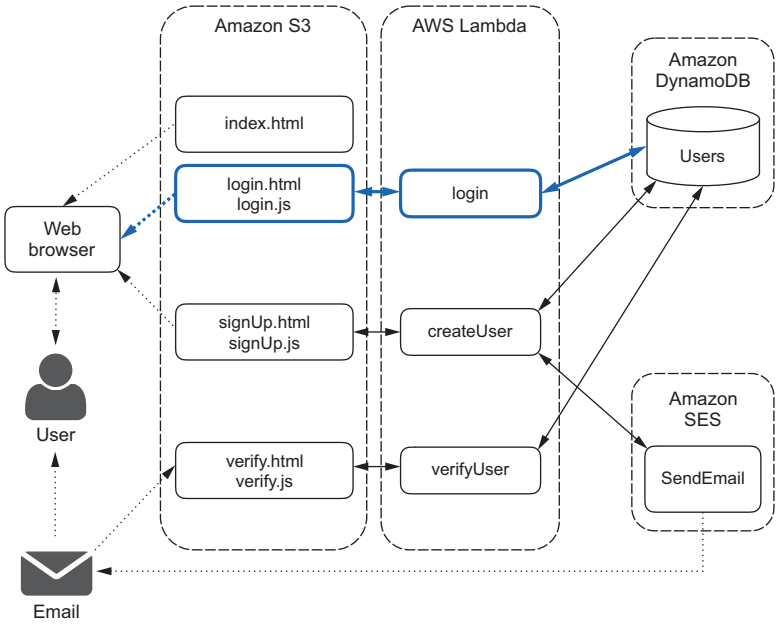


Figure 8.5 Adding a login page to test the provided credentials and the validity of the user in the Users repository

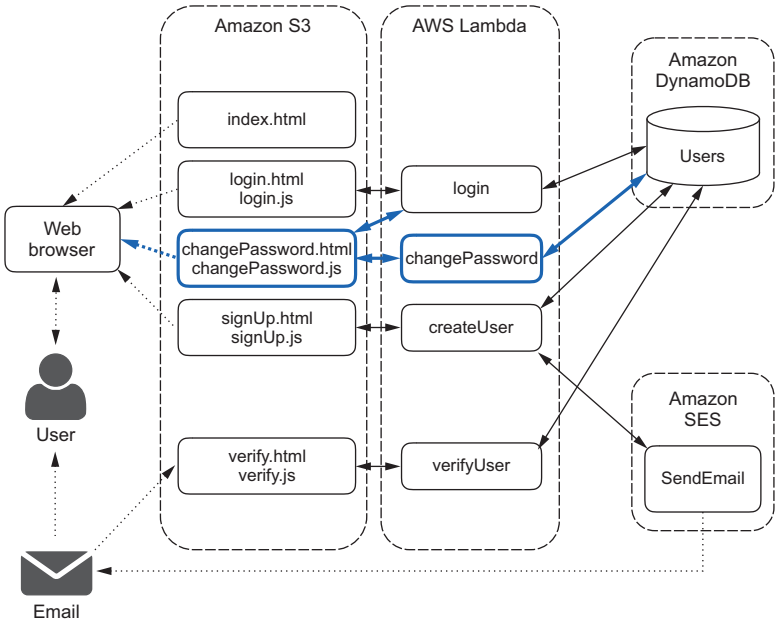


Figure 8.6 The page to allow users to change their passwords is calling a function that must be protected so that only authenticated users can use it.

The first solution is easy to implement (for example, reusing code from the `login` function), but because we're going to federate this authentication service with Amazon Cognito, let's make this example more interesting and go for the second option.

As you may recall, HTML pages need to get AWS credentials from Amazon Cognito to invoke Lambda functions. In all examples so far, we used only unauthenticated users; to allow those users to invoke a Lambda function, we added those functions to the unauthenticated IAM role associated with the Cognito identity pool.

To protect access to the `changePassword` function, you'll add this function to the authenticated IAM role (and not to the unauthenticated role). The same approach will work for any function that needs to be executed by only authenticated users.

Sometimes users need to change passwords because they forgot their current one. In those cases, you can use their email address to validate their request in a way similar to what you did for the initial sign-up: send an email with an embedded link and a unique identifier.

The `lostPassword.html` page is calling a `lostPassword` Lambda function to generate a unique identifier (`resetToken`) that's stored in the `Users` DynamoDB table. The `resetToken` is then sent to the user as a query parameter in a link embedded in a verification email (figure 8.7).

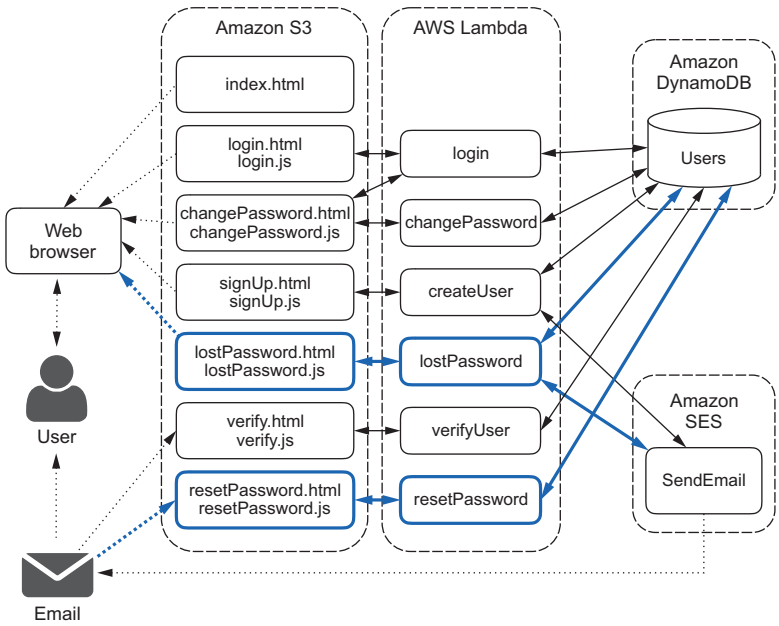


Figure 8.7 In case of a lost password, a lost password page is used to send an email with an embedded link to reset the password. A unique identifier, stored in the DynamoDB table and part of the reset password link, is used to verify that the user making the request to reset the password is the same user who receives the email.

For example, the link can be something similar to the following:

```
http://some.domain/resetPassword?resetToken=<some unique identifier>
```

The user can then open the email and click the link to the `resetPassword.html` page, which will ask for a new password and then call a `resetPassword` Lambda function to check the unique identifier (`resetToken`) in the `Users` DynamoDB table. If the identifier is correct, the function will change the password to the new value.

You've now designed the overall flow and the necessary components to cover the basic functionalities for implementing the authentication service. But before you move into the implementation phase in the next chapter, you'll learn how to *federate* the authentication with Amazon Cognito, and define how to implement other details. By *identity federation* I mean having an authorization service (Amazon Cognito in this case) trusting the authentication of an external service (the sample authentication service you are building).

> **NOTE** Instead of creating multiple Lambda functions, one for each HTML page, you could create a single Lambda function and pass the kind of action (for example `signUp` or `resetPassword`) as part of the input event. You'd have fewer functions to manage (potentially, only one) but the codebase of that function would be larger and more difficult to evolve and extend with further functionalities. Following a microservices approach, my advice is to have multiple smaller functions, each one with a well-defined input/output interface that you can update and deploy separately. However, the right balance between function size and the number of functions to implement depends on your actual use case and programming style. If you need to aggregate multiple functions into a single service call, the Amazon API Gateway is the place to do that instead of the functions themselves.

8.3 Working with Amazon Cognito

To use the authentication service with Amazon Cognito, you need to add to the `login` Lambda function a call to Amazon Cognito to get a token for a developer identity. The `login` function can then return the authentication token for a correct authentication.

The JavaScript code in the page can use that token to authenticate with Amazon Cognito and get AWS temporary credentials for the authenticated role (figure 8.8).

> **WARNING** The AWS credentials returned by Amazon Cognito are temporary and expire after a period of time. You need to manage credential rotation—for example, using the JavaScript `setInterval()` method to periodically call Amazon Cognito to refresh the credentials.

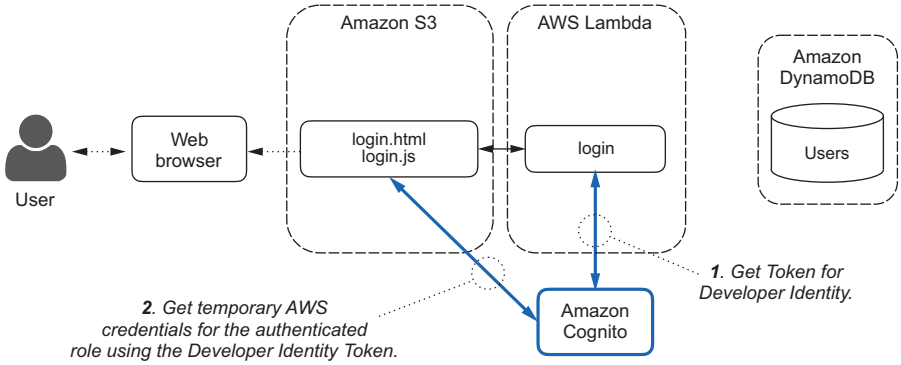


Figure 8.8 Integrating the login function with Cognito Developer Authenticated Identities. The login function gets the token from Amazon Cognito, and then the JavaScript code running in the browser can get AWS Credentials for the authenticated role by passing the token as a login.

8.4 Storing user profiles

To store user profiles, you're using the `Users` DynamoDB table in this sample application. Generally speaking, in a Lambda function you can use any repository reachable via the internet, or that's deployed on AWS in an Amazon Virtual Private Cloud (VPC), or deployed on-premises and connected to an Amazon VPC with a VPN connection. I'm using Amazon DynamoDB because it's a fully-managed NoSQL database service that embraces the serverless approach of this book.

In Amazon DynamoDB, when you create a new table, only the primary key must be declared and must be used in all items in the table. The rest of the table schema is flexible and other attributes can be used (or not) to add more information to any item.

> **NOTE** A DynamoDB item is a collection of attributes, and each attribute has a name and a value. For more details on how to work with items, see https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/WorkingWithItems.html.

The primary key must be unique for an item and can be composed of a single *hash key* (for example, a user ID), or of a hash key together with a *range key* (such as a user ID and a validity date).

For this authentication service, the email of the user is a unique identifier that you can use as hash key, without a range key. If you want to have multiple items for the same users—for example, to keep track of changes and updates in the user profile—you could use a composed primary key with the email as hash key and a validity date in the sort key.

8.5 Adding more data to user profiles

Because Amazon DynamoDB doesn't enforce a schema outside of the primary key, you can freely add more attributes to any item in a table. Different items can have

different attributes. For example, to flag newly created users as unverified, you can add an `unverified` attribute equal to `true`.

When a user email is verified, instead of keeping the `unverified` attribute with a `false` value, you can remove it from the item using the assumption that if the `unverified` attribute isn't present, the user is verified. This approach (that can be easily used with Boolean values) provides a compact and efficient usage of the database storage, especially if you create an index on the `unverified` attribute, because only items with that attribute are part of the index.

Amazon DynamoDB also supports a JSON Document Model, so that the value of an attribute can be a JSON document. In this way, you can further extend the possibility of storing data in a hierarchical and structured way. For example, in the AWS JavaScript SDK, you can use the document client to have native JavaScript data types mapped to and from DynamoDB attributes.

For more information on the document client in the AWS JavaScript SDK, see http://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DynamoDB/DocumentClient.html.

8.6 Encrypting passwords

When managing passwords, certain interactions are critical and must be secured. For example, the following are not secure:

- Storing passwords in plain text in a database table, because any user who has read access to the database table can intercept user credentials
- Sending passwords on an insecure channel, where malicious eavesdropping users can intercept user credentials

For this authentication service, you'll store the password as encrypted using a *salt*. In cryptography, a salt is random data that's generated for each password and used as an additional input to a one-way function that computes a hash of the password that's stored in the user profile, together with the salt:

```
hashingFunction(password, salt) = hash
```

To test the password in a login, the salt is read from the user profile and the same hashing function is used to compare the result with the stored hash. For example,

```
if hashingFunction(inputPassword, salt) == hash then // Logged in...
```

If user profiles are compromised and a malicious user has access to the database content, the use of a salt can protect against dictionary attacks, which use a list of common passwords versus a list of password hashes.

> **TIP** Common hashing functions, used in the past for salting passwords, were MD5 and SHA1, but they've been demonstrated to not be robust enough to protect against specific attacks. You have to check the robustness of a hashing function at the time you use it.

In the login phase, you send the password over a secure channel, because the AWS API, used by the `login.html` page to invoke the `login` Lambda function, is using HTTPS as transport.

> **TIP** This approach is secure enough for a sample implementation, but for a more robust solution you should never send the password as plain text. Use a challenge-response authentication, such as that implemented by the Secure Remote Password (SRP) protocol, used by Amazon Cognito User Pools. For more information on the SRP protocol, see http://srp.stanford.edu.

For a more in-depth analysis of password security in case of remote access, I suggest you to have a look at "Password Security: A Case History" by Robert Morris and Ken Thompson (1978), https://www.bell-labs.com/usr/dmr/www/passwd.ps.

Summary

In this chapter you designed the overall architecture of your first event-driven application, a sample authentication service using AWS lambda to implement the back-end logic.

In particular, you learned how to do the following:

- Interact with a client application via a static HTML page using JavaScript
- Differentiate between authenticated and unauthenticated access
- Send emails and interact using custom links in the email body
- Map application functionality to different components in the architecture
- Federate the custom authentication service with Amazon Cognito
- Use Amazon DynamoDB to store user profiles
- Use encryption to protect passwords from being intercepted and compromised

In the next chapter, you'll implement this sample authentication service.

EXERCISE

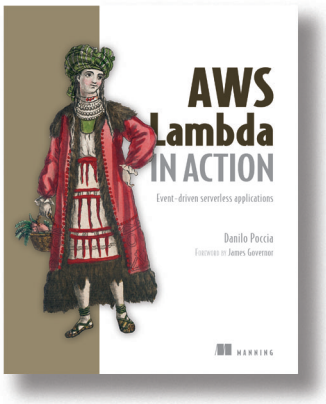
1. To send an email from a web page, you can
   a. Use JavaScript in the browser to use SMTP
   b. Use JavaScript in the browser to use IMAP
   c. Use a Lambda function to call Amazon SES
   d. Use a Lambda function to call Amazon SQS
2. To give access to a Lambda function only to authenticated users coming from a web or mobile app, you can
   a. Use AWS IAM users and groups to give access to the function to authenticated users only
   b. Use Amazon Cognito and give access to the function to the authenticated role only

c. Use AWS IAM users and groups to give access to the function to unauthenticated users only
d. Use Amazon Cognito and give access to the function to the unauthenticated role only

3. The most secure way to validate a user password with a login service is to
a. Use a challenge-response interface such as CAPTCHA
b. Send the password over HTTP
c. Use a challenge-response protocol such as SRP
d. Send the password via email

Solution

1. c
2. b
3. c



With AWS Lambda, you write your code and upload it to the AWS cloud. AWS Lambda responds to the events triggered by your application or your users, and automatically manages the underlying computer resources for you. Back-end tasks like analyzing a new document or processing requests from a mobile app are easy to implement. Your application is divided into small functions, leading naturally to a reactive architecture and the adoption of microservices.

AWS Lambda in Action is an example-driven tutorial that teaches you how to build applications that use an event-driven approach on the back-end. Starting with an overview of AWS Lambda, the book moves on to show you common examples and patterns that you can use to call Lambda functions from a web page or a mobile app. The second part of the book puts these smaller examples together to build larger applications. By the end, you'll be ready to create applications that take advantage of the high availability, security, performance, and scalability of AWS.

What's inside

- Create a simple API
- Create an event-driven media-sharing application
- Secure access to your application in the cloud
- Use functions from different clients like web pages or mobile apps
- Connect your application with external services

Requires basic knowledge of JavaScript. Some examples are also provided in Python. No AWS experience is assumed.

Automating Deployment: CloudFormation, Elastic Beanstalk, and OpsWorks

This chapter shows you where the serverless + agile approach really pays off. You'll look at an automated deployment pipeline built using AWS services and CloudFormation templates. The process shown here is instantly useful to AWS devs and ops engineers, and it illustrates an agile approach that you can apply in any cloud-based system (e.g. Azure or Google Cloud Platform).

Chapter 5 from *Amazon Web Services in Action* by Michael Wittig and Andreas Wittig

Automating deployment: CloudFormation, Elastic Beanstalk, and OpsWorks

This chapter covers

- Running a script on server startup to deploy applications
- Deploying common web applications with the help of AWS Elastic Beanstalk
- Deploying multilayer applications with the help of AWS OpsWorks
- Comparing the different deployment services available on AWS

Whether you want to use software from in-house development, open source projects, or commercial vendors, you need to install, update, and configure the application and its dependencies. This process is called *deployment*. In this chapter, you'll learn about three tools for deploying applications to virtual servers on AWS:

- Deploying a VPN solution with the help of AWS CloudFormation and a script started at the end of the boot process.
- Deploying a collaborative text editor with AWS Elastic Beanstalk. The text editor Etherpad is a simple web application and a perfect fit for AWS Elastic Beanstalk, because the Node.js platform is supported by default.
- Deploying an IRC web client and IRC server with AWS OpsWorks. The setup consists of two parts: a Node.js server that delivers the IRC web client and the IRC server itself. The example consists of multiple layers and is perfect for AWS OpsWorks.

We've chosen examples that don't need a storage solution for this chapter, but all three deployment solutions would support delivering an application together with a storage solution. You'll find examples using storage in the next part of the book.

Examples are 100% covered by the Free Tier

The examples in this chapter are completely covered by the Free Tier. As long as you don't run the examples for longer than a few days, you won't pay anything. Keep in mind that this only applies if you created a fresh AWS account for this book and nothing else is going on in your AWS account. Try to complete the examples of the chapter within a few days; you'll clean up your account at the end of each example.

Which steps are required to deploy a typical web application like WordPress—a widely used blogging platform—to a server?

1. Install an Apache HTTP server, a MySQL database, a PHP runtime environment, a MySQL library for PHP, and an SMTP mail server.
2. Download the WordPress application and unpack the archive on your server.
3. Configure the Apache web server to serve the PHP application.
4. Configure the PHP runtime environment to tweak performance and increase security.
5. Edit the wp-config.php file to configure the WordPress application.
6. Edit the configuration of the SMTP server, and make sure mail can only be sent from the virtual server to avoid misuse from spammers.
7. Start the MySQL, SMTP, and HTTP services.

Steps 1–2 handle installing and updating the executables. These executables are configured in steps 3–6. Step 7 starts the services.

System administrators often perform these steps manually by following how-tos. Deploying applications manually is no longer recommended in a flexible cloud environment. Instead your goal will be to automate these steps with the help of the tools you'll discover next.

5.1 Deploying applications in a flexible cloud environment

If you want to use cloud advantages like scaling the number of servers depending on the current load or building a highly available infrastructure, you'll need to start new virtual servers several times a day. On top of that, the number of virtual servers you'll have to supply with updates will grow. The steps required to deploy an application don't change, but as figure 5.1 shows, you need to perform them on multiple servers. Deploying software manually to a growing number of servers becomes impossible over time and has a high risk of human failure. This is why we recommend that you automate the deployment of applications.

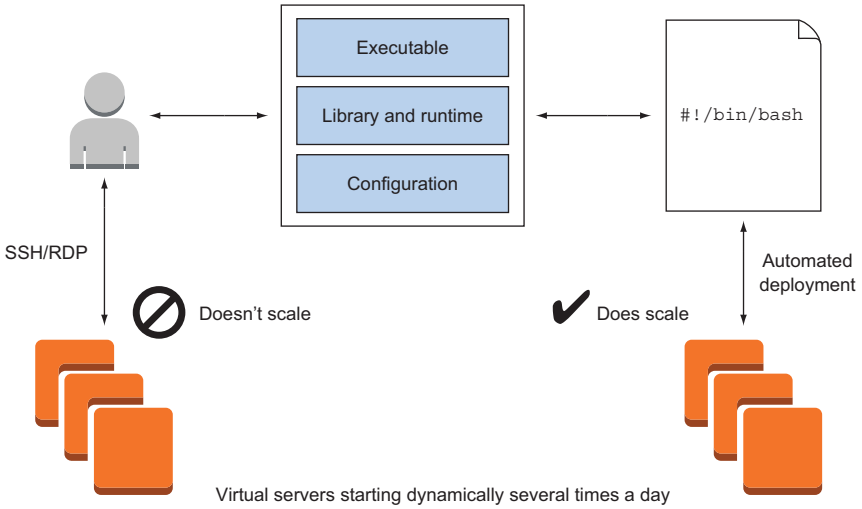


Figure 5.1 Deployment must be automated in a flexible and scalable cloud environment.

The investment in an automated deployment process will pay off in the future by increasing efficiency and decreasing human failures.

5.2 Running a script on server startup using CloudFormation

A simple but powerful and flexible way of automating application deployment is to run a script on server startup. To go from a plain OS to a fully installed and configured server, you need to follow these three steps:

1. Start a plain virtual server containing just an OS.
2. Execute a script at the end of the boot process.
3. Install and configure your applications with the help of a script.

First you need to choose an AMI from which to start your virtual server. An AMI bundles the OS and preinstalled software for your virtual server. When you're starting your server from an AMI containing a plain OS without any additional software installed, you need to provision the virtual server at the end of the boot process. Translating the

necessary steps to install and configure your application into a script allows you to automate this task. But how do you execute this script automatically after booting your virtual server?

5.2.1 Using user data to run a script on server startup

You can inject a small amount—not more than 16 KB—of data called *user data* into every virtual server. You specify the user data during the creation of a new virtual server. A typical way of using the user data feature is built into most AMIs, such as the Amazon Linux Image and the Ubuntu AMI. Whenever you boot a virtual server based on these AMIs, user data is executed as a shell script at the end of the boot process. The script is executed as user root.

The user data is always accessible from the virtual server with a HTTP `GET` request to http://169.254.169.254/latest/user-data. The user data behind this URL is only accessible from the virtual server itself. As you'll see in the following example, you can deploy applications of any kind with the help of user data executed as a script.

5.2.2 Deploying OpenSwan as a VPN server to a virtual server

If you're working with a laptop from a coffee house over Wi-Fi, you may want to tunnel your traffic to the internet through a VPN. You'll learn how to deploy a VPN server to a virtual server with the help of user data and a shell script. The VPN solution, called OpenSwan, offers an IPSec-based tunnel that's easy to use with Windows, OS X, and Linux. Figure 5.2 shows the example setup.

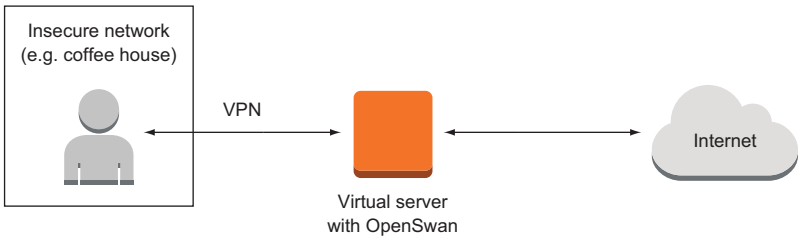


Figure 5.2 Using OpenSwan on a virtual server to tunnel traffic from a personal computer

Open your command line and execute the commands shown in the next listing step by step to start a virtual server and deploy a VPN server on it. We've prepared a CloudFormation template that starts the virtual server and its dependencies.

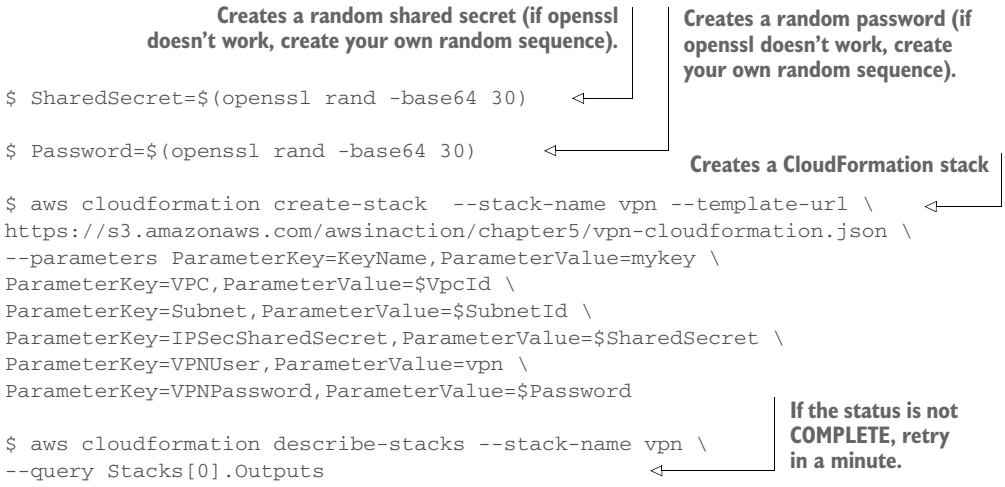
Listing 5.1 Deploying a VPN server to a virtual server: CloudFormation and a shell script

```
$ VpcId=$(aws ec2 describe-vpcs --query Vpcs[0].VpcId --output text)

$ SubnetId=$(aws ec2 describe-subnets --filters Name=vpc-id,Values=$VpcId \
--query Subnets[0].SubnetId --output text)
```

Gets the default VPC

Gets the default subnet




```
$ SharedSecret=$(openssl rand -base64 30)

$ Password=$(openssl rand -base64 30)

$ aws cloudformation create-stack  --stack-name vpn --template-url \
https://s3.amazonaws.com/awsinaction/chapter5/vpn-cloudformation.json \
--parameters ParameterKey=KeyName,ParameterValue=mykey \
ParameterKey=VPC,ParameterValue=$VpcId \
ParameterKey=Subnet,ParameterValue=$SubnetId \
ParameterKey=IPSecSharedSecret,ParameterValue=$SharedSecret \
ParameterKey=VPNUser,ParameterValue=vpn \
ParameterKey=VPNPassword,ParameterValue=$Password

$ aws cloudformation describe-stacks --stack-name vpn \
--query Stacks[0].Outputs
```

Shortcut for OS X and Linux

You can avoid typing these commands manually at your command line by using the following command to download a bash script and execute it directly on your local machine. The bash script contains the same steps as shown in listing 5.1:

```
$ curl -s https://raw.githubusercontent.com/AWSinAction/\
code/master/chapter5/\
vpn-create-cloudformation-stack.sh | bash -ex
```

The output of the last command should print out the public IP address of the VPN server, a shared secret, the VPN username, and the VPN password. You can use this information to establish a VPN connection from your computer, if you like:

```
[...]
[
  {
    "Description": "Public IP address of the vpn server",
    "OutputKey": "ServerIP",
    "OutputValue": "52.4.68.225"
  },
  {
    "Description": "The shared key for the VPN connection (IPSec)",
    "OutputKey": "IPSecSharedSecret",
    "OutputValue": "sqmvJ1l/13bD6YqpmsKkPSMs9RrPL8itpr7m5V8g"
  },
  {
    "Description": "The username for the vpn connection",
```

```
    "OutputKey": "VPNUser",
    "OutputValue": "vpn"
  },
  {
    "Description": "The password for the vpn connection",
    "OutputKey": "VPNPassword",
    "OutputValue": "aZQVFufFlUjJkesUfDmMj6DcHrWjuKShyFB/d0lE"
  }
]
```

Let's take a deeper look at the deployment process of the VPN server. You'll dive into the following tasks, which you've used unnoticed so far:

- Starting a virtual server with custom user data and configuring a firewall for the virtual server with AWS CloudFormation
- Executing a shell script at the end of the boot process to install an application and its dependencies with the help of a package manager, and to edit configuration files

USING CLOUDFORMATION TO START A VIRTUAL SERVER WITH USER DATA

You can use CloudFormation to start a virtual server and configure a firewall. The template for the VPN server includes a shell script packed into user data, as shown in listing 5.2.

Fn::Join and Fn::Base64

The CloudFormation template includes two new functions: `Fn::Join` and `Fn::Base64`. With `Fn::Join`, you can join a set of values to make a single value with a specified delimiter:

```
{"Fn::Join": ["delimiter", ["value1", "value2", "value3"]]}
```

The function `Fn::Base64` encodes the input with Base64. You'll need this function because the user data must be encoded in Base64:

```
{"Fn::Base64": "value"}
```

Listing 5.2 Parts of a CloudFormation template to start a virtual server with user data

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "Starts an virtual server (EC2) with OpenSwan [...]",
  "Parameters": {
    "KeyName": {
      "Description": "key for SSH access",
      "Type": "AWS::EC2::KeyPair::KeyName"
    },
    "VPC": {
```

Parameters to make it possible to reuse the template

```
      "Description": "Just select the one and only default VPC.",
      "Type": "AWS::EC2::VPC::Id"
    },
    "Subnet": {
      "Description": "Just select one of the available subnets.",
      "Type": "AWS::EC2::Subnet::Id"
    },
    "IPSecSharedSecret": {
      "Description": "The shared secret key for IPSec.",
      "Type": "String"
    },
    "VPNUser": {
      "Description": "The VPN user.",
      "Type": "String"
    },
    "VPNPassword": {
      "Description": "The VPN password.",
      "Type": "String"
    }
  },
  "Resources": {
    "EC2Instance": {
      "Type": "AWS::EC2::Instance",
      "Properties": {
        "InstanceType": "t2.micro",
        "SecurityGroupIds": [{"Ref": "InstanceSecurityGroup"}],
        "KeyName": {"Ref": "KeyName"},
        "ImageId": "ami-1ecae776",
        "SubnetId": {"Ref": "Subnet"},
        "UserData":
          {"Fn::Base64": {"Fn::Join": ["", [
            "#!/bin/bash -ex\n",
            "export IPSEC_PSK=", {"Ref": "IPSecSharedSecret"}, "\n",
            "export VPN_USER=", {"Ref": "VPNUser"}, "\n",
            "export VPN_PASSWORD=", {"Ref": "VPNPassword"}, "\n",
            "export STACK_NAME=", {"Ref": "AWS::StackName"}, "\n",
            "export REGION=", {"Ref": "AWS::Region"}, "\n",
            "curl -s https://…/vpn-setup.sh | bash -ex\n"
          ]]}}
      },
      [...]
    },
    [...]
  },
  "Outputs": {
    [...]
  }
}
```

Describes the virtual server

Concatenates and encodes a string value

Defines a shell script as user data for the virtual server

Exports parameters to environment variables to make them available in an external shell script called next

Fetches the shell script via HTTP and executes it

Basically, the user data contains a small script to fetch and execute the real script, vpn-setup.sh, which contains all the commands for installing the executables and configuring the services. Doing so frees you from inserting scripts in the unreadable format needed for the JSON CloudFormation template.

INSTALLING AND CONFIGURING A VPN SERVER WITH A SCRIPT

The vpn-setup.sh script shown in listing 5.3 installs packages with the help of the package manager `yum` and writes some configuration files. You don't have to understand the details of the configuration of the VPN server; you just need to know that this shell script is executed during the boot process to install and configure a VPN server.

Listing 5.3 Installing packages and writing configuration files on server startup

```
#!/bin/bash -ex

[...]

PRIVATE_IP=`curl -s http://169.254.169.254/latest/meta-data/local-ipv4`

PUBLIC_IP=`curl -s http://169.254.169.254/latest/meta-data/public-ipv4`

yum-config-manager --enable epel && yum clean all

yum install -y openswan xl2tpd

cat > /etc/ipsec.conf <<EOF
[...]
EOF

cat > /etc/ipsec.secrets <<EOF
$PUBLIC_IP %any : PSK "${IPSEC_PSK}"
EOF

cat > /etc/xl2tpd/xl2tpd.conf <<EOF
[...]
EOF

cat > /etc/ppp/options.xl2tpd <<EOF
[...]
EOF

service ipsec start && service xl2tpd start

chkconfig ipsec on && chkconfig xl2tpd on
```

Fetches the private IP address of the virtual server

Fetches the public IP address of the virtual server

Installs software packages

Adds extra packages to the package manager yum

Writes a configuration file for IPSec (OpenSwan)

Writes a file containing the shared secret for IPSec

Writes a configuration file for the L2TP tunnel

Writes a configuration file for the PPP service

Starts the services need for the VPN server

Configures the runlevel for the VPN services

That's it. You've learned how to deploy a VPN server to a virtual server with the help of EC2 user data and a shell script. After you terminate your virtual server, you'll be ready to learn how to deploy a common web application without writing a custom script.

Cleaning up

You've reached the end of the VPN server example. Don't forget to terminate your virtual server and clean up your environment. To do so, enter `aws cloudformation delete-stack --stack-name vpn` at your terminal.

5.2.3 Starting from scratch instead of updating

You've learned how to deploy an application with the help of user data in this section. The script from the user data is executed at the end of the boot process. But how do you update your application with this approach?

You've automated the installation and configuration of software during the boot process of your virtual server, so you can start a new virtual server without any extra effort. If you have to update your application or its dependencies, you can do so with the following steps:

1. Make sure the up-to-date version of your application or software is available through the package repository of your OS, or edit the user data script.
2. Start a new virtual server based on your CloudFormation template and user data script.
3. Test the application deployed to the new virtual server. Proceed with the next step if everything works as it should.
4. Switch your workload to the new virtual server (for example, by updating a DNS record).
5. Terminate the old virtual server, and throw away its unused dependencies.

5.3 Deploying a simple web application with Elastic Beanstalk

It isn't necessary to reinvent the wheel if you have to deploy a common web application. AWS offers a service that can help you to deploy web applications based on PHP, Java, .NET, Ruby, Node.js, Python, Go, and Docker; it's called *AWS Elastic Beanstalk*. With Elastic Beanstalk, you don't have to worry about your OS or virtual servers because it adds another layer of abstraction on top of them.

Elastic Beanstalk lets you handle the following recurring problems:

- Providing a runtime environment for a web application (PHP, Java, and so on)
- Installing and updating a web application automatically
- Configuring a web application and its environment
- Scaling a web application to balance load
- Monitoring and debugging a web application

5.3.1 Components of Elastic Beanstalk

Getting to know the different components of Elastic Beanstalk will help you to understand its functionality. Figure 5.3 shows these elements:

- An *application* is a logical container. It contains versions, environments, and configurations. If you start to use Elastic Beanstalk in a region, you have to create an application first.
- A *version* contains a specific version of your application. To create a new version, you have to upload your executables (packed into an archive) to the service

Amazon S3, which stores static files. A version is basically a pointer to this archive of executables.

- A *configuration template* contains your default configuration. You can manage your application's configuration (such as the port your application listens on) as well as the environment's configuration (such as the size of the virtual server) with your custom configuration template.
- An *environment* is where Elastic Beanstalk executes your application. It consists of a *version* and the *configuration*. You can run multiple environments for one application using the versions and configurations multiple times.

Enough theory for the moment. Let's proceed with deploying a simple web application.

5.3.2 Using Elastic Beanstalk to deploy Etherpad, a Node.js application

Editing a document collaboratively can be painful with the wrong tools. *Etherpad* is an open source online editor that lets you edit a document with many people in real time. You'll deploy this Node.js-based application with the help of Elastic Beanstalk in three steps:

1. Create an application: the logical container.
2. Create a version: a pointer to a specific version of Etherpad.
3. Create an environment: the place where Etherpad will run.

CREATING AN APPLICATION FOR AWS ELASTIC BEANSTALK

Open your command line and execute the following command to create an application for the Elastic Beanstalk service:

```
$ aws elasticbeanstalk create-application --application-name etherpad
```

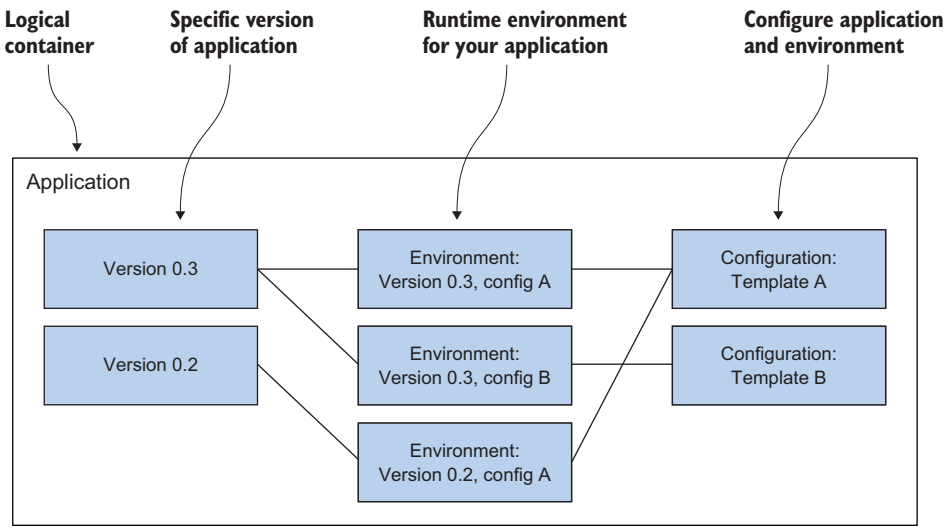


Figure 5.3 An Elastic Beanstalk application consists of versions, configurations, and environments.

You've created a container for all the other components that are necessary to deploy Etherpad with the help of AWS Elastic Beanstalk.

Creating a version for AWS Elastic Beanstalk

You can create a new version of your Etherpad application with the following command:

```
$ aws elasticbeanstalk create-application-version \
--application-name etherpad --version-label 1.5.2 \
--source-bundle S3Bucket=awsinaction,S3Key=chapter5/etherpad.zip
```

For this example, we uploaded a zip archive containing version 1.5.2 of Etherpad. If you want to deploy another application, you can upload your own application to the AWS S3 service for static files.

Creating an environment to execute Etherpad with Elastic Beanstalk

To deploy Etherpad with the help of Elastic Beanstalk, you have to create an environment for Node.js based on Amazon Linux and the version of Etherpad you just created. To get the latest Node.js environment version, called a *solution stack name*, run this command:

```
$ aws elasticbeanstalk list-available-solution-stacks --output text \
--query "SolutionStacks[?contains(@, 'running Node.js')] | [0]"\
64bit Amazon Linux 2015.03 v1.4.6 running Node.js
```

The option `EnvironmentType = SingleInstance` launches a single virtual server without the ability to scale and load-balance automatically. Replace `$SolutionStackName` with the output from the previous command:

```
$ aws elasticbeanstalk create-environment --environment-name etherpad \
--application-name etherpad \
--option-settings Namespace=aws:elasticbeanstalk:environment,\
OptionName=EnvironmentType,Value=SingleInstance \
--solution-stack-name "$SolutionStackName" \
--version-label 1.5.2
```

Having fun with Etherpad

You've created an environment for Etherpad. It will take several minutes before you can point your browser to your Etherpad installation. The following command helps you track the state of your Etherpad environment:

```
$ aws elasticbeanstalk describe-environments --environment-names etherpad
```

If Status turns to Ready and Health turns to Green, you're ready to create your first Etherpad document. The output of the `describe` command should look similar to the following example.

Listing 5.4 Describing the status of the Elastic Beanstalk environment

```
{
  "Environments": [{
    "ApplicationName": "etherpad",
    "EnvironmentName": "etherpad",
    "VersionLabel": "1",
    "Status": "Ready",
    "EnvironmentId": "e-pwbfmgrsjp",
    "EndpointURL": "23.23.223.115",
    "SolutionStackName": "64bit Amazon Linux 2015.03 v1.4.6 running Node.js",
    "CNAME": "etherpad-cxzshvfjzu.elasticbeanstalk.com",
    "Health": "Green",
    "Tier": {
      "Version": " ",
      "Type": "Standard",
      "Name": "WebServer"
    },
    "DateUpdated": "2015-04-07T08:45:07.658Z",
    "DateCreated": "2015-04-07T08:40:21.698Z"
  }]
}
```

Wait until Status turns to Ready.

Wait until Health turns to Green.

DNS record for the environment (for example, to open with a browser)

You've deployed a Node.js web application to AWS with three commands. Point your browser to the URL shown in `CNAME` and open a new document by typing in a name for it and clicking the OK button. Figure 5.4 shows an Etherpad document in action.

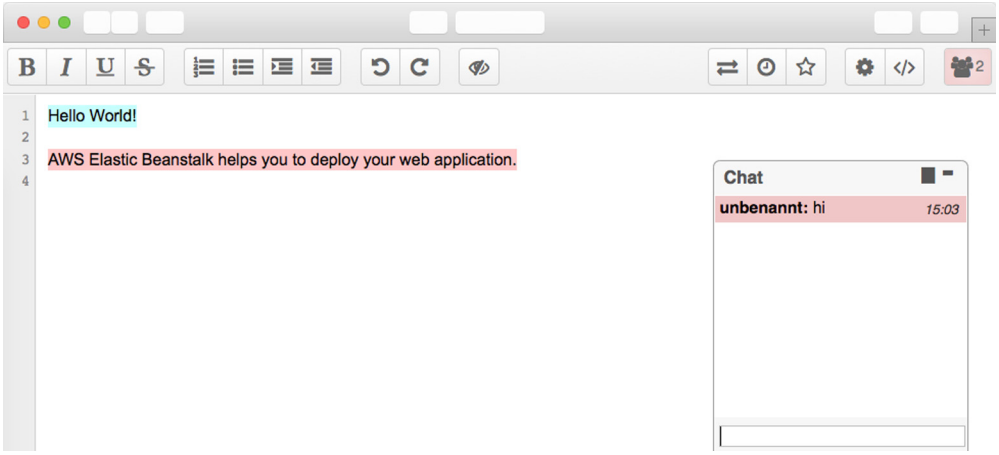


Figure 5.4 Online text editor Etherpad in action

Exploring Elastic Beanstalk with the Management Console

You've deployed Etherpad with the help of Elastic Beanstalk and the AWS command-line interface (CLI) by creating an application, a version, and an environment. You can also control the Elastic Beanstalk service with the help of the Management Console, a web-based user interface:

1. Open the AWS Management Console at https://console.aws.amazon.com.
2. Click Services in the navigation bar, and click the Elastic Beanstalk service.
3. Click the etherpad environment, represented by a green box. An overview of the Etherpad application is shown, as in figure 5.5.

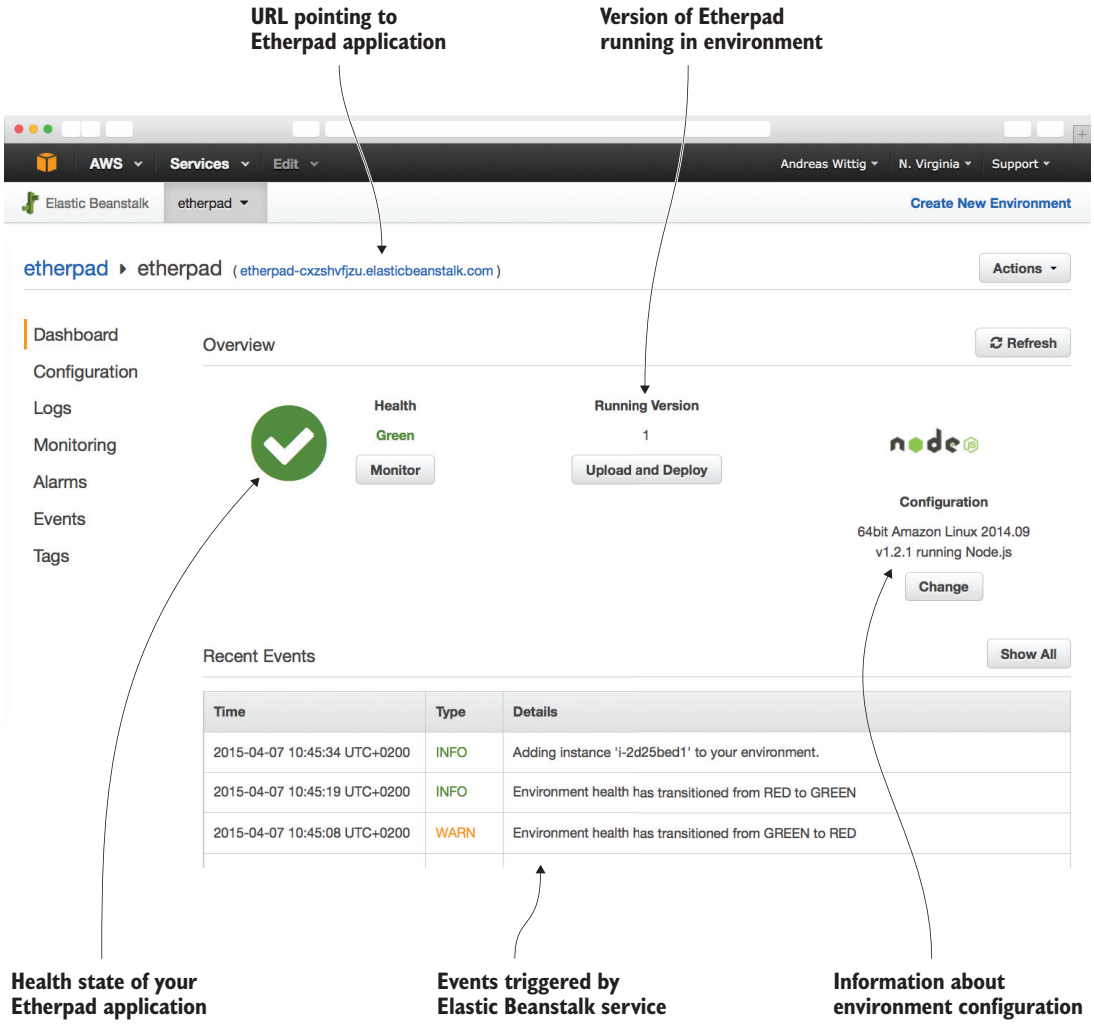


Figure 5.5 Overview of AWS Elastic Beanstalk environment running Etherpad

You can also fetch the log messages from your application with the help of Elastic Beanstalk. Download the latest log messages with the following steps:

1. Choose Logs from the submenu. You'll see a screen like that shown in figure 5.6.
2. Click Request Logs, and choose Last 100 Lines.
3. After a few seconds, a new entry will appear in the table. Click Download to download the log file to your computer.

Cleaning up

Now that you've successfully deployed Etherpad with the help of AWS Elastic Beanstalk and learned about the service's different components, it's time to clean up. Run the following command to terminate the Etherpad environment:

```
$ aws elasticbeanstalk terminate-environment --environment-name etherpad
```

You can check the state of the environment by executing the following command:

```
$ aws elasticbeanstalk describe-environments --environment-names etherpad
```

Wait until Status has changed to Terminated, and then proceed with the following command:

```
$ aws elasticbeanstalk delete-application --application-name etherpad
```

That's it. You've terminated the virtual server providing the environment for Etherpad and deleted all components of Elastic Beanstalk.

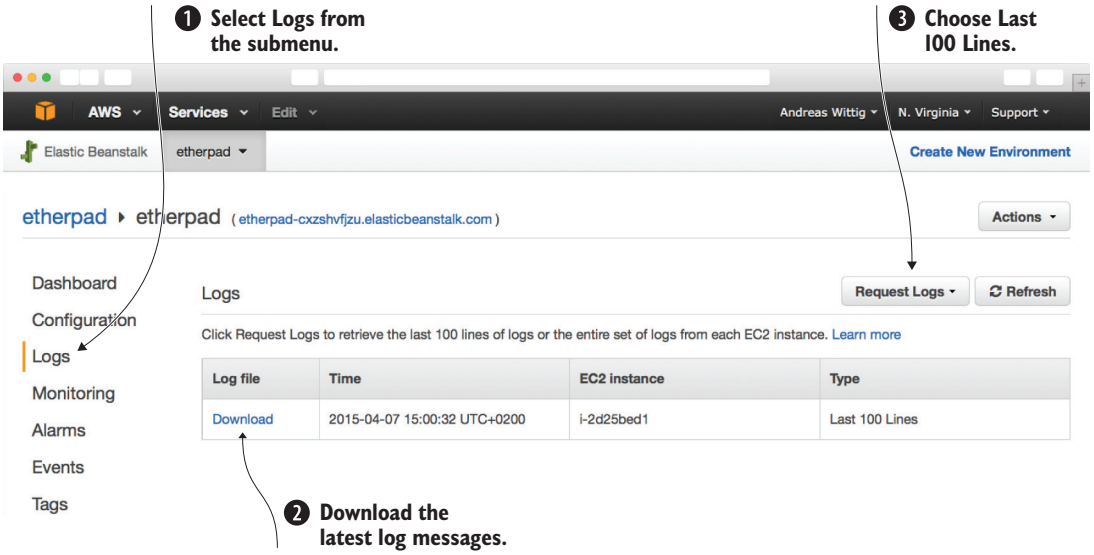


Figure 5.6 Downloading logs from a Node.js application via AWS Elastic Beanstalk

5.4 Deploying a multilayer application with OpsWorks

Deploying a basic web application with the help of Elastic Beanstalk is convenient. But if you have to deploy a more complex application consisting of different services—also called *layers*—you'll reach the limits of Elastic Beanstalk. In this section, you'll learn about AWS OpsWorks, a free service offered by AWS that can help you to deploy a multilayer application.

OpsWorks helps you control AWS resources like virtual servers, load balancers, and databases and lets you deploy applications. The service offers some standard layers with the following runtimes:

- HAProxy (load balancer)
- Rails app server (Ruby on Rails)
- Static web server
- PHP app server
- Java app server (Tomcat server)
- AWS Flow (Ruby)
- MySQL (database)
- Memcached (in-memory cache)
- Ganglia (monitoring)

You can also add a custom layer to deploy anything you want. The deployment is controlled with the help of *Chef*, a configuration-management tool. Chef uses *recipes* organized in *cookbooks* to deploy applications to any kind of system. You can adopt the standard recipes or create your own.

About Chef

Chef is a configuration-management tool like Puppet, SaltStack, and Ansible. Chef transforms templates (recipes) written in a domain-specific language (DSL) into actions, to configure and deploy applications. A recipe can include packages to install, services to run, or configuration files to write, for example. Related recipes can be combined into cookbooks. Chef analyzes the status quo and changes resources where necessary to reach the described state from the recipe.

You can reuse cookbooks and recipes from others with the help of Chef. The community publishes a variety of cookbooks and recipes at https://supermarket.chef.io under open source licenses.

Chef can be run in solo or client/server mode. It acts as a fleet-management tool in client/server mode. This can help if you have to manage a distributed system consisting of many virtual servers. In solo mode, you can execute recipes on a single virtual server. AWS OpsWorks uses solo mode integrated in its own fleet management without needing to configure and operate a setup in client/server mode.

In addition to letting you deploy applications, OpsWorks helps you to scale, monitor and update your virtual servers running beneath the different layers.

5.4.1 Components of OpsWorks

Getting to know the different components of OpsWorks will help you understand its functionality. Figure 5.7 shows these elements:

- A *stack* is a container for all other components of OpsWorks. You can create one or more stacks and add one or more layers to each stack. You could use different stacks to separate the production environment from the testing environment, for example. Or you could use different stacks to separate different applications.
- A *layer* belongs to a stack. A layer represents an application; you can also call it a service. OpsWorks offers predefined layers for standard web applications like PHP and Java, but you're free to use a custom stack for any application you can think of. A layer is responsible for configuring and deploying software to virtual servers. You can add one or multiple virtual servers to a layer. The virtual servers are called *instances* in this context.
- An *instance* is the representation for a virtual server. You can launch one or multiple instances for each layer. You can use different versions of Amazon Linux and Ubuntu or a custom AMI as a basis for the instances, and you can specify rules for launching and terminating instances based on load or timeframes for scaling.
- An *app* is the software you want to deploy. OpsWorks deploys your app to a suitable layer automatically. You can fetch apps from a Git or Subversion repository or as archives via HTTP. OpsWorks helps you to install and update your apps onto one or multiple instances.

Let's look at how to deploy a multilayer application with the help of OpsWorks.

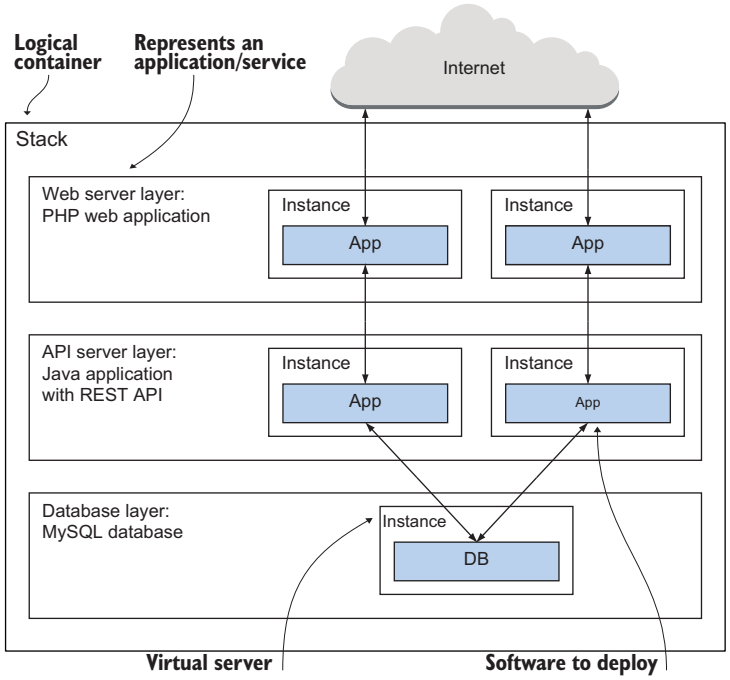


Figure 5.7 Stacks, layers, instances, and apps are the main components of OpsWorks.

5.4.2 Using OpsWorks to deploy an IRC chat application

Internet Relay Chat (IRC) is still a popular means of communication. In this section, you'll deploy *kiwiIRC*, a web-based IRC client, and your own IRC server. Figure 5.8 shows the setup of a distributed system consisting of a web application delivering the IRC client and an IRC server.

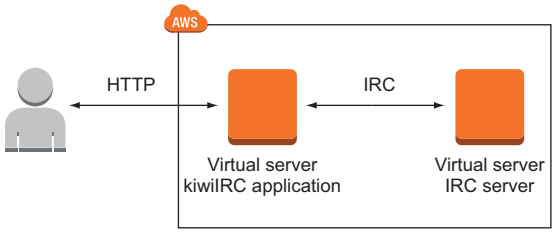


Figure 5.8 Building your own IRC infrastructure consisting of a web application and an IRC server

kiwiIRC is an open source web application written in JavaScript for Node.js. The following steps are necessary to deploy a two-layer application with the help of OpsWorks:

1. Create a stack, the container for all other components.
2. Create a Node.js layer for kiwiIRC.
3. Create a custom layer for the IRC server.
4. Create an app to deploy kiwiIRC to the Node.js layer.
5. Add an instance for each layer.

You'll learn how to handle these steps with the Management Console. You can also control OpsWorks from the command line, as you did Elastic Beanstalk, or with CloudFormation.

Creating a new OpsWorks stack

Open the Management Console at https://console.aws.amazon.com/opsworks, and create a new stack. Figure 5.9 illustrates the necessary steps:

1. Click Add Stack under Select Stack or Add Your First Stack.
2. For Name, type in `irc`.
3. For Region, choose US East (N. Virginia).
4. The default VPC is the only one available. Select it.
5. For Default Subnet, select us-east-1a.
6. For Default Operating System, choose Ubuntu 14.04 LTS.
7. For Default Root Device Type, select EBS Backed.
8. For IAM Role, choose New IAM Role. Doing so automatically creates the necessary dependency.
9. Select your SSH key, mykey, for Default SSH Key.
10. For Default IAM Instance Profile, choose New IAM Instance Profile. Doing so automatically creates the necessary dependency.
11. For Hostname Theme, choose Layer Dependent. Your virtual servers will be named depending on their layer.
12. Click Add Stack to create the stack.

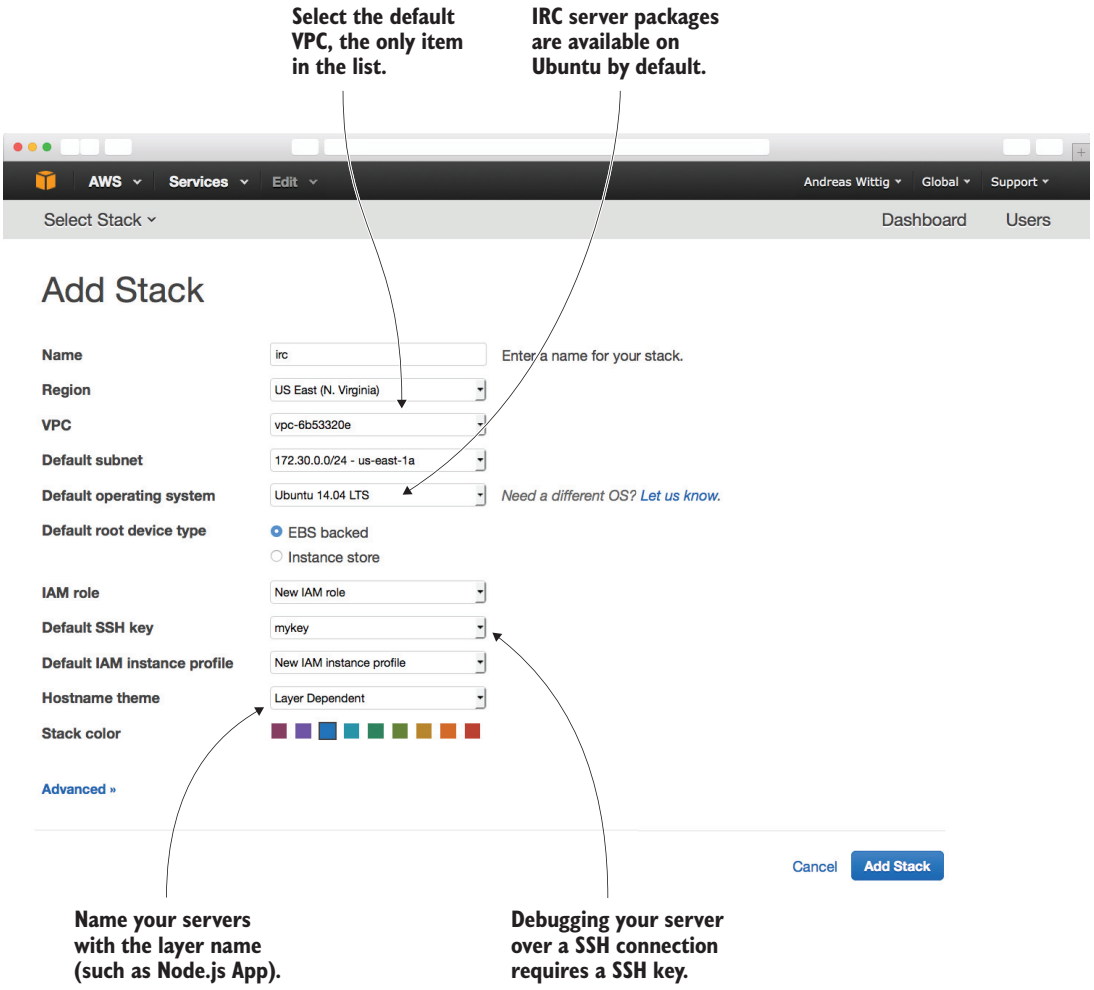


Figure 5.9 Creating a new stack with OpsWorks

You're redirected to an overview of your irc stack. Everything is ready for you to create the first layer.

CREATING A NODE.JS LAYER FOR AN OPSWORKS STACK

kiwiIRC is a Node.js application, so you need to create a Node.js layer for the irc stack. Follow these steps to do so:

1. Select Layers from the submenu.
2. Click the Add Layer button.
3. For Layer Type, select Node.js App Server, as shown in figure 5.10.
4. Select the latest 0.10.x version of Node.js.
5. Click Add Layer.

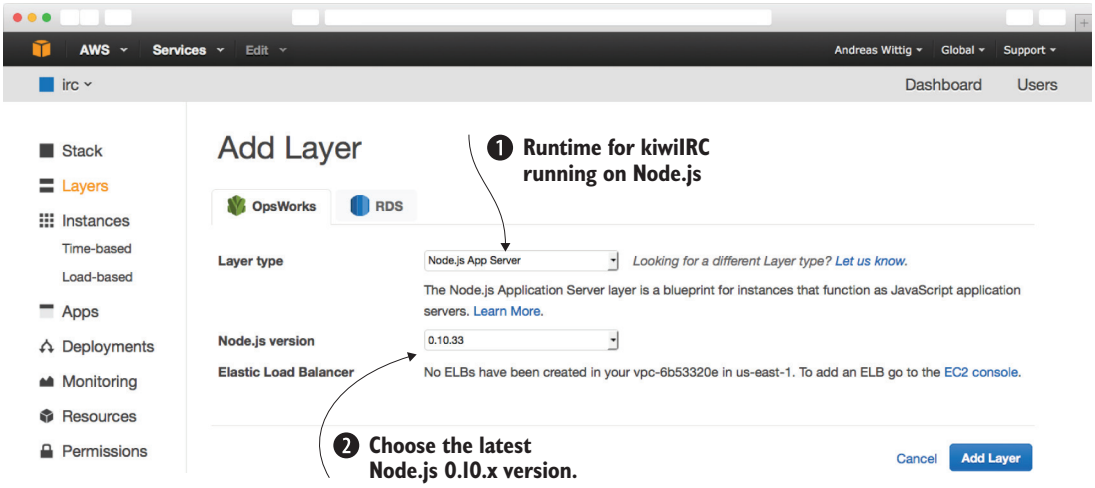


Figure 5.10 Creating a layer with Node.js for kiwiIRC

You've created a Node.js layer. Now you need to repeat these steps to add another layer and deploy your own IRC server.

Creating a custom layer for an OpsWorks stack

An IRC server isn't a typical web application, so the default layer types are out of the question. You'll use a custom layer to deploy an IRC server. The Ubuntu package repository includes various IRC server implementations; you'll use the `ircd-ircu` package. Follow these steps to create a custom stack for the IRC server:

1. Select Layers from the submenu.
2. Click Add Layer.
3. For Layer Type, select Custom, as shown in figure 5.11.
4. For Name and for Short Name, type in `irc-server`.
5. Click Add Layer.

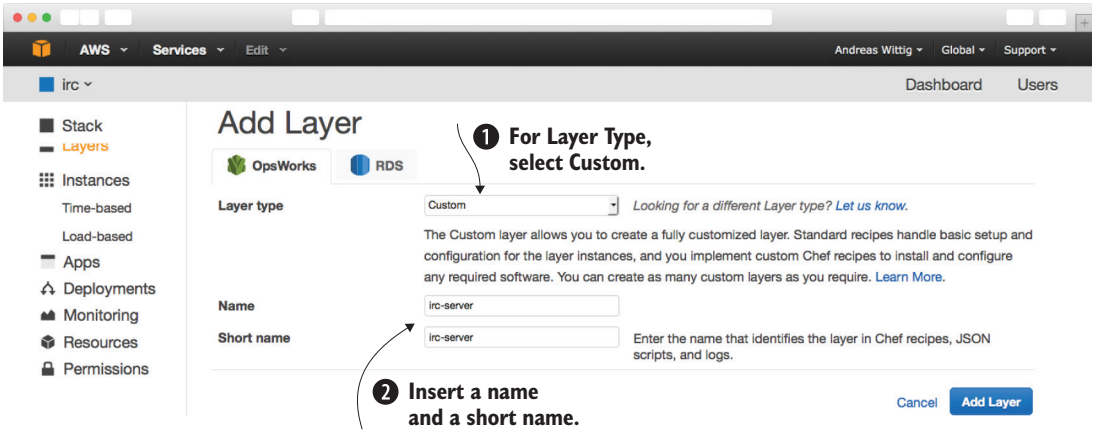


Figure 5.11 Creating a custom layer to deploy an IRC server

You've created a custom layer.

The IRC server needs to be reachable through port 6667. To allow access to this port, you need to define a custom firewall. Execute the commands shown in listing 5.5 to create a custom firewall for your IRC server.

Shortcut for OS X and Linux

You can avoid typing these commands manually to your command line by using the following command to download a bash script and execute it directly on your local machine. The bash script contains the same steps as shown in listing 5.5:

```
$ curl -s https://raw.githubusercontent.com/AWSinAction/\
code/master/chapter5/irc-create-cloudformation-stack.sh \
| bash -ex
```

Listing 5.5 Creating a custom firewall with the help of CloudFormation

```
$ aws ec2 describe-vpcs --query Vpcs[0].VpcId --output text

$ aws cloudformation create-stack --stack-name irc \
--template-url https://s3.amazonaws.com/awsinaction/\
chapter5/irc-cloudformation.json \
--parameters ParameterKey=VPC,ParameterValue=$VpcId

$ aws cloudformation describe-stacks --stack-name irc \
--query Stacks[0].StackStatus
```

Gets the default VPC, remember as $VpcId

Creates a CloudFormation stack

If the status is not COMPLETE, retry in 10 seconds.

Next you need to attach this custom firewall configuration to the custom OpsWorks layer. Follow these steps:

1. Select Layers from the submenu.
2. Open the irc-server layer by clicking it.
3. Change to the Security tab and click Edit.
4. For Custom Security Groups, select the security group that starts with irc, as shown in figure 5.12.
5. Click Save.

You need to configure one last thing for the IRC server layer: the layer recipes to deploy an IRC server. Follow these steps to do so:

1. Select Layers from the submenu.
2. Open the irc-server layer by clicking it.
3. Change to the Recipes tab and click Edit.
4. For OS Packages, add the package `ircd-ircu`, as shown in figure 5.13.
5. Click the + button and then the Save button.

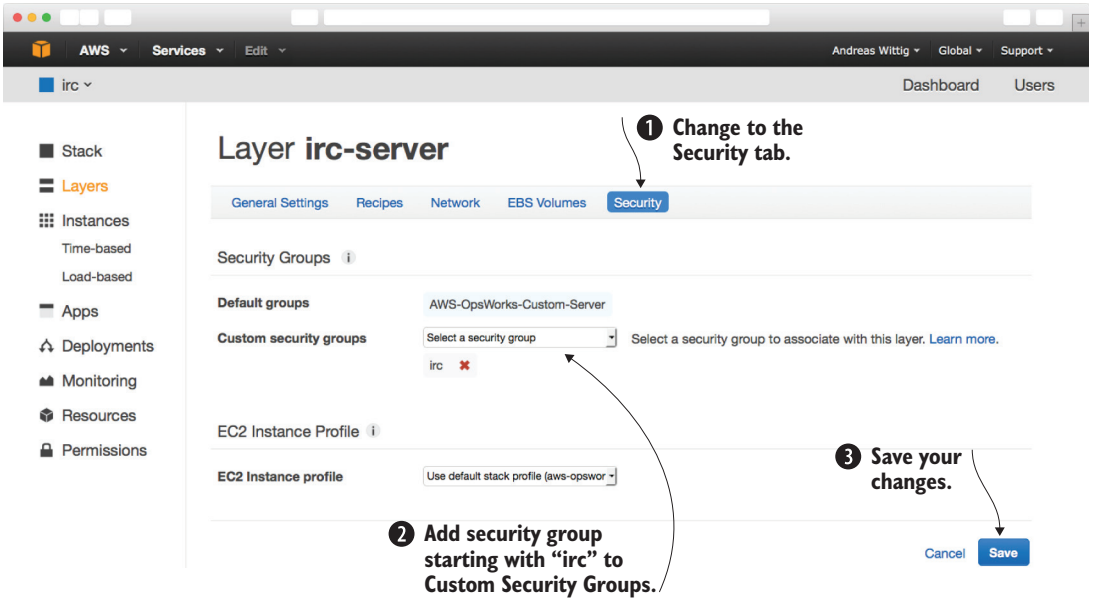


Figure 5.12 Adding a custom firewall configuration to the IRC server layer

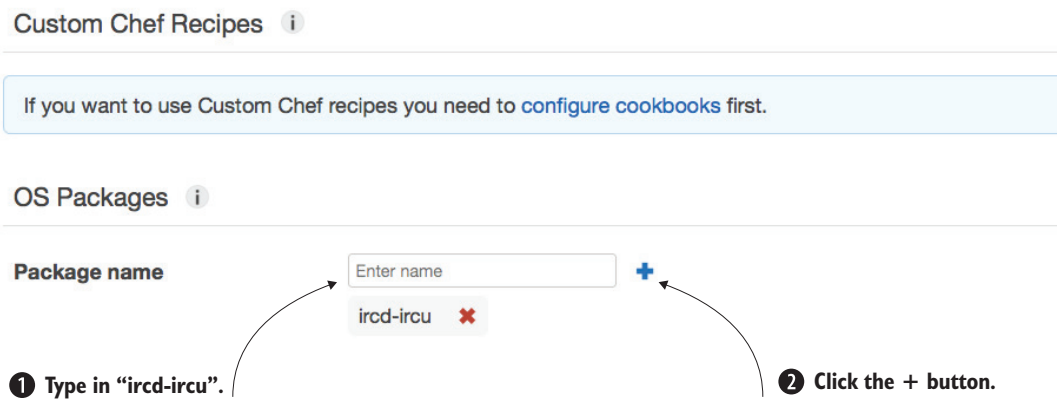


Figure 5.13 Adding an IRC package to a custom layer

You've successfully created and configured a custom layer to deploy the IRC server. Next you'll add the kiwiIRC web application as an app to OpsWorks.

ADDING AN APP TO THE NODE.JS LAYER

OpsWorks can deploy apps to a default layer. You've already created a Node.js layer. With the following steps, you'll add an app to this layer:

1. Select Apps from the submenu.
2. Click the Add an App button.
3. For Name, type in `kiwiIRC`.
4. For Type, select Node.js.
5. For Repository Type, select Git, and type in `https://github.com/AWSinAction/KiwiIRC.git` for Repository URL, as shown in figure 5.14.
6. Click the Add App button.

Your first OpsWorks stack is now fully configured. Only one thing is missing: you need to start some instances.

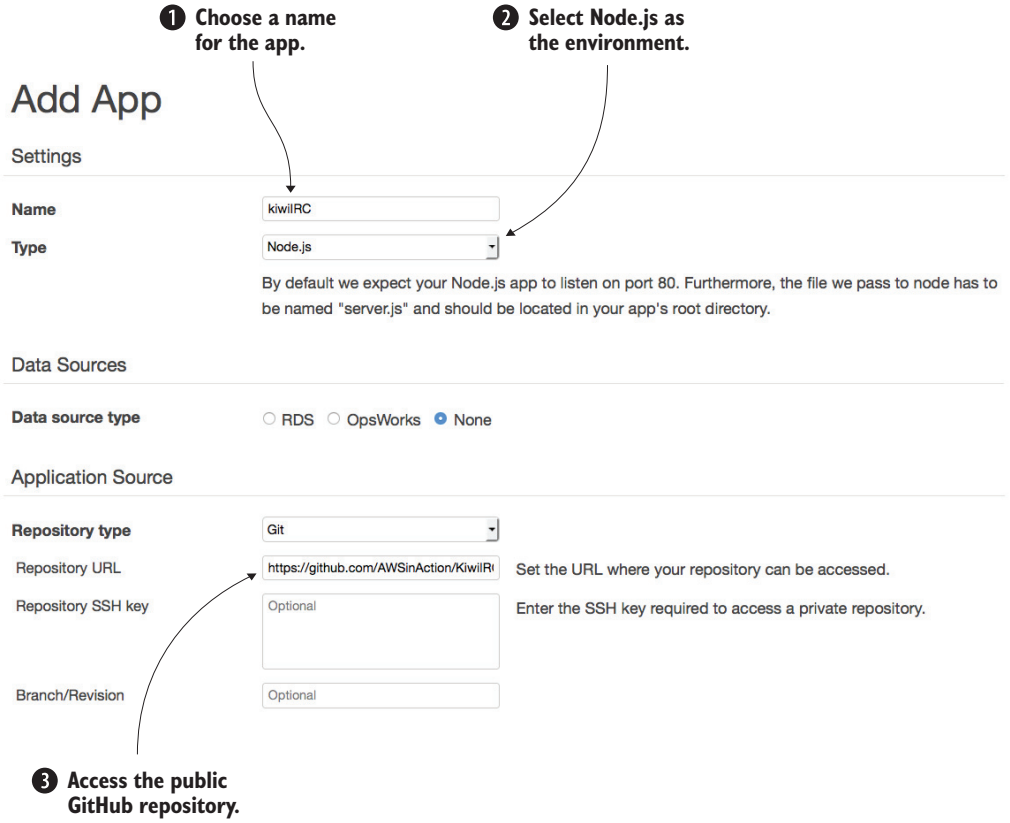


Figure 5.14 Adding kiwiIRC, a Node.js app, to OpsWorks

ADDING INSTANCES TO RUN THE IRC CLIENT AND SERVER

Adding two instances will bring the kiwiIRC client and the IRC server into being. Adding a new instance to a layer is easy—follow these steps:

1. Select Instances from the submenu.
2. Click the Add an Instance button on the Node.js App Server layer.
3. For Size, select t2.micro, the smallest and cheapest virtual server, as shown in figure 5.15.
4. Click Add Instance.

You've added an instance to the Node.js App Server layer. Repeat these steps for the irc-server layer as well.

The overview of instances should be similar to figure 5.16. To start the instances, click Start for both.

It will take some time for the virtual servers to boot and the deployment to run. It's a good time to get some coffee or tea.

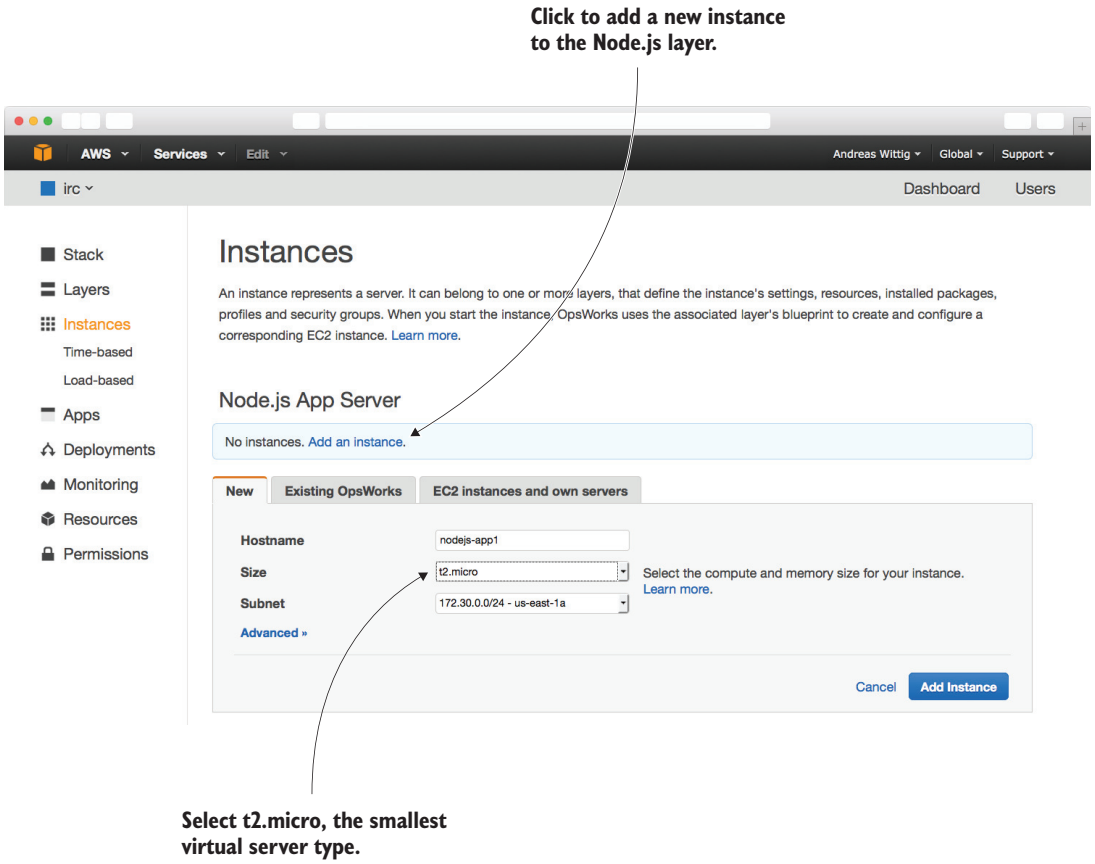


Figure 5.15 Adding a new instance to the Node.js layer

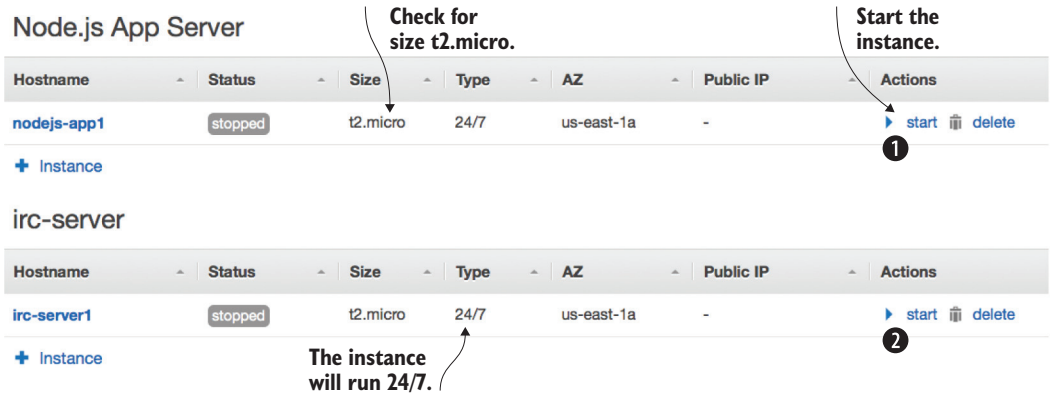


Figure 5.16 Starting the instances for the IRC web client and server

HAVING FUN WITH KIWIIRC

Be patient until the status of both instances changes to Online, as shown in figure 5.17. You can now open kiwiIRC in your browser by following these steps:

1. Keep in mind the public IP address of the instance irc-server1. You'll need it to connect to your IRC server later.
2. Click the public IP address of the nodejs-app1 instance to open the kiwiIRC web application in a new tab of your browser.

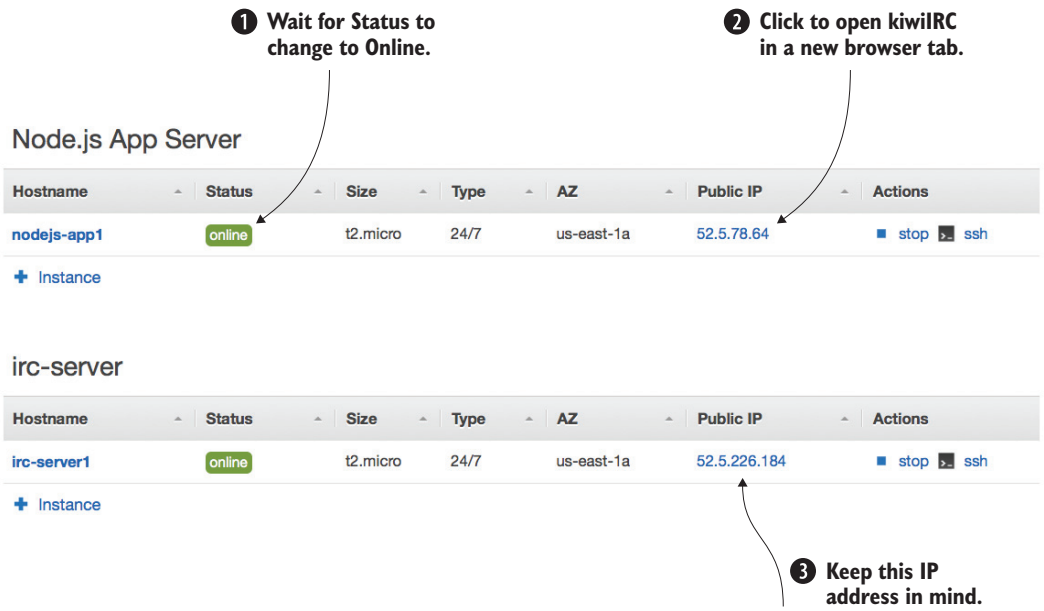


Figure 5.17 Waiting for deployment to open kiwiIRC in the browser

The kiwiIRC application should load in your browser, and you should see a login screen like the one shown in figure 5.18. Follow these steps to log in to your IRC server with the kiwiIRC web client:

1. Type in a nickname.
2. For Channel, type in `#awsinaction`.
3. Open the details of the connection by clicking Server and Network.
4. Type the IP address of irc-server1 into the Server field.
5. For Port, type in `6667`.
6. Disable SSL.
7. Click Start, and wait a few seconds.

Congratulations! You've deployed a web-based IRC client and an IRC server with the help of AWS OpsWorks.

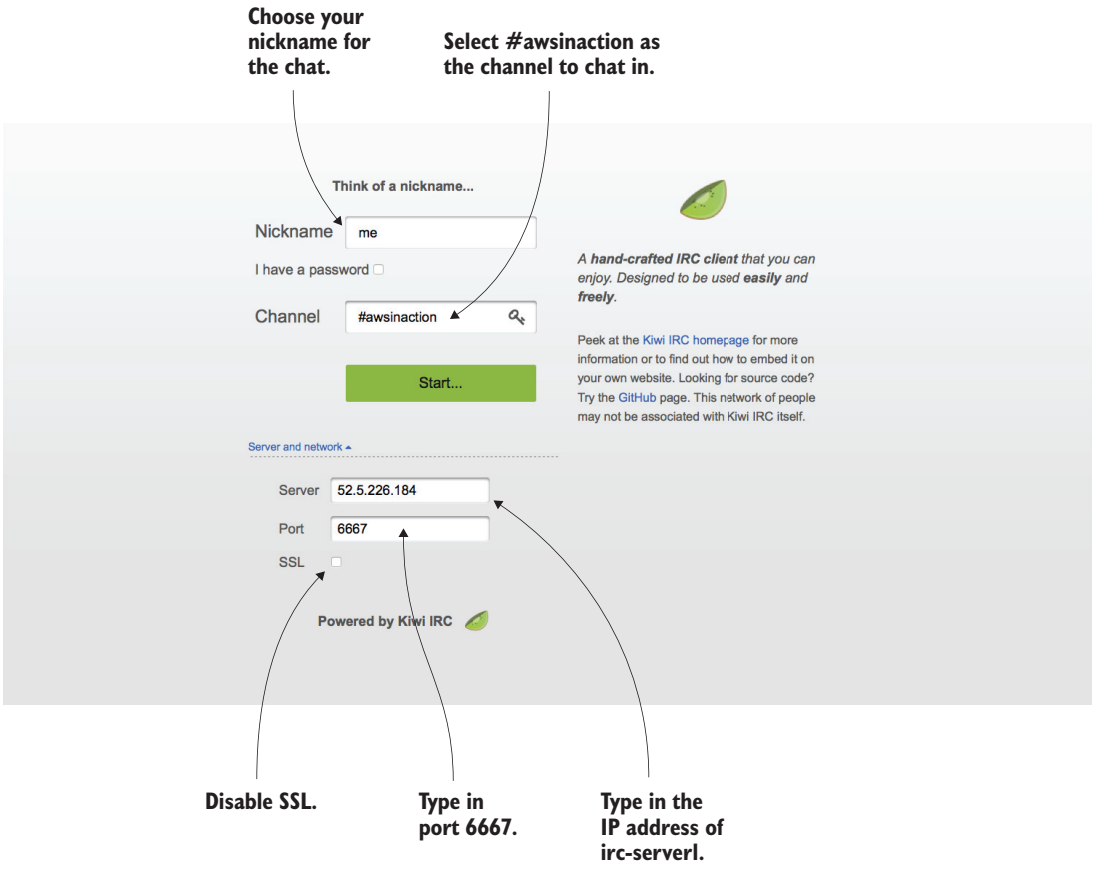


Figure 5.18 Using kiwiIRC to log in to your IRC server on channel #awsinaction

Cleaning up

It's time to clean up. Follow these steps to avoid unintentional costs:

1. Open the OpsWorks service with the Management Console.
2. Select the irc stack by clicking it.
3. Select Instances from the submenu.
4. Stop both instances and wait until Status is Stopped for both.
5. Delete both instances, and wait until they disappear from the overview.
6. Select Apps from the submenu.
7. Delete the kiwiIRC app.
8. Select Stack from the submenu.
9. Click the Delete Stack button, and confirm the deletion.
10. Execute `aws cloudformation delete-stack --stack-name irc` from your terminal.

5.5 Comparing deployment tools

You have deployed applications in three ways in this chapter:

- Using AWS CloudFormation to run a script on server startup
- Using AWS Elastic Beanstalk to deploy a common web application
- Using AWS OpsWorks to deploy a multilayer application

In this section, we'll discuss the differences between these solutions.

5.5.1 Classifying the deployment tools

Figure 5.19 classifies the three AWS deployment options. The effort required to deploy an application with the help of AWS Elastic Beanstalk is low. To benefit from this, your application has to fit into the conventions of Elastic Beanstalk. For example, the application must run in one of the standardized runtime environments. If you're using OpsWorks, you'll have more freedom to adapt the service to the needs of your application. For example, you can deploy different layers that depend on each other, or you can use a custom layer to deploy any application with the help of a Chef recipe; this takes extra effort but gives you additional freedom. On the other end of the spectrum, you'll find CloudFormation and deploying applications with the help of a script running at the end of the boot process. You can deploy any application with the help of CloudFormation. The disadvantage of this approach is that you have to do more work because you don't use standard tooling.

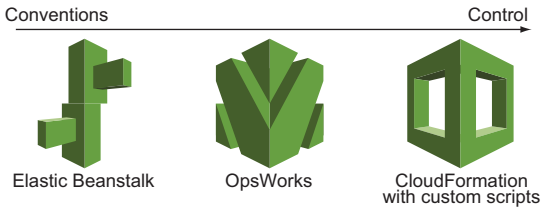


Figure 5.19 Comparing different ways to deploy applications on AWS

5.5.2 Comparing the deployment services

The previous classification can help you decide the best fit to deploy an application. The comparison in table 5.1 highlights other important considerations.

Table 5.1 Differences between using CloudFormation with a script on server startup, Elastic Beanstalk, and OpsWorks

	CloudFormation with a script on server startup	Elastic Beanstalk	OpsWorks
Configuration-management tool	All available tools	Proprietary	Chef
Supported platforms	Any	▪ PHP ▪ Node.js ▪ IIS ▪ Java/Tomcat ▪ Python ▪ Ruby ▪ Docker	▪ Ruby on Rails ▪ Node.js ▪ PHP ▪ Java/Tomcat ▪ Custom/any
Supported deployment artifacts	Any	Zip archive on Amazon S3	Git, SVN, archive (such as Zip)
Common use case	Complex and nonstandard environments	Common web application	Micro-services environment
Update without downtime	Possible	Yes	Yes
Vendor lock-in effect	Medium	High	Medium

Many other options are available for deploying applications on AWS, from open source software to third-party services. Our advice is to use one of the AWS deployment services because they're well integrated into many other AWS services. We recommend that you use CloudFormation with user data to deploy applications because it's a flexible approach. It is also possible to manage Elastic Beanstalk and Ops Works with the help of CloudFormation.

An automated deployment process will help you to iterate and innovate more quickly. You'll deploy new versions of your applications more often. To avoid service interruptions, you need to think about testing changes to software and infrastructure in an automated way and being able to roll back to a previous version quickly if necessary.

5.6 Summary

- It isn't advisable to deploy applications to virtual servers manually because virtual servers pop up more often in a dynamic cloud environment.
- AWS offers different tools that can help you deploy applications onto virtual servers. Using one of these tools prevents you from reinventing the wheel.

- You can throw away a server to update an application if you've automated your deployment process.
- Injecting Bash or PowerShell scripts into a virtual server during startup allows you to initialize servers individually—for example, for installing software or configuring services.
- OpsWorks is good for deploying multilayer applications with the help of Chef.
- Elastic Beanstalk is best suited for deploying common web applications.
- CloudFormation gives you the most control when you're deploying more complex applications.

Physical data centers require lots of equipment and take time and resources to manage. If you need a data center, but don't want to build your own, Amazon Web Services may be your solution. Whether you're analyzing real-time data, building software as a service, or running an e-commerce site, AWS offers you a reliable cloud-based platform with services that scale.

Amazon Web Services in Action introduces you to computing, storing, and networking in the AWS cloud. The book will teach you about the most important services on AWS. You will also learn about best practices regarding security, high availability and scalability. You'll start with a broad overview of cloud computing and AWS and learn how to spin-up servers manually and from the command line. You'll learn how to automate your infrastructure by programmatically calling the AWS API to control every part of AWS. You will be introduced to the concept of Infrastructure as Code with the help of AWS CloudFormation. You will learn about different approaches to deploy applications on AWS. You'll also learn how to secure your infrastructure by isolating networks, controlling traffic and managing access to AWS resources. Next, you'll learn options and techniques for storing your data. You will experience how to integrate AWS services into your own applications by the use of SDKs. Finally, this book teaches you how to design for high availability, fault tolerance, and scalability.

What's inside

- Overview of AWS cloud concepts and best practices
- Manage servers on EC2 for cost-effectiveness
- Infrastructure automation with Infrastructure as Code (AWS CloudFormation)
- Deploy applications on AWS
- Store data on AWS: SQL, NoSQL, object storage and block storage
- Integrate Amazon's pre-built services
- Architect highly available and fault tolerant systems

Written for developers and DevOps engineers moving distributed applications to the AWS platform.

index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

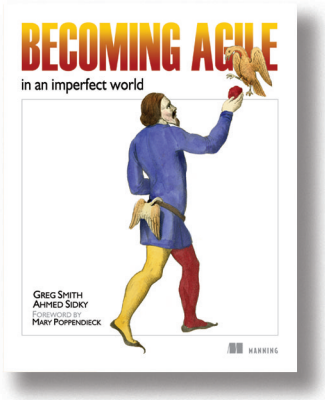
T

U

V

W

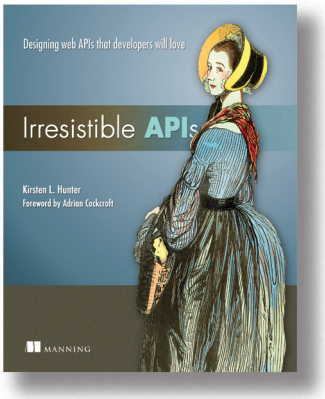
Y



Becoming Agile

By Greg Smith and Ahmed Sidky

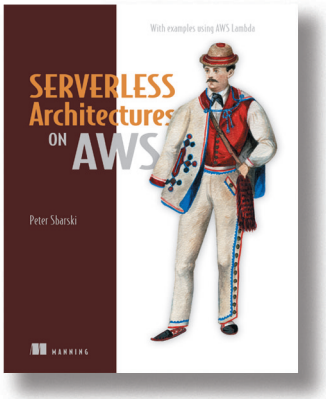
ISBN: 9781933988252
408 pages
$44.99
2009



Irresistible APIs

By Kirsten Hunter

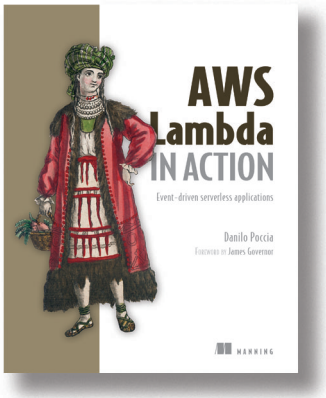
ISBN: 9781617292552
232 pages
$44.99
2016



Serverless Architectures on AWS

By Peter Sbarski

ISBN: 9781617293825
425 pages
$44.99
2017 (estimated)



AWS Lambda in Action

By Danilo Poccia

ISBN: ISBN 9781617293719
384 pages
$49.99
2016

Amazon Web Services in Action

By Michael Wittig and Andreas Wittig

ISBN: 9781617292880
424 pages
$49.99
2015