SignalR Blueprints

Build real-time ASP.NET web applications with SignalR and create various interesting projects to improve your user experience

Einar Ingebrigtsen
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 ‘The Primer’
- A synopsis of the book’s content
- More information on **SignalR Blueprints**

**About the Author**

**Einar Ingebrigtsen** has been working professionally with software since 1994, ranging from games development on platforms such as the PlayStation, Xbox, and PC to the enterprise line of business application development since 2002. He has always kept his focus on creating great products with great user experiences, putting the user first. Einar received the Microsoft MVP award in October 2008, awarded for his work in the community and in Silverlight space with open source projects such as Balder—a 3D engine for Silverlight. Today, Einar runs a company called Dolittle together with his partners, doing consultancy and building his own products with their own open source projects at the heart of what they do. The clients that Dolittle has had in the last couple of years include NRK (the largest TV broadcaster in Norway), Statoil (a Norwegian oil company), Komplett (the largest e-commerce company in Norway), and Holte (a leading Norwegian developer for construction software).

A strong believer in open source, Einar runs a few projects in addition to Balder, the largest being Bifrost ([http://bifrost.dolittle.com/](http://bifrost.dolittle.com/)), a line of business platform for .NET developers, and Forseti ([https://github.com/dolittle/forseti](https://github.com/dolittle/forseti)), a headless auto-running JavaScript test runner.

Additionally, Einar loves talking at user groups and conferences and has been a frequent speaker at Microsoft venues, talking about different topics—the last couple of years mostly about architecture, code quality, and cloud computing. His personal blog is at [http://www.ingebrigtsen.info/](http://www.ingebrigtsen.info/) and the company blog is at [http://blog.dolittle.com/](http://blog.dolittle.com/).

Einar is also the author of the book **SignalR: Real-time Application Development**, Packt Publishing.
Acknowledgments

It might sound like a cliché, but seriously, without my wife, Anne Grethe, this book could not have happened. Her patience with me and her support is truly what pretty much makes just about anything I do a reality. My kids, Mia and Herman, you rock! Thanks to them for keeping me mentally younger and making me playful. I'd also like to thank my colleagues, who have been kind enough not to point out the fact that I've had too much going on in the period of writing this book. I'll be sure to buy a round the next time we're having a company get together.

Last but not least, thanks to my clients for their patience and flexibility during the busiest periods of this process.
SignalR Blueprints

The purpose of software is to be a tool for us humans to help us perform tasks. A lot of software is also a replacement for something physical for which we had an opportunity to increase productivity by making it digital and also more accessible. When replacing a paper form with a digital solution, we as developers pretty much just copied the form field by field and never really thought through what we were trying to solve. This made the improvement all about the data rather than what the users were really trying to do. One of the benefits of having the forms digitally is that multiple users can see the same data at the same time and even edit it at the same time. However, since it has all been modeled as data, with often a single, large model representing it, we introduce new problems we never had in the real world on paper. Things such as transactions and data staleness make our software more complex and they never make sense at all for the user. These are technical requirements that we, as developers, have introduced to make sure the data is correct at all times.

Users are becoming better; they have new requirements based on their experience with software. Even in the enterprise, users are now demanding more of their IT systems. With the advent of the real-time Web, driven by services such as Facebook, Twitter, and other social media, our users are now used to different experiences that are more responsive and user friendly.

What is the state of the Web?

The Web has changed a lot over the years, but the core protocol, HTTP, has been pretty consistent since its first documented version was released in 1991. The protocol was optimized for the document delivery system that makes up the World Wide Web. Later, the protocol included information that helped us keep a session state on the server and be able to link subsequent requests coming into the same session state. Later with DHTML and AJAX, we didn't have to do full post backs to get the full document but get parts of the document, or maybe even just get the data and perform the necessary rendering or manipulation of the document in the client.

With the introduction of standards such as WebSockets, server sent events, and the like, we can go even further. We can now make our web solutions come even more alive by having persistent connections with a server and get notified from the server when something happens. This solves some of the problems discussed earlier, such as transactions and data staleness. By basically getting the changes continuously from any other users as they are doing them, we don't need to run into any conditions that put the system in a mode that it can't get out of. This will increase the user experience and make our job as developers a lot easier.
*SignalR Blueprints* will allow you to utilize SignalR to its fullest, showing you how to create different application types on the Web and mobile devices, along with a few tips and tricks along the way. In addition, this project book aims to show you the patterns that are not only good for SignalR but generally with cloud scale in mind. Most significantly, you will learn to think differently about software for users, keeping them in focus all the time.

**Personal style**

Throughout the book, you'll run into things you might disagree with. It could be things in naming the classes or methods in C#, for instance, at times, I like to drop camel casing, both upper and lower and just separate the words with an underscore yielding "some_type_with_spaces". In addition, I don't use modifiers, without them adding any value. You'll see that I completely avoid private as that is the default modifier for fields or properties on types. I'll also avoid things such as read-only, especially, if it's a private member. Most annoyingly, you might see that I drop scoping for single line statements following an IF or FOR. Don't worry, this is my personal style; you can do as you please. All I'm asking is that you don't judge me by how my code looks. I'm not a huge fan of measuring code quality with tools such as R# and its default setting for squiggles. In fact, a colleague and I have been toying with the idea of using the underscore trick for all our code, as it really makes it a lot easier to read.

You'll notice throughout that I'm using built-in functions in the browser in JavaScript, where you might expect jQuery. The reason for this is basically that I try to limit the usage of jQuery, in fact, it's a dependency I'd prefer not to have in my solutions as it is not adding anything to the way I do things. There is a bit of an educational, quite intentional reason for me not using jQuery as well; we now have most of the things we need in the browser already.

**What This Book Covers**

*Chapter 1, The Primer,* shows us that in order to hit the ground running with SignalR, it is important to understand why we need SignalR, but even more importantly, the architectural decisions that lead to the thinking behind SignalR. For the remainder of this book, there will be patterns and practices applied; this chapter covers that. In addition, it also covers the libraries being used.

*Chapter 2, Overheating the Discussion,* starts gradually by getting to know the basics, without introducing too much technology and patterns and practices. This chapter goes through the building of a forum that benefits from SignalR.
Chapter 3, *Extra! Extra! Read All About It!*, introduces e-newspapers—a great scenario for SignalR and a quite common feature found on the Web. Having SignalR at the core could be the tiny thing that differentiates you from the crowd. You'll learn how to scale to meet demand when things go viral.

Chapter 4, *Can You Measure It?*, introduces increasingly popular dashboards that will give you numbers at a glance. Often, the aim is to have the dashboards as up to date as possible but without having to do a timer that refreshes. SignalR can help here and light it all up, and with the right technique, make it visually appealing.

Chapter 5, *What Line of Business Are You In?*, shows that enterprise line of business apps are often referred to as where user experiences go to die, leaving users out of the equation when the software is designed and implemented. There are no reasons whatsoever for this. This chapter investigates how can we start to think differently about things and make them twinkle, like the stars discovered by the many voyages of the Starship Enterprise.

Chapter 6, *An Architectural Taste*, shows that software architecture is very important for many reasons, and this chapter looks more closely at a particular flavor that lends itself to the idea of real-time applications.

Chapter 7, *The Three Screens – Mobile First*, teaches how to connect the phone as a frontend for what we built in the previous chapter. SignalR is not only for the Web; it supports a wide variety of platforms, one of them being the Windows Phone.

Chapter 8, *Putting the X in .NET – Xamarin*, builds a frontend for the forum in Chapter 2, *Overheating the Discussion*, for the iPhone.

Chapter 9, *Debugging or Troubleshooting*, shows a few techniques that you can apply to find out why the code gets broken or systems in general don't do what they are told to. Then, you need to figure out what went wrong.

Chapter 10, *Hosting and Deployment*, walks through the varieties and particularly focuses on cloud and Microsoft Azure. An important fact here will be how one scales.
This chapter serves as a primer covering of all the terms, patterns, and practices applied in the book. Also, you will learn about the tools, libraries, and frameworks being used and what their use cases are. More importantly, you will find out why you should be performing these different things and, in particular, why you should use SignalR, and how the methods you employ will naturally find their way into your software.

Where are we coming from?
By asking where we are coming from, I'm not trying to ask an existential question that dates back to the first signs of life on this planet. Rather, we are looking at the scope of our industry, and what has directed us to where we are now and how we create software today. The software industry is very young and is in constant movement. We haven't quite settled in yet like other professions have. The rapid advances in computer hardware present opportunities for software all the time. We find better ways of doing things as we improve our skills as a community. With the Internet and the means of communication we have today, these changes are happening fast and frequently. This is to say that we are changing a lot more than any other industry. With all this being said, a lot of these changes go back to the roots of our industry. Computers and software are the tools meant to solve problems for humans, and often in the line of business applications that we write, these tools and software are there to remove manual labor or remove paper clutter. The way these applications are modeled is therefore often closely related to the manual or paper version, not really modeling the process or applying the full capability of what the computer could do to actually improve the experience of the particular process.
The terminal

Back in the early days of computing, computers lacked CPU power and memory. They were expensive, and if you wanted something powerful, it would fill the room with refrigerator-sized computers. The idea of a computer, at least a powerful one, on each desk was not feasible. Instead of delivering rich computers onto desks, the notion of terminals became a reality. These were connected to the mainframe and were completely stateless. The entirety of each terminal was kept in the mainframe, and the only thing transferred from the client was user input and the only thing coming back from the mainframe was any screen updates.

![Diagram showing the relationship between multiple terminals and a mainframe](image)

The relationship between multiple terminals connected to a mainframe and all terminals exist without state, with the mainframe maintaining the state and views.

Fast forwarding

The previous methods of thinking established the pattern for software moving through the decades. Looking at web applications with a server component in the early days of the Web, you'll see the exact same pattern: a server that keeps the state of the user and the clients being pretty limited; this being the web browser. In fact, the only thing going back and forth between them was the user input from the client and the result in the form of HTML going back.
Bringing this image really up to speed with the advancement of AJAX, the image would be represented as shown in the following diagram:

![Diagram of HTTP flow](image)

**Completing the circle**

Of course, by skipping three decades of evolution in computing, we are bound to miss a few things. However, the gist of most techniques has been that we keep the state on the server and we have to go from the client in the sense of request, be it a keystroke or a HTTP request, before receiving a response. At the core of this sits a network stack with capabilities beyond what the overlying techniques have been doing. In games, for instance, the underlying sockets have been used much more in order for us to be able to actually play multiplayer games, starting off with games on your local network to massive multiplayer online games with thousands of users connected at once. In games, the request/response pattern will not work as they yield different techniques and patterns. We can't apply all the things that have been achieved in games because a lot of it is based on approximation due to network latency. However, we don't have the requirements of games either to reflect the truth in an interval of every 16-20 milliseconds. Accuracy is far more important in the world of line of business application development where it needs to be accurate constantly. The user has to trust the outcome of their operations in the system.

Having said this, it does not mean that the output has to be in synchrony. Things can eventually be consistent and accurate, just as long as the user is well informed. By allowing eventual consistency, you open up a lot of benefits about how we build our software and you have a great opportunity to improve the user experience of the software you are building, which should be at the very forefront of your thinking when making software.
Eventual consistency basically means that the user performs an action and, asynchronously, it will be dealt with by the system and also eventually be performed. When it's actually performed, you could notify the user. If it fails, let the client know so that it can perform any compensating action or present something to the user. This is becoming a very common approach. It does impose a few new things to think about. We seldom build software that targets us as developers but rather has other users in mind when building it. This is the reason we go to work and build software for users. The user experience should therefore be the most important aspect and should always be the driving force and the main motive to apply a new technique. Of course, there are other aspects to decision making (such as budget) as this gives us business value, and so on. These are also vital parts of decision-making, but make sure that you never lose focus on the user.

How can we complete the circle and improve the model and take what we've learned and mix in a bit of real-time thinking? Instead of thinking that we need a response right away and pretty much locking up the user interface, we can send off the request for what we want and not wait for it at all. So, let the user carry on and then let the server tell us the result when it is ready. But hang on, I mentioned accuracy; doesn't this mean that we would be sitting with that client in an incorrect state? There are ways to deal with this in a user-friendly fashion. They are as follows:

- For simple things, you could assume that the server will perform the action and just perform the same thing on the client side. This will give instant feedback to the user and the user can then carry on. If, for some reason, the action didn't succeed on the server, the server can, at a later stage, send the error related to the action that was performed and the client can perform a compensating action. Undoing this and notifying the user that it couldn't be performed is an example. An error should only be considered an edge case, so instead of modeling everything around the error, model the happy path and deal with the error on its own.

- Another approach would be to lock the particular element that was changed in the client but not the entire user interface, just the part that was modified or created. When the action succeeds and the server tells you, you can easily mark the element(s) as succeeded and apply the result from the server. Both of these techniques are valid and I would argue that you should apply both, depending on the circumstances.
Chapter 1

SignalR
What does this all mean and how does SignalR fit into all this?

A regular vanilla web application without even being AJAX-enabled will do a full round-trip from the client to server for the entire page and all its parts when something is performed. This puts a strain on the server to serve the content and maybe even having to perform rendering on the server before returning the request. However, it also puts a strain on the bandwidth, having to return all the content all the time. AJAX-enabled web apps made this a lot better by typically not posting a full page back all the time. Today, with Single Page Applications (SPA), we never do a full-page rendering or reloading and often not even rely on the server rendering anything. Instead, it just sits there serving static content in the form of HTML, CSS, and JavaScript files and then provides an API that can be consumed by the client.

SignalR goes a step further by representing an abstraction that gives you a persistent connection between the server and the client. You can send anything to the server and the server can at any time send anything back to the client, breaking the request/response pattern completely. We lose the overhead of the regular request or response pattern of the Web for every little thing that we need to do. From a resource perspective, you will end up needing less from both your server and your client. For instance, web requests are returned back to the request pool of ASP.NET as soon as possible and reduce the memory and CPU usage on the server.

By default, SignalR will choose the best way to accomplish this based on the capabilities of the client and the server combined. Ranging from WebSockets to Server Sent Events to Long Polling Requests, it promises to be able to connect a client and a server. If a connection is broken, SignalR will try to re-establish it from the client immediately.

Although SignalR uses long polling, the response going back from the server to a client is vastly improved rather than having to do a pull on an interval, which was the approach for AJAX-enabled applications before.

You can force SignalR to choose a specific technique as long as you have requirements that limit what is allowed. However, when left as default, it will negotiate what is the best fit.

Terminology
As in any industry, we have a language that we use and it is not always ubiquitous. We might be saying the same thing but the meaning might vary. Throughout the book, you’ll find terms being used in order for you to understand what is being referred to; I’ll summarize what these terms mean.
The Primer

**Messaging**

A message in computer software refers to a unit of communication that contains information that the source wants to communicate to the outside world, either to specific recipients or as a broadcast to all recipients connected now or in the future. The message itself is nothing but a container holding information to identify the type of the message and the data associated with it. Messaging is used in a variety of ways. One way is either through the Win16/32 APIs with WM_* messages being sent for any user input or changes occurring in the UI. Another is things affecting the application to XML messages used to integrate systems. It could also be typed messages inside the software, modeled directly as a type. It comes in various forms, but the general purpose is to be able to do it in a decoupled manner that tells other parts that something has happened. The message and its identifier with its payload becomes the contract in which the decoupled systems know about. The two systems would not know about each other.

**Publish/Subscribe**

With your message in place, you want to typically send it. Publish/Subscribe, or in shorthand "PubSub", is often what you're looking for. The message can be broadcasted and any part of your system can subscribe to the message by type and react to it. This decouples the components in your system by leaving a message. This is achieved by having a message box sitting in the middle that all systems know about, which could be a local or global message box, depending on how your model thinks. The message box will then be given message calls, or will activate subscriptions, which are often specific to a message type or identifier.

The message box can be made smarter, which for instance could be by persisting all messages going through so that any future subscribers can be told what happened in the past. This is presented by the following diagram:

![Diagram](image)

A representation of how the subsystems have a direct relationship with a message box, enabling the two systems to be decoupled from each other.
Decoupling

There are quite a few paradigms in the art of programming and it all boils down to what is right for you. It's hard to argue what is right or wrong because the success of any paradigm is really hard to measure. Some people like a procedural approach to things where you can read end-to-end how a system is put together, which often leads to a much coupled solution. Solutions are things put together in a sequence and the elements can know about each other. The complete opposite approach would be to completely decouple things and break each problem into its own isolated unit with each unit not knowing about the other. This approach breaks everything down into more manageable units and helps keep the complexity down. It really helps in the long term velocity of development and explains also how you can grow the functionality. In fact, it also helps with taking things out if one discovers one has features that aren't being used. By decoupling software and putting things in isolation and even sprinkle some SOLID on top of this (which is known as a collection of principles; this being the Single responsibility principle – Open/closed principle – Liskov substitution principle – Interface segregation principle – Dependency inversion principle). You can find more information about this at http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf.

By applying these practices with decoupling in mind, we can:

- Make it easier to scale up your team with more developers; things are separated out and responsibilities within the team can be done as well.
- Make more maintainable solutions.
- Take resource hungry parts of your system and put them on separate servers, something that is harder to accomplish if it all is coupled together.
- Gain more flexibility by focusing more on each individual parts and then compose it back together any way you like.
- Make it easier to identify bottlenecks in isolation.
- Have less chance of breaking other parts of the system when fixing or expanding your code.
- Gain higher development pace.
- Finally, this might be a bold claim, but you could encounter fewer bugs! Or at least, they would be more maintainable bugs that sit inside isolated and focused code, making it easier to identity and safer to fix.

The ability to publish messages rather than calling concrete implementations becomes vital. These become the contracts within your system.
This book will constantly remind you of one thing: users are a big part in making this decision. Making your system flexible and more maintainable is of interest to your users. The turnaround time to fix bugs along with delivering new features is very much in their interest. One of the things I see a lot in projects is that we tend to try to define everything way too early and often upfront of development, taking an end-to-end design approach. This often leads to overthinking and often coupling, making it harder to change later on when we know more. By making exactly what is asked for and not trying to be too creative and add things that could be nice to have, and then really thinking of small problems and rather compose it back together, the chance of success is bigger and also easier to maintain and change. Having said this, decoupling is, ironically enough, tightly coupled with the SOLID principles along with other principles to really accomplish this. For instance, take the S in SOLID. This represents the Single Responsibility Principle; it governs that a single unit should not do more than one thing. A unit can go all the way down to a method. Breaking up things into more tangible units rather than huge unreadable units makes your code more flexible and more readable.

Decoupling will play a vital role in the remainder of the book.

Patterns

Techniques that repeat can be classified as patterns; you probably already have a bunch of patterns in your own code that you might classify even as your own patterns. Some of these become popular outside the realms of one developer's head and are promoted beyond just this one guy. A pattern is a well-understood solution to a particular problem. They are identified rather than "created". That is, they emerge and are abstracted from solutions to real-world problems rather than being imposed on a problem from the outside. It's also a common vocabulary that allows developers to communicate more efficiently. A popular book that aims to gather some of these patterns is Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley Professional. You can find a copy at http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612.

We will be using different patterns throughout this book, so it's important to understand what they are, the motivation behind them, and how they are applied successfully. The following sections will give you a short summary of the patterns being referred to and used.
Model-View-Controller
Interestingly enough, most of the patterns we have started applying have been around for quite a while. The Model-View-Controller (MVC) pattern is a great example of this.

MVC was first introduced by a fellow Norwegian national called Dr. Trygve Reenskaug in 1973 in a paper called Administrative Control in the Shipyard (http://martinfowler.com/eaaDev/uiArcha.html). Since then, it has been applied successfully in a variety of frameworks and platforms. With the introduction of Ruby on Rails in 2005, I would argue the focus on MVC really started to get traction in the modern web development sphere. When Microsoft published ASP.NET MVC at the end of 2007, they helped gain focus in the .NET community as well.

The purpose of MVC is to decouple the elements in the frontend and create a better isolated focus on these different concerns. Basically, what one has is a controller that governs the actions that are allowed to be performed for a particular feature of your application. The actions can return a result in the form of either data or concrete new views to navigate to. The controller is responsible for holding and providing any state to the views through the actions it exposes. By state, we often think of the model and often the data comes from a database, either directly exposed or adapted into a view-specific model that suits the view better than the raw data from the database. The relationship between model, controller, view, and the user is summarized in the following diagram:

A representation of how the artifacts make up MVC (don't forget there is a user that will interact with all of these artifacts)
With this approach, you separate out the presentation aspect of the business logic into the controller. The controller then has a relationship with other subsystems that knows the other aspects of the business logic in a better manner, letting the controller only focus on the logic that is specific to the presentation and not on any concrete business logic but more on the presentation aspect of any business logic. This decouples it from the underlying subsystem and thus more specialized. The view now has to concern itself with only view-related things, which are typically HTML and CSS for web applications. The model, either a concrete model from the database or adapted for the view, is fetched from whatever data source you have.

**Model-View-ViewModel**

Extending on the promise of decoupling in the frontend, we get something called Model-View-ViewModel (MVVM); for more information, visit [http://www.codeproject.com/Articles/100175/Model-View-ViewModel-MVVM-Explained](http://www.codeproject.com/Articles/100175/Model-View-ViewModel-MVVM-Explained). This is a design pattern for the frontend based largely on MVC but it takes it a bit further in terms of decoupling. From this, Microsoft created a specialization called MVVM, as it is called today.

MVVM was presented by Martin Fowler in 2004 to what he referred to as the Presentation Model (which you can access at [http://martinfowler.com/eaaDev/PresentationModel.html](http://martinfowler.com/eaaDev/PresentationModel.html)).

The ViewModel is a key player in this that holds the state and behavior needed for your feature to be able to do its job without it knowing about the view. The view will then be observing the ViewModel for any changes it might get and utilize any behaviors it exposes. In the ViewModel, we keep the state, and as with MVC, the state is in the form of a model that could be a direct model coming from your data source or an adapted model that is more fine-tuned to the purpose of the frontend.

The additional decoupling, which this model represents, lies within the fact that the ViewModel has no clue to any view, and in fact should be blissfully unaware that it is being used in a view. This makes the code even more focused and it opens an opportunity of being able to swap out the view at any given time or even reuse the same ViewModel with its logic and state for the second view.
The relationship between the Model, View, ViewModel, and the user is summarized in the following diagram:

![Diagram](image_url)

The artifacts that make up MVVM (don’t forget the user interacts with these artifacts through the view)

**Command Query Responsibility Segregation**

Back in 1988, Betrand Meyer published a book, *Object-oriented Software Construction* (https://archive.eiffel.com/doc/oosc/page.html). In this book, one of the things being addressed was the separation of actions being performed in the system and data being returned. The data displayed and the actions performed are, in fact, two completely different concerns and it’s described as commands for the tasks being performed and queries for the data one gets. At its core, there are no methods or functions that can perform an action and return data. These are separated out as two different operations, leading to the concept of **Command Query Separation (CQS)**; more information can be found at [http://codebetter.com/gregyoung/2009/08/13/command-query-separation/]. Greg Young and Udi Dahan reached the conclusion of refining CQS into something called **Command Query Responsibility Segregation (CQRS)**. At the heart of this sits the SOLID principles, especially with SRP, whereas the ideas of separation of concerns is on top (more information is available at [http://deviq.com/separation-of-concerns]). The evolution in CQRS over CQS was to take what was identified by Bertrand Meyer on the functional level and apply it to the object and architectural levels.
The basics of this is to really treat read and the write as two different pathways and to never mix them. This means that you never reuse any code between the two sides. Getting data through queries should never have side effects on the data as it can’t write anything. Commands do not return any data as they only get to perform the action it is set out to do. A command is a data holder that holds the data that is only specific to this command. It should never be looked on as a vessel for large object graphs that gets sent from the client.

Going beyond the separation of read and write, CQRS is really about creating the models needed for the different aspects of your solution (never reuse a model between them). This would mean that you will end up having read models, write models, reporting models, search models, view models, and so on. Each of these is highly specialized for their purpose, leading again to decoupling your system even further. CQRS can be deployed with or without events that connect the segregated parts at its core; this all depends on whether or not your organization can be event driven or not. We will discuss CQRS in more depth later in this book.

CQRS is often seen as complementary to **Domain Driven Design (DDD)** — a practice that focuses on establishing the model that represents the domain you’re making the software for (for more information, visit [http://dddcommunity.org/learning-ddd/what_is_ddd/](http://dddcommunity.org/learning-ddd/what_is_ddd/)). It holds terminology for how you do this modeling and defines well-defined patterns for the different artifacts that make up your domain model. At the core of this sits the idea of a ubiquitous language that represents your domain; a language that is not only understood by a developer, but also by the domain experts and ultimately the users of the system. Some key facts that you need to bear in mind are as follows:

- Avoid unwanted confusion and also avoid having the need for translations between the different stakeholders of a software project.
- Capture the real use cases and how you should focus on capturing them while not focusing on technical things. Often, we find ourselves not really modeling the business processes accurately, leading to monster domain models that potentially could bring the entire database into memory. Instead of this, focus on the commands or tasks, if you will, which the user is performing. From this, you will reach different conclusions for things (such as transactional boundaries).
- Bounded contexts, another huge aspect of DDD, is the idea that you necessarily don’t have one big application but rather many small ones that don’t know about each other and live in isolation only to be composed together in the bounded context of the composition itself. Again, this leads to yet another level of decoupling.
The following diagram shows the division found in a full CQRS system with event sourcing and the event store and how they are connected together through events being published from the execution side.

**Libraries and frameworks**

We will not be doing much from scratch in this book as it does not serve our purpose. Instead, we will be relying on third-party libraries and frameworks to do things for us that don't have anything to do with the particular thing we will perform. The range of libraries will be big and some of these represent architectural patterns and decisions sitting behind them. Some of these are in direct conflict with each other and for consistency in your code base, you should be picking one over the other and stick to it. The chapters in this book will make it clear what I consider as conflict and why and what libraries are right for you, whereas your architecture is something you will have to decide for yourself. This book will just show a few of the possibilities.
jQuery
Browsing the Web for JavaScript-related topics often yields results with jQuery mentioned in the subject or in the article itself. At one point, I was almost convinced that JavaScript started with $, followed by a dot, and then a function to perform. It turns out that this is not true. jQuery just happens to be one of the most popular libraries out there when performing web development. It puts in place abstractions for parts that are different between the browsers, but most importantly, it gives you a powerful way to query the Document Object Model (DOM) as well as modify it as your application runs. A lot of the things jQuery has historically solved are now being solved by the browser vendors themselves by being true to the specifications of the standards, along with the standards. Its demand has been decreasing over the years, but you will find it useful if you need to target all browsers and not just the modern ones. Personally, I would highly recommend not using jQuery as it will most likely lead you down the path of breaking the SOLID principles and mixing up your concerns.

SignalR has a dependency on jQuery directly, meaning that all the web projects in this book will have jQuery in them as a result.

ASP.NET MVC 5
Microsoft's web story consists of two major and different stories at the root level. One of these is the story related to web forms that came with the first version of the .NET Framework back in 2002. Since then, it has been iteratively developed and improved with each new version of the framework. The other is the MVC story, which was started in 2007 with the version 1 release in 2009 that represents something very different and built from the ground up from different concepts than found in the web forms story. In 2014, we saw the release of version 5 with quite a few new ideas, making it even simpler to do the type of decoupling one aims for and also making it easier to bring in things (such as SignalR). We will use ASP.NET MVC for the first samples, not taking full advantage of its potential, but enough to be able to show the integration with SignalR and how you can benefit from it.

KnockoutJS
It seems that over the last couple of years, you can pretty much take any noun or verb and throw a JS behind it, Google it, and you will find a framework at the other end of it. KnockoutJS (http://www.knockoutjs.com) represents a solution to MVVM for JavaScript in the web browser. It's a focused library with the aim of solving the case of having views that are able to observe your ViewModel. It also takes advantage of any behavior being exposed.
Bifrost
In some of the chapters, a platform called Bifrost (which you can access at http://bifrost.dolittle.com/) will be used. It's an end-to-end opinionated platform that focuses on CQRS and MVVM. It also shows a few other things, such as convention over configuration (http://www.techopedia.com/definition/27478/convention-over-configuration), along with a few ways of decoupling your software. The platform is open sourced and it's worth mentioning that I am the lead developer, visionary, and initiator of the project. The project got started in 2008 as a means to solve business cases while working on different projects.

Within Bifrost, you will only find things that are based on real business value, rather than imagined solutions to problems that have never been experienced.

When Bifrost is applied, there are other dependencies it pulls in as well, which will be discussed when they are being used. A few of these dependencies introduce a couple of other aspects of software development and will be explained once they are used.

Making it look good – using Twitter bootstrap
In the interest of saving time and focus more on code, we will "outsource" the design in this book and layout to Twitter bootstrap (which you can access at http://getbootstrap.com). Bootstrap defines a grid system that governs all layouts and it also has well-defined CSS to make things look good. It comes with a predefined theme that looks great, and there are other themes out there if you want to change the themes.

Tools
As with any craft, we need tools to build anything. Here is a summary of some of the tools we will be using to create our applications.

Visual Studio 2013
In this book, you will find that Visual Studio 2013 professional is being used for all cases for development except for Chapter 8, Putting the X in .NET – Xamarin, which makes use of Xamarin Studio. You can use the community edition of Visual Studio 2013 if you don't have a license Visual Studio 2013 professional or higher. It can be downloaded from http://www.visualstudio.com/.
NuGet

All third-party dependencies and all the libraries mentioned in this chapter, for instance, will be pulled in using NuGet.

In the interest of saving space in the book, the description of how to use NuGet sits here and only here. The other chapters will refer back to this recipe.

If you need to install NuGet first, visit http://www.nuget.org to download and install it. Once this is done, you can use NuGet by following these steps:

1. To add a reference to a project, we start by right-clicking on References of your project and selecting Manage NuGet Packages, as shown here:

2. Next, select Online and enter the name of the package that you want to add a reference to in the search box. When you have found the proper package, click on the Install button, as shown in the following screenshot:
In some cases, we will need a specific version of a file. This is not something we can do through the UI, and we will need the package manager console.

3. Following this, go to **TOOLS** and then **NuGet Package Manager**. Click on **Package Manager Console**, as shown here:
4. You then need to go to the **Package Manager Console** window that appears and you need to make sure that the project that will have the reference is selected:

![Package Manager Console](image)

By now, you should be familiar with how you can add NuGet packages to reference third-party dependencies, which will be used throughout the book.

**Summary**

You now have a backdrop of knowledge, if you didn't already know it all. We explained the terminology in this chapter so that the terms will be clear to you throughout. It's now time to get concrete and actually start applying what we've discussed. Although this chapter mentions quite a few concepts and they might be new to you, don't worry as we'll revisit them throughout the book and gain more knowledge about them as we go along.

The next chapter will start out with a simple sample, showing the very basics of SignalR so that you get the feeling of what it is and how its APIs are. It will also show you how to do this with ASP.NET MVC and throw in the mix of the usage of bootstrap and jQuery.
Where to buy this book
You can buy SignalR Blueprints from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.