This book will help you to understand the fundamentals behind the ServiceNow platform. Throughout the book, we develop a simple example application that is designed to highlight the key capabilities of ServiceNow.

In the initial chapters, we build our foundations, walking through the data model, exploring how to write effective code to implement business logic, and avoiding the pitfalls that come with flexibility and choice in client-side scripting.

Then, we explore the Task table, a powerful kick-start to any application. To head off potential problems, we look at debugging and diagnosis techniques. And finally, we think about automation and orchestration—how ServiceNow can help control the rest of your IT environment. Packed with hints, tips, and examples, this book helps you to master ServiceNow quickly and efficiently.

Who this book is written for
This book is intended for those who are familiar with web technologies such as JavaScript. We start from the computing fundamentals to understand how ServiceNow is architected, working from the ground up to really leverage the full capabilities of the platform.

What you will learn from this book

- Dig into the foundations of the ServiceNow platform and build powerful custom applications
- Design feature-rich, responsive, automated workflow systems
- Leverage the vast capabilities of ServiceNow to super-power your business
- Design powerful data-driven applications
- Control information flow and apply business logic with Business Rules
- Write efficient and effective client-side JavaScript
- Explore the working of the Task table
- Integrate and exchange data with people and systems
- Create and secure your systems with proper access control
- Discover and automate other IT systems

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'ServiceNow Foundations'
- A synopsis of the book’s content
- More information on Mastering ServiceNow

About the Author

Martin Wood has spent the last few years spreading the word about ServiceNow. His first exposure to the platform was as one of its first customers in the UK in 2008, when he built a custom ITSM application. He enjoyed the experience so much that he joined the company. Now, he works with a variety of clients, ranging from blue-chip enterprises to SMEs, helping them utilize the power of ServiceNow. Martin is passionate about helping people make an informed choice and enjoys speaking at ServiceNow user conferences.

He lives in the beautiful Berkshire countryside with his wife, Sarah. They both love exploring the world and enjoying good food and great wine!
Mastering ServiceNow

Congratulations! You have just become the ServiceNow System Administrator for Gardiner Hotels—one of the most successful chains in the region. The CIO realized that ServiceNow exactly fits his needs; a suite of feature-rich applications built upon a robust platform that makes the process of configuration and customization quick and easy. The SaaS architecture means that installation time is virtually nonexistent, which means that you can concentrate on what you want to do. The single tenancy model gives you power to configure and customize the system as needed, but without worrying about hosting or availability. You are in control of this power!

The simplicity of the basic operation is a little alluring. You can easily log in and start working. However, don't be fooled! Due to the breadth and depth of options, ServiceNow can quickly become overwhelming. Just like any other technology, ServiceNow has its own language, best practices, and idiosyncrasies. This is where this book comes in. By focusing on key areas within the platform, we look beyond the basics, in order to understand exactly what is going on. After reading this book, you should have a much better knowledge of how ServiceNow is best utilized, enabling you to create new custom process applications, and configure and customize built-in functionalities. All the applications you may use are built on top of the ServiceNow platform; by understanding this, you will be able to tackle almost any app.

What This Book Covers

Each chapter dives deep into the foundations of the platform. The book focuses on the specifics of ServiceNow, not what JavaScript is. We'll look at the hows, but more importantly, the whys. We'll explore options, and there is advice, but always along with justification. By understanding how things work, you will be better equipped to make your own decisions.

In this book, you will be configuring ServiceNow to be more useful to Gardiner Hotels. We'll achieve this by building out a sample Hotel Management application designed to showcase many of the capabilities within ServiceNow.

Chapter 1, ServiceNow Foundations, looks at how ServiceNow is structured from an architectural perspective. We explore how the platform is hosted and then dive into the building block of tables, fields, and building interfaces.

Chapter 2, Server-side Control, shows how you can implement your business logic, and then start to automate, validate, and verify data and processes.

Chapter 3, Client-side Interaction, explores how you can make the life of the people using your application just a little bit better by providing validation, feedback, and quick interaction techniques.
Chapter 4, Getting Things Done with Tasks, the Service Catalog and Service Portfolio looks at some of the base application functionalities in ServiceNow. Understand how a task-driven process system is kickstarted by the Task table, by taking advantage of Graphical Workflow, the Service Catalog and Service Level Management.

Chapter 5, Events, Notifications, and Reporting, introduces another level of interaction with your users, by generating reports and scheduling jobs, and handle incoming and outgoing email. Keep everyone informed about what's happening.

Chapter 6, Exchanging Data – Import Sets, Web Services, and Other Integrations, is about importing data from other systems, integrating ServiceNow in your application landscape. No instance is an island!

Chapter 7, Securing Applications and Data, focuses on the challenges of protecting your most important assets: your information. We make sure the right people have the right data.

Chapter 8, Diagnosing ServiceNow – Knowing What Is Going On, helps you when things go wrong. Troubleshooting and investigation hints and tips are explored, so you can get back to full power quickly.

Chapter 9, Moving Scripts with Clones, Update Sets, and Upgrades, builds on the previous chapters to explore how you can get your hard work in the right place. Understand how upgrades work, and how teams can work together to get stuff done.

Chapter 10, Making ServiceNow Beautiful with CMS and Jelly, focuses on creating a self service portal, but also discusses how the ServiceNow interface is built. Having a good looking, well-designed frontend really enhances adoption.

Chapter 11, Automating Your Data Center, looks at having ServiceNow in charge of your IT. By discovering what is out there, then automating the maintenance and deployment of the new functionality, ServiceNow will become an invaluable part of your business.
This opening chapter picks out the most significant foundations of ServiceNow, starting from the bottom up. Understanding the fundamentals of the ServiceNow platform is important. It gives insight into the concepts that underpin how everything else works.

Although long, the chapter is not exhaustive and does expect a basic familiarity with the ServiceNow interface. Remember to reference the ServiceNow documentation and any training material you may have.

Perhaps you've decided to build a new hotel and you want to ensure it won't fall down. The architect's drawings need to be understood and the right building materials ordered. It's costly (and career limiting!) if it collapses in the week after opening!

In this chapter, we review the blueprints. We understand the important design aspects of ServiceNow so that we can build on them later. The data structure available to us is critical, to enable us to model information and processes in the right way.

In this chapter, we will cover:

• The physical components of the ServiceNow architecture
• How everything you see and do is in the database
• A review of the most important field types
• The magic of reference fields and table inheritance
• Using and building a good interface
Diving into the infrastructure

An instance is several things. It is a URL (something like https://<instance>.service-now.com/); it's software running in the cloud; it's your copy of the ServiceNow platform.

ServiceNow provides a platform and suite of applications as a service. They worry about the hardware, Internet connectivity, and operating system security and provide you with the URL. All you need to get going is a web browser.

Being in charge

An instance is an independent implementation of ServiceNow. It is isolated and autonomous, meaning your instance is not shared with other customers. ServiceNow uses a single-tenancy architecture, which means your instance is yours: you can do what you want with it, such as changing logic, updating the UI, and adding fields.

Every customer has several instances; again, all isolated and independent. One instance might be marked out for developing on, another for testing, and one for production. And, because each instance is independent, each one can be running a different release of ServiceNow. The production instance only differs because it has more powerful hardware.

Chapter 9, Moving Scripts with Clones, Update Sets, and Upgrades, discusses how you can use your instances for tasks such as building functionality, testing it, and then making it live.

Changing your instance

A new instance starts with a few ServiceNow applications, some demo configuration, and example data. This is often called the out-of-the-box state. One of the example data elements is the System Administrator user. You are able to log in and get going, getting full control immediately.

Everyone makes changes to their instance. Unless the people who will be using the system are called Beth Anglin or David Dan (some of the default example users), you'll need to load some users at the very least. Some ServiceNow customers configure a lot and some do the bare minimum. You can choose how much you wish to do. Because it is single-tenant, you can alter the configuration and data in almost any way you see fit. Now, it might not always be smart to do that, but you can!
My favorite analogy, if you haven't guessed it, is a building. ServiceNow gives you an office that is yours. It starts off identical, built to the standard plans, but you can redecorate or remodel as you see fit. Perhaps even knock down a wall! (Let's hope it's not load-bearing.) This is the benefit of single-tenancy.

Multitenancy might be an apartment in a block. It is generally more efficient to pack lots of people together in a single building, and you can build it pretty high. However, you don't have the flexibility that being in control gives you. The landlords of the block won't let you knock down a wall!

The vast majority of customers have their instance hosted by ServiceNow. This means the people who built the house will also look after it, on their land. You get great economies of scale, and the benefit of tools and automation design to perform maintenance and give support fast. All the gardeners and janitors are on site, ready to work—they don't need to travel to help out.

Choosing functionality with plugins

All ServiceNow functionalities are delivered as plugins. When an instance is turned on, one of its first tasks is to load all the plugins that are turned on out of the box. There are quite a few of those, over 200 in the Eureka version of ServiceNow. And there are several hundred more that you can turn on if you want. A plugin may provide an app, like Human Resources Management, or provide new platform functionality, like Domain Separation. Each new version of ServiceNow brings new plugins and updates to existing ones.

Chapter 9, *Moving Scripts with Clones, Update Sets, and Upgrades*, talks about plugins and upgrading ServiceNow in more detail.

When a plugin is turned on, all the data and configuration that the application needs is loaded into the database, meaning that it is ready for work in just a few moments. Many also contain demo data, giving you examples of how it could work.
Digging into hosting

A ServiceNow-hosted instance is split over two physical datacenters, a high-availability pair. Each location runs independently of the other, giving a semiclustered environment. In the event of a catastrophic disaster, with one location being completely unavailable, the other nodes will just pick up the load, with almost no downtime. In fact, the process of switching between locations is used for maintenance procedures, enabling your instance to be well-protected against hardware and other failures.

When you visit your instance, you are directed through several layers:

- By looking up the DNS records, you are directed to the currently active datacenter
- The load balancer, by reading a cookie, directs you to the application server you have your session with
- If you aren't logged in, you get directed to the least-busy application server
- Your application server then uses the database currently determined as active
Knowing the nodes
From an architecture perspective, a ServiceNow instance is made up of several application and database servers or nodes. These are generally running on shared hardware, meaning that although your instance is logically separate and independent, it is physically hosted alongside another customer. At each location, there are generally at least two application nodes, each running a copy of the ServiceNow platform, which work together to share load. Additionally, there may be worker nodes installed to process the non-interactive jobs, such as event processing. Even though you'll never directly log in to these worker nodes, they perform some background processing, allowing the interactive application servers to respond more quickly to user requests. For example, a worker instance might send out e-mails or deal with integrations. While there are generally lots of application nodes, there is only one active database server, running on a separate physical server. It does have a redundant pair hosted in the remote datacenter.

Chapter 5, Events, Notifications, and Reporting, explores the concept of event queues in more detail.

Exploring the database
So you’ve got an instance and have logged in. Great! What can we see? We can see database records.
You may not realize it, but the homepage, the reports, and the menus to the left are all database records:

Almost everything in ServiceNow is an entry in a database. When you look at the user interface, virtually everything you see—from the data typed in by a user, to log files, to how the views are structured—is stored in the instance’s relational database. Even the scripts you write are kept in a string field in a record, and the files you upload are stored in chunks in the database.

Everything is built on the top of this structure. You don’t need to reboot the server to apply new functionality; you are just updating data records. You don’t need to reload configuration files—any properties you set will be read on the next operation. Even the database metadata, information about the fields themselves, is stored in another table.
This gives you extraordinary control and ability. You can organize, search, and manage the data in an unprecedented manner. You can find scripts the same way you find users, by searching tables. You can control and secure any data, regardless of what it is, by using Access Control Rules. This means you can focus on designing and building great business applications, since the platform works in a consistent manner.

ServiceNow may be considered a high-level platform that is based on the concept of Model-View-Controller. When building a ServiceNow application, you can first think of the data structure. You determine what information you need to store and how it all links together, creating tables and fields. This is the model aspect.

Automatically, you get a simple view on this data, with forms and lists showing your information.

And you can build simple ways to manipulate and change the data, through automation and simple manual updates. This is the controller aspect.

**Introducing the Gardiner Hotel data structure**

One of the first things that many people learn how to do in ServiceNow is to add a field. In ServiceNow, this is a straightforward operation—you add the new field to a form or a list using the UI. Under the covers, the ServiceNow platform is performing, among other operations, a simple SQL command to add a column to the table you are manipulating. When you add a field, you add a column to the table. When you remove a field, you are dropping it. There is no magic—the platform interface just makes it easy to do.

ServiceNow allows you to add fields to every table in the system. If you decide that adding another is a good idea, you have the power to do it!

In order to work through the ServiceNow functionality, we are building a hotel management application for your new employer, Gardiner Hotels. It will involve building a simple data structure, but one that is highly interlinked.
Here is a representation of the tables we will create in this chapter:

![Diagram of tables](image)

This diagram represents several common activities within a hotel, and their links to each other.

- **Guests**: The reason why Gardiner Hotels exists! Our guests' details are the most important information we have. We definitely need their names, and optionally their e-mail addresses and phone numbers.
- **Room**: This represents where our guests will sleep. We store the room number and the floor it is on.
- **Check-in**: When guests want their room key, they check in. We record who is checking in to a room, when, and who made the entry. We will create a link to the **Room** table so we can easily see information about the room, such as what floor it is on.
- **Reservation**: Our guests like staying with Gardiner Hotels, and they often book months in advance. One reservation might be for many guests, especially if the whole family is coming. A big family might need a big room. We need to record where exactly they may stay.

Over the course of the book, we will expand and further develop the application. Its primary use is to show you as much of the capability of ServiceNow as possible, so some of the examples may be done better in other ways.
Creating tables

Firstly, let’s create an app to hold our configuration in. Navigate to System Applications > Create Application. Fill in the following value and click on Submit:

- **Name:** Hotel

The convention for navigating through ServiceNow uses the following structure: Application Menu > Module. For modules with separators, it will be Application Menu > Section > Module. The easiest way to find a particular link is to type it in to the Application Filter at the top left of the menu. Make sure you are choosing the right one, though, because some modules are identically named.

Then find the Tables Related List, click on New, and fill in the following values:

- **Label:** Check-in
- **Name:** u_check_in
The rest of the default entries should be fine.

Many technical items in ServiceNow have a label and a database name. In this case, the database table is called u_check_in, while an entry in the Field Labels [sys_documentation] table contains the mapping between the name and the label. I'll use the Label [database_name] format throughout the book.

When you click on the Save button, the application server creates the table in the database.

I recommend using the Save button to commit records to the database, accessible via the 3-line menu icon, or by pressing Ctrl + S. This ensures that the record is renamed on screen rather than redirecting to the previous page.

**Adding fields**

When you create a new table, you get some system fields. They include two dates (when a record was created and when it was last updated), two string fields (containing the user ID of who created it and who updated it), a unique GUID called the sys_id, and a field that counts the number of updates to the record. They are all updated automatically, and it is generally good practice to leave them alone. They are useful just as they are!

We'll be discussing the sys_id field a lot in just a moment!

The following screenshot shows how the system fields are represented within the tables:
The autogenerated fields are very helpful to the System Administrator for finding records quickly. I always add the *Updated on* field to my lists, since it makes finding the records I've been working on (such as scripts) much faster.

In addition to these automatic fields, we need to create some of our own. We will need several, but right now, let's create something to store any requests that the guest may have. Perhaps they may have specifically requested a high floor.

Create a new field called **Comments** by double-clicking on **Insert a new row**. Fill out the row with the following data and then save the record:

- **Column label**: Comments
- **Type**: String
- **Max length**: 500

The following screenshot displays how the columns list looks before saving the changes:

![Table Columns Screenshot]

All the fields and tables that you create will get a *u_* prefix in the name, while the label will stay as is. This helps you (and the upgrade process) to separate them from the out-of-the-box functionality.

Upgrades are discussed in more depth in Chapter 9, *Moving Scripts with Clones, Update Sets, and Upgrades*.

**Knowing what's happening**

Behind the scenes, the application server is running SQL commands against the database. Specifically, at the time of creating the field, the following was executed:

```
ALTER TABLE u_check_in ADD 'u_comments' MEDIUMTEXT
```
If you wish to see these commands, navigate to System Diagnostics > Debug SQL. This will place lots of information at the bottom of the page. Other diagnostic tools like this are discussed in Chapter 8, Diagnosing ServiceNow – Knowing What Is Going On.

This demonstrates a key concept. Whenever you perform an action in ServiceNow, it results in a string of database commands. The database is altered and the platform’s internal state is updated. These actions are generally carried out quickly, with the whole process completed in about half a second. No downtime is necessary.

**Introducing the dictionary**

The dictionary is a metatable. It describes your table and fields—what their name is, how big they are, along with any special attributes they may have. For example, one field might be dependent upon another. A dictionary entry might also be referred to as an Element Descriptor.

The table is represented by an entry in the dictionary with a table value, type of Collection, and no column name. When the Comments field was added to the Check-in table, the platform also made a dictionary entry with the table and a column name. You can view it by navigating to System Definition > Dictionary.

The options that are available in the dictionary are dependent upon what type of field you are working with. Reference fields, which we will explore in a moment, have extra functionality that is controlled here, such as dynamic creation.

As we work through ServiceNow, we’ll spot functionality that is enabled through the dictionary. However, much of it can be achieved in other ways, often in a better manner.

Older versions of ServiceNow have a read-only tick box available on the dictionary form by default. While it can be included and used in later versions, it is usually a better idea to use security rules instead. Having a field marked as read-only is a binary choice, and giving System Administrators control over data is a good idea. You get that granularity with security rules, which we will explore in Chapter 7, Securing Applications and Data.

The easiest way to navigate to a dictionary entry is by right-clicking on the label of the field and choosing Personalize Dictionary. You can see some details about a field, such as what table it is in, by choosing the Show option on the menu that appears on right-clicking.
The Globally Unique Identifier

The ServiceNow database is a relational database. This means that one bit of data can relate to another. To ensure that every record or row can be referenced easily, every record has a unique identifier: a **primary key**.

In ServiceNow, this primary key is something that isn't related to the data itself. It is a **globally unique identifier** or GUID. This GUID is stored as a 32-character string, made of hexadecimal characters (the numbers 0-9, plus the letters a-f). The number of unique GUID values is so large that the probability of two accidently being the same is negligible. This is an example GUID: 5137153cc611227c000bbd1bd8cd2005.

This type of identifier is sometimes known as an **OID** or **object identifier**. It has no special significance; it just uniquely identifies a data row. It can also be called a **surrogate key**.

Whenever you create a record in ServiceNow, the platform generates a new GUID. The characters that are generated are random—a mixture of several sources, including the date and time and details specific to the instance, meaning it is not sequential or predictable. The GUID is saved alongside the record, in a special field called `sys_id`. The `sys_id` field is heavily used in the ServiceNow platform—you will start seeing GUIDs everywhere!

As an example of how ServiceNow uses the `sys_id` field, conduct the following experiment. Construct a URL similar to the one that follows, substituting `<instance>` with the name of your instance, and visit it with your browser:

```
```

In a new instance, you should happen across the user record of Fred Luddy. (If the demo data has been removed, you will get a **Record not found** message.)

It is useful to examine the structure of the URL. Firstly, spot the `sys_user` table. Then spot the GUID. With these two items, the instance knows exactly what data it needs to pull up and present to you.

Every record has a `sys_id` field. It is the only field that ServiceNow really cares about. It also looks after itself. You don't need to worry about it during day-to-day operations.

Reference fields, as we'll see, are very reliant upon the `sys_id` fields. When we get into scripting in **Chapter 2, Server-side Control**, you'll be seeing more of them!
Every other field is non-mandatory and non-unique to the database platform. You can have two records that are otherwise identical but only have a differing sys_id field. (It is possible to enforce uniqueness in other fields too, as we’ll see later.)

This means that, in general, you can change the value of fields to whatever you like and still maintain referential integrity; no system errors will occur. If you want to rename a user or a group, go ahead. Since everything related to that user will be associated to it via the sys_id field, the name of the user is not important.

Many other products do not use surrogate keys; data is linked together using user-provided data. If you change the name of a group, for example, this can remove all group memberships and assignment to tasks. Not in ServiceNow!

An important exception to this behavior is with roles. Roles are referred to by their name in scripts, so if you change the name, all scripts that use it will need to be altered (though security rules do refer to the role through the sys_id field.) In general, it is a good idea to keep the name of roles the same.

It is a good idea not to interfere with this flexibility. When you are building functionality, try not to refer to records using their name or sys_id in scripts. Instead, use the properties or attributes of the record itself to identify it. So, rather than hardcoding that a particular room in our hotel needs special treatment, create another field and use it as a flag. The VIP flag on the User table is a good example of this.

Building hierarchical tables

ServiceNow is built on a relational database. Instances hosted by ServiceNow use MySQL—a popular open source database that is robust, well featured, and scalable. Relational databases are relatively simple to understand, which is one of the reasons why they are most commonly used; data is held in a tabular format with each table storing information about a particular item. Relationships may exist between these items. Our database design described in the previous section has four tables, each relating to the others. For example, the Check-in table will relate to the Guest table to know who checked in.

The ServiceNow platform can run on almost any relational database, such as Oracle or SQL Server. But supporting different architectures is difficult, so it is not a standard offering.
Benefiting from an object-oriented design

The simplicity of a relational database means that, on its own, it does not easily represent the data structures used in modern object-oriented programming languages. One particularly useful function of an object-oriented approach is inheritance.

Inheritance allows one object to build on the functionality of another. Why duplicate effort when you can reuse existing capability?

In ServiceNow, a table can inherit another. The parent table defines the base functionality, while the child table, built on top, can continue to use it. That means that any fields you add to the base table are automatically available to the child too. In fact, almost all functionality you add to the base table is available to the child.

In our Hotel application, we want to store information about our guests. We need to know their names, their telephone numbers, and perhaps their addresses. ServiceNow has got a built-in table for storing people—the User table. But we want a special type of person—guests. Let’s keep staff in the User table and guests in a new extension table.

The User table in ServiceNow defines who can log in and use the platform functionality. Sometimes you need a contact database, which stores information about people: their names, phone numbers, location, and who their manager might be. It's tempting to build the contact database as a separate table and keep the two separate, but I recommend using the User table as the basis for both. It saves the duplication of data and allows reuse of a special functionality that is built specifically for the built-in table. But do be aware of licensing.

Extending the User table

Let’s extend the ServiceNow User table in order to have a special class for guests. First, we have to mark the User table as extendable. Go to System Definition > Tables and find the User [sys_user] table. Make the following change and save once you’re done:

- Extensible: <ticked>

It is likely you will get a pop-up message asking if you want the User table to be included in the Hotel application. Click on Leave Alone; we want to keep things as they are. We don’t want to include the User table in our application, only the Guest table.
Now, go to our application by navigating to **System Definition > Custom Applications > Hotel** and create another new table. Use the following data:

- **Label**: Guest
- **Name**: u_guest
- **Extends table**: User

If you look at the fields available on this new table, you’ll see lots of fields already, beyond the normal automatic five. These additional fields are those defined on the **User** table.

![Field Table](image)

What does this mean? The **Guest** table has *inherited* the fields of the **User** table. This is extremely useful; I don’t need to create the name, telephone, and e-mail fields – they are already available for me.

Indeed, when you create a table that inherits another, you gain all the functionality of the parent. Most of the scripts, rules, and policies of the parent automatically apply to the new table. But sometimes you want to create the functionality for the child table only. So, ServiceNow lets you place it at the level you need.

We'll cover how scripts are handled in ServiceNow in the next chapter.

**Interacting with hierarchical tables**

Our new table is the right place for storing information about our valued customers. While useful fields have been inherited from the **User** table, it doesn’t contain everything. Make a new field to store their membership number. For simplicity, create a text field with a Column label named **Membership Number**.
To do this, click on Design Form on the Guest table record. Find our new field from the left-hand-side list, drag it underneath the Last name field in the layout, and click on Save:

[Diagram showing the Design Form interface with the field being dragged]

Check out the wiki if you need help on this step: http://wiki.servicenow.com/?title=Form_Design.

To test, let's create a new guest record. Navigate to Hotel > Guest and use this data:

- **First name**: Alice
- **Last name**: Richards
- **Membership number**: S2E1

Great! We can enter a membership number fine. And if we look at a standard User record such as Fred Luddy (User Administration > Users), the Membership number field does not show up.
Viewing hierarchical tables
You may have noticed that our new guest, Alice, showed up when you visited the User table. That's because Alice is both a user and a guest. A Guest record will be treated just like a User record, unless there is something specific that overrides that behavior. In our case, the only difference between a User and Guest right now is that the latter has an extra field.

This behavior is called polymorphism, and is a very useful functionality. Use the base or extended functionality, as you need it.

But this gives rise to something that confuses many. If I look at the Guest table, I can add, through Personalize List, the Membership number field.

However, if I try to add a Membership number on the User table, I can't.
This is because a User doesn't have a membership number; only a Guest does. Think carefully about where you position fields to ensure they can be seen at the right level.

You can turn on a property that lets you view inherited fields in the base table. This is discussed more in the Dot-walking section in this chapter, as well as the Reporting section, in Chapter 5, Events, Notifications, and Reporting.

Overriding field properties
Inherited fields allow you to easily reuse functionality. However, sometimes, you want the fields in the extended table to work differently to the base table. This is accomplished with dictionary overrides.

For example, let's change the default time zone for new guests so that it's different to that of the User table. The current default for Users is the system time zone, and Guests inherits this setting.

Navigate to the dictionary entry for the Time zone field (perhaps through System Definition > Dictionary or by right-clicking on the label) and look for the Dictionary Overrides tab. Click on New. Use the following data:

- **Override default value**: <ticked> (tick this option)
- **Default value**: Europe/London (or your own choice!)

Now, when you create a new Guest record, it sets the default time zone to be Europe/London. Any new User records will be unaffected.

You can also change field labels so that they are different for the base and extended tables. Navigate to System Definition > Language File and create a new entry, populating Table with the extended table name (such as u_guest). The Element field should be the field name.

Understanding the background behavior
You might be wondering how this all works.

A child table is a normal database table. However, it does not recreate all the fields of the parent. Instead, the only columns in that new table are the new fields. For example, if I were to run the DESCRIBE u_guest SQL command on the database, I'd only see two fields: u_membership_number and sys_id.
So, when I look at Alice's record on the Guest table, the ServiceNow platform is actually joining the parent table and the child table together, behind the scenes. ServiceNow takes the independent tables and (invisibly) joins them together, creating the illusion of a single, bigger table.

Our friend, the sys_id field enables the platform to do this. If you remember, the sys_id field uniquely identifies a record. In the case of an extended table, the sys_id field is actually stored in two places: on both the parent and child tables. The platform joins both together whenever you query the Guest table.

When you mark a table as extendable, you are also adding a second system field: Class [sys_class_name]. It contains the name of the table that the record represents. For example, the User record representing Fred would have sys_user in the Class field, while the User record for Alice would be u_guest. With this information, ServiceNow can join tables if necessary and present you the appropriate data.

In the Dublin version of ServiceNow, this backend behavior has changed. There are now two models for table extension: hierarchical and flat. The hierarchical method consists of multiple tables that are joined together as needed just as described, while a flat structure consists of one very large table with all the columns of every table. When you make a new table and add a new field, in reality it is simply adding another column to the base table. The platform again hides this from the user. This does not have an impact on how table extension works in ServiceNow and is purely undertaken for performance reasons on larger instances.

The ServiceNow interface knows about this behavior. When you navigate to a record, ServiceNow will always show you its actual class. So, even if I am viewing a list of Users, when I click on Alice, I will see the Guest form, with all of the appropriate attributes. The Schema Map is really helpful to visualize what is going on. Navigate to System Definition> Tables, choose the table you are interested in, and then click on Show Schema Map. The following screenshot shows the Schema Map of the Guest table.
This wiki article has more information on the Schema Map: http://wiki.servicenow.com/?title=Schema_Map_for_Tables.

### Changing class

Once a record is stored in a particular table, a neat trick is to move it. If I decide that Alice is actually a user, I can alter the value of the Class field. The platform will drop any Guest-specific information and start working just like a User. The Class field can be added to a form or list, and is represented as a choice list. Often, you will want it to be read-only.

The ability to change the class is a powerful feature and you should be aware of the consequences. It is unusual to reclassify a record, and it may throw off reporting; for example, if you counted ten users and nine guests, and suddenly one switched, you might have an overbooking. It may also clear reference fields, which we are about to discuss.
So far, we've discussed how you can add fields into a specific class, and seen how they are inherited. But this will work with much more than fields! As we work through the chapters, we'll see how functionality such as Business Rules, Access Control Rules, and Import Sets all benefit from hierarchical tables.

**Storing data**

There are lots of different fields provided by the ServiceNow platform. We will explore some of the simpler ones first, before moving on to the fundamental backbone of the data structure with reference fields:

- **Journal input**: These fields provide an always-empty textbox designed to store notes, but does not store the information in the field on which it's created. Instead, when you enter text into a journal input field, it's saved in the `sys_journal_field` table.

- **Journal list**: They don't have any entry capabilities at all. Instead, you make it dependent upon a journal input field, and it shows all its entries by querying the `Journal Entries` table automatically. This allows you to have two journal input fields for one journal list. Journal lists are also known as Journal outputs.

- **Journal**: These are a combination of both Journal input and Journal lists, doing both jobs together.

- **Choice**: They are text fields, but rendered as HTML select fields. The value that is stored in the database is plain text. Another table, the `Choices` table, stores the options and labels. This lets the platform convert "wip" in the database to present "Work in Progress" to the user. Any values that don't have a label are highlighted in blue in the dropdown.

- **Integer choice**: These fields use numbers instead of text to achieve the same result. They are useful for representing states, since they allow you to use greater than or less than conditions, but have proven difficult to work with since the numbers don't mean much!

Use caution when dealing with the out-of-the-box integer choice fields, such as **State** on the **Task** table. If you reuse them (which is a good idea), you should always align your states to the existing ones. For example, 3 should represent **Closed**. If you do not align them, then users will be confused when reporting. This is discussed in detail in Chapter 4, *Getting Things Done with Tasks*. 
• **Currency**: They are string fields that combine the currency and the amount together. \textdollar{1,000} represents $1,000. The platform uses this information to provide conversions between different currencies. For example, if I prefer to see amounts in GBP, the platform will, if it has the latest currency rates, display £675.

• **Date**: There are several **Date** fields in ServiceNow. The time is stored as UTC in the database, and the appropriate display value is calculated by the user’s profile.

• **True/False**: These fields are simple Boolean values in the database. They are rendered as tick boxes.

• **URL**: These fields provide a space to enter a link, which can easily be made clickable.

• **HTML and Wikitext**: Other fields, like these, provide different interfaces to manipulate strings. It is tempting to use HTML fields in lots of places, but they do come with overhead, and browsers have different capabilities. For example, Firefox is able to encode an image into a data URI and store it in these fields, while IE is not able to. Test carefully if you want to use capabilities like this.

### Attachments

In addition to text, ServiceNow can also store binary data. This means that anything (images, music, or even a multitude of PowerPoint documents), can be saved in ServiceNow. Just like everything else, binary data is stored in the database. However, rather than using a blob field, binary data is split into 4k chunks and saved into the **Attachment Documents** [sys_attachment_doc] table. Each chunk of a file refers back to the **Attachments** [sys_attachment] table, where the filename, content type and size, and other metadata are stored.

An attachment is always related to another record. Information on this other record is stored with the other metadata in the **Attachments** table. For example, if a **Reservation** record had a PDF of the booking form attached to it, the **Attachment** record would contain the file name of the document, as well as the **sys_id** of the **Reservation** record.

We’ll see in later chapters that there are often better ways than manually adding attachments containing booking information. Why not have the e-mail come directly into ServiceNow? (We’ll see how in Chapter 5, Events, Notifications, and Reporting.) Or even better, have the guests perform the booking directly with ServiceNow? (Chapter 10, Making ServiceNow Beautiful with CMS and Jelly, shows us how to do this.)
Setting properties

One of the simplest ways to control the platform is to set properties. There are lots of things you can change by just clicking on a box or changing a value. And just like everything else in ServiceNow, the configuration properties that you set are stored in a table: the System Properties [sys_properties] table to be precise.

To see how many options you can choose, type in Properties into the filter text of the Application Navigator. The many matches are shown, including System Properties > UI Properties. This collection contains some very useful options, including how forms look and feel, if list editing is enabled, and whether Insert and Stay is always available. You may want to take some time to find out what they do.

Some properties are not categorized, but all are accessible via System Properties > All. This gives a large list—almost 700 in Eureka. This book will guide you to the more relevant ones, but many are documented on the wiki:


In older versions of the platform, the list is available by typing in sys_properties.list in the filter text of the Application Navigator.

Reference fields

When designing the data structure for a hotel, you may want to link a guest’s record with the room they have checked in to. It won’t be good for business if we don’t know who has checked in where! This is exactly what a reference field does; it creates a link between two records, one pointing to another.

When we examined the URL for a particular record, it contained two parts: the table and the sys_id value of the record. These are the two items needed to reference a record. So when you create a reference field, you need to select which table it should point to. And the contents of the field will be a 32-character string. Sounds familiar? Yep, you will be storing a sys_id in that field.
Reference fields are one of the most important items to understand in ServiceNow. The database sees a string field containing the sys_id, a foreign key. However, this is meaningless to a person. Therefore, the platform allows you to pick a field that will be displayed. For a person, this might be their name. Other records might have a user-friendly reference number. This is usually an incremental number; there are scripts that can generate one automatically. You can choose which field to show by ticking the Display field in the Dictionary entry, but this is only useful to the user. Only the sys_id is important to the platform.

Reference fields are used throughout ServiceNow, just like a proper relational system should. Scripting, lists, and forms all understand reference fields, as we'll see as we work through the chapters.

**Creating a reference field**

Let's think about something that the Hotel application needs—a room directory. Each room has several attributes that defines it: its room number, how many beds it has, and what floor it is on. We can represent this information as fields in a Room record, all stored in a dedicated room table. Once we have this, we can modify the Check-in table to record which room has been taken by which guest.

As before, navigate to the Hotel record under Custom Application, and create a new table with the following data:

- **Label:** Room
- **Name:** u_room

Create a field to store the room number:

- **Column label:** Number

  Don't use the Auto-number option, but create a new field using the Related List.

And another field to store the floor it is located on:

- **Column label:** Floor
- **Type:** Integer
The final result should look like this:

Create a few example records, giving each one a different number. Make several on the same floor. Perhaps one of them might be Room 101 on the first floor.
Note that the platform does not force you to choose different numbers for each record. Unless you mark a field as unique, or you create a rule to check, the platform will allow you to create them.

To mark a field as unique, you can edit the dictionary entry of that field. (You will need to configure the dictionary form and add the Unique checkbox.) By ticking that field, you are asking the database to enforce it. It does this by making that field a unique key. This has two impacts. Positively, it creates an index on that field. However, if a user attempts to save a duplicate value, they will get a message saying Unique Key violation detected by database. This can be a little jarring for a non-technical user. Try to catch the error with a Business Rule first.

Now that we have a list of rooms, we need to create the link between the Room and Check-in records.

A reference field can be referred to as a "one-to-many" relationship. Since a room may be checked in to multiple times (one day you have one particular guest, and the next day another may sleep in it after our fabulous cleaners have done their work), but for a single check-in, you can only select one room. You can only sleep in one bed at once!

A classic example of a one-to-many relationship is between a mother and her children. A child can only have one biological mother, but a mother can have many children.

The following diagram shows the relationship between Room and the Check-in record:
In the Check-in table, create two new fields. One for the room, using the following data:

- **Column label**: Room
- **Type**: Reference
- **Reference**: Room

And another for the guest:

- **Column label**: Guest
- **Type**: Reference
- **Reference**: Guest

Now, create a few example Check-in records. To simulate someone going into Room 101, create a new entry in the Check-in table. You may want to rearrange the form through Form Design to make it look a little better.

Note that when you try to select a guest, only Alice is available. Since the reference field is pointing to the Guest table, users are not shown.

You can view the Room-to-Check-in relationship from both directions. If you are on the Check-in form, you can see which room is in use through the reference field. The reference icon is very useful for viewing more details about the record—just hover over it.

If you hold the Shift key on your keyboard while you move your mouse cursor over the reference icon and then into the pop-up window, the information will remain until you click on the Close button. This is quite useful when copying data from that record without losing place.
You can easily view the relationship from the other perspective too. When you create a reference field, you can add a Related List on to the form or list of the referenced table. This will let you see all the records that are pointing to it.

Click on the reference icon next to the Room reference field to go to its record. On the Room form, navigate to Personalize > Related Lists. It will be named in this format: table->field; in our case, Check-in->Room. Right-click on the list headings and go to Personalize > List Layout and add the Guest field alongside Created.

The Related List gives you a New button. When you click on this, you will see the Check-in form but with the reference field already filled in. So, if we know which room we want to check a guest in to, we can navigate to the Room record, click on the New button in the Check-in Related List, and need not type in the room number again.

**Using Reference Qualifiers**

By default, a reference field can select from any record on the referenced table. However, often, you want to filter the results. For example, you may want to specify a guest as inactive, perhaps representing someone that won't be visiting Gardiner Hotels any longer. Therefore, let's filter out the inactive guests so they cannot be inadvertently checked in.
Reference Qualifiers allow you to do this. When you edit the dictionary entry of a reference field, you can specify the filter you want to apply. These can be specified in three different ways:

- **Simple:** Lets you specify which records should be returned using a condition builder.

  Simple is the default Reference Qualifier. Click on Advanced view in Related Links to see the other options.

- **Dynamic:** Lets you pick from prebuilt scripts. The choices they give often differ depending on the context of the record or the session. A good example is Me, one of the Dynamic Filter Options. This will return whoever is currently logged in, meaning that users who look at the same query will have personalized results.

  You can build your own Dynamic Filter Options by navigating to System Definition > Dynamic Filter Options. This is shown in Chapter 2, Server-side Control.

- **Advanced:** This is the original way to create Reference Qualifiers. It accepts an encoded query. JavaScript can be embedded in these queries, by prefixing it with javascript:. Creating a Dynamic Filter Option is the more reusable option.

  An encoded query is a field-operator-value triplet, separated by the caret (^) symbol. This string represents part of the where clause of the resulting SQL query. For example, active=true specifies all records where the active field is true, while active=true^last_name=Smith represents active being ticked and the contents of the last_name field being Smith. One easy way to get an encoded query is to build a filter in the list view, right-click on the result, and choose Copy Query.

For our Hotel application, let's use a simple Reference Qualifier. On the Check-In form, right-click on the Guest field label, and choose Personalize Dictionary. Set the Reference qual condition field to Active - is - true.

Now, if you mark a guest as inactive, they cannot be selected when checking in, neither through the magnifying glass lookup window and nor using the type ahead functionality.
Dot-walking
Dot-walking is a very important concept in ServiceNow. It means you can access information through reference fields quickly and easily. It can be leveraged throughout ServiceNow — both through the interface and through code.

You've already used dot-walking. When you hover over the reference icon, you can see information from that record. That's the whole concept! We are using the platform's capability to 'see through' reference fields and pull out information from that record. And, as we'll see in the next chapter, the same is possible through code.

By default, extended fields are not available while dot-walking. The **Membership number** field would not be available when dot-walking through a **User reference** field.

The **Allow base table lists to include extended table fields** property in **UI Properties** changes this for the UI. Scripts can use a special syntax when dot-walking. This is discussed in more depth in *Chapter 2, Server-side Control*, and *Chapter 5, Events, Notifications, and Reporting*.

Using derived fields
Dot-walking can be used throughout the user interface. Another example is adding **derived fields** to lists, forms, and queries. A derived field is a field from another record that is found through a reference field.

For example, we could add the floor number of the room as a derived field on the check-in form. The floor number doesn't belong to the check-in record, and if we change the room on the form, the system will dynamically change what floor number is displayed.

With scripting, you have the option to copy data through the reference field onto the record you are dealing with. That data then becomes part of the record. Derived fields will exist through the link only.

This concept is important to understand. If the referenced record gets deleted or changed, it will then affect our current record. For example, if we delete the room record, the check-in form won't be able to show what floor it was on. If we change the floor value on the room record, our check-in form will show the new value.

The simplest example of information that is derived is the display value, which was mentioned earlier. If the display value of the referenced record changes, you'll see it altered everywhere. Since the **sys_id** is the primary key for a record, you can easily rename groups, alter the names of users, or update virtually any record without penalty.
Navigate to Personalize > Form Layout on the Check-in form. Even though the Room field is already added to the form, it is still in the available list. It should have [+] as a suffix to the field name, showing it is a reference field that can be dot-walked in to; for example, Room [+]. A green plus icon is made available on selecting a reference field. If you click on that icon, you get to see the fields in the Room table. Choose the Floor field and add it to the form. It should be labeled Room.Floor, showing that you are dot-walking. Click on Save.

Derived fields can only be added via Form Layout (rather than Form Design) as of the Eureka and Fuji versions of ServiceNow.

The Check-in form should now have several fields in it: the Room reference field, the Floor derived field, and a simple Comments field.

Dynamic creation
What happens if you try to associate with a record that doesn't exist, such as doing a check-in for a guest that has never been to the hotel before? If you type in the name into the reference field that doesn't match an existing record, then the red background warns the user that the record won't be saved properly. Indeed, if you try, you will get a message saying: Invalid update.
One way to create a record quickly is to use the reference picker and click on the New button on the list in the popup window. You then get a form that you can fill out, which comes with a Save button. But a faster way is to use dynamic creation, which allows you to type the name directly into the reference field.

Go to the dictionary entry of the Guest field on the Check-in table. Click on Advanced View Related Link, tick the Dynamic creation checkbox, and then click on Save. Now, if the guest doesn't exist, a green background will be present instead. This indicates that the record will be created.

By default, the new record will have the display value populated. For a User record, the display value is a field that is called Name that is actually dynamically created from the First name and Last name fields. The scripts on the user table will organize the data into the most appropriate place.

When you turn on Dynamic creation, you'll see an area in the dictionary for scripting a more complex scenario. Sometimes, this is very helpful, if only to flag that this record was dynamically created. Often, you want to track this is happening, since it is very easy to create lots of duplicates with dynamic creation; for example, is it Tom, Thomas, or Tommy?

Deleting records
When you click on the Delete button, the platform removes it from the database. But what if there is a reference field value that points to that record? For example, if we delete a Room record, what happens to the Check-in records? The Check-in table has a reference field that points to the Room table. What happens?
There are several ways that ServiceNow deals with it. The choice is set on the dictionary entry of a reference field:

- **By default** (or when the choice is set to Clear): the platform will empty all reference fields that point to that record. When deleting a Room record, all of the Check-in records that point to it will have the Room field emptied.

- **Delete or Cascade**: Any record that pointed to the deleted record is also deleted. This means that deleting a room would also delete all the Check-in records that pointed to it! **Delete no workflow** is an extension of this; with this option, only directly related records will be deleted (it does not cascade).

  Delete is a useful choice if the related record has no purpose without its referenced record. For example, if you delete a user, then a reservation makes no sense.

- **Restrict**: This option will stop the transaction if there are related records. If there are any records pointing to the deleted record, then abort the deletion. The platform would prevent you from deleting the room. This is the most conservative option, useful for preventing mistakes.

  For instance, once a room has been checked in to, you may not want to delete it. You should first contact the customers staying there and let them know that a wrecking ball may come through their walls!

- **None**: This option will mean that the platform does not alter any related records. Reference fields will stay populated with a sys_id that points to an invalid record.

The majority of the time, the default option of clear is the right choice. It does mean, however, that you lose information when deletions occur. So, in general, the best idea is not to delete anything! Users should be deactivated in production systems, not deleted.

If you accidently delete something, it may be found by navigating to **System Definition > Deleted Records**. The platform will even allow you to restore data that has been removed through a cascade delete. Check out the wiki for more information: [http://wiki.servicenow.com/?title=Restoring_Deleted_Records_and_References](http://wiki.servicenow.com/?title=Restoring_Deleted_Records_and_References).
This scenario also illustrates why the automatic fields are not reference fields, but instead copy information into the record. The Created By and Updated By fields store the text value of the user who performed the action, so they are not dependent upon the user record itself.

**Many-to-many relationships**

The other type of relationship between records is many-to-many. The relationship between siblings is many-to-many. I can have many brothers and sisters, as can they. But how can I store this? A reference field can only point to one record. Adding a lot of reference fields into a form is one way. Each reference field could point to another sibling. However, that's not great design. What if there were five populated reference fields and another brother or sister was born?

Instead, we could create another table that sits in between each target, acting as the "glue" that sticks the two together. A special many-to-many table can then have two reference fields, each pointing to a different side.

In the **Hotel** application, we want to take reservations for our guests. Each reservation might be for more than one person, and each person might have more than one reservation. This sounds like the perfect use for a many-to-many table:
This diagram shows how this might work out. Richard is staying one night on 1 Feb. That's easy enough. Lionel is staying two nights, on 1 Feb and 1 Mar. He liked our hotel so much that he came back, and encouraged his wife Lotte to stay with him.

The ServiceNow platform makes this a little easier to visualize, since it hides the complexities of the many-to-many table in most situations. It focuses on the records in the two target tables.

**Building a many-to-many table**

Let's begin building a many-to-many table. To do this, perform the following steps:

1. Create a new table named **Reservation**:
   - **Label**: Reservation
   - **Name**: u_reservation

2. Add the first date field to the **Reservation** table:
   - **Column label**: Arrival
   - **Type**: Date
   
   Then add another as seen here:
   - **Column label**: Departure
   - **Type**: Date

3. Finally, since we want to reserve a room, let's also create a reference field called **Room**:
   - **Column label**: Room
   - **Type**: Reference
   - **Reference**: Room

4. Then navigate to the **Many to Many** table. The easiest way to do this is to enter `sys_m2m.list` in the filter text in the **Application Navigator**.
The Application Navigator accepts a few shortcuts like this, `<table_name>.form`, will show the form of the table. For more information, refer to the following link:
http://wiki.servicenow.com/?title=Navigating_Applications#Using_the_Navigation_Filter

5. Click on New. Use the following data:
   - From table: Reservation [u_reservation]
   - To table: Guest [u_guest]
   - Type: Reference
   - Reference: Room

   The From and To tables are where we want to point our reference fields. It doesn't matter which way round you do it:

   ![Many to Many Definition](image)

   You'll see the other fields populate automatically. Make sure the table name makes sense – but leave the m2m part in, so you know what it is. The default (u_m2m_guests_reservations) makes sense in this case and isn't too long.

6. Click on Create Many to Many.

7. Go back to the Guest form. Navigate to Personalize > Related Lists and add in the new Reservations entry.
Adding fields to a many-to-many table

Sometimes, just having the two reference fields on the many-to-many table is enough. However, since it is a table, you can also add new fields to it. This technique is useful to identify something in particular about the relationship. Let's identify who the lead passenger in a reservation is:

1. Create a new field on the many-to-many table you created. You will find the table by going to System Definition > Tables.
   The table is called M2m Guests Reservations. Tidy it up to say Guest Reservations.

2. Add the following fields:
   - **Column label**: Lead
   - **Type**: True/False

3. Then go to the Reservations form and create a sample Reservation record.
4. Once done, add the Guests Related List to the form.
5. Once saved, right-click on the column headings of the Related List and go to Personalize > List Layout. Add in Lead.
6. Finally, right-click on the column headings again and navigate to Personalize > List Control. Once there, tick List edit insert row.

You can then use list editing on the Related List to record the information you want.
The Reservation record currently has no field set as the display value. By default, a field called Name is used. If that doesn't exist, the sys_id is used, and that's the case here.

Many-to-many tables are very flexible, but by using them, you lose some advantages of simple reference fields. The biggest disadvantage is that you can't dot-walk in the same way. This makes scripting more challenging.

Also, on a simple list view, you can't easily identify related records. One way round this is through hierarchical list views, which we will discuss later.

We'll see further disadvantages of many-to-many tables as we progress.

Deleting a many-to-many table
Deleting a many-to-many table isn't straightforward. You need to do it in two parts: delete the table and then delete the entry from the sys_m2m table. However, there are security rules that prevent you from deleting records on this table. You will need to disable or modify those rules to proceed. But beware of what you are doing!

Glide Lists
Glide Lists store an array of sys_id values. That means that one field can reference multiple records. One field can work in a similar way to a many-to-many table. In our earlier example, a Glide List field could be added in to the Reservations form instead, pointing towards the Guest table.

One disadvantage of Glide Lists is the interface. It is more difficult to interact with compared to other fields, both on the list and the forms. Since it contains multiple values, you can't dot-walk through it. Which one would you walk to?

When wanting to reference many records, consider the advantages of a Glide List against a many-to-many table:

- Glide Lists are represented as a field. They are more compact than many-to-many tables, and many built-in functions in ServiceNow accept the comma-separated reference fields as input. For example, a comma-separated list of users can easily be sent an e-mail. Glide Lists are usually simpler to deal with.
Many-to-many relationships are represented as records in a table. This gives no limit to the amount of records stored, and you can easily extend the functionality. You can add extra fields (like the lead passenger representation). It also has better hooks for scripts and other functionalities. Many-to-many tables are generally more flexible.

Building the interface

We've already spent some time with the ServiceNow interface. But understanding some of the fundamentals of how the platform is used and what it provides deserves repeating.

If we rewind to our opening consideration that everything is in a database, ServiceNow provides two basic interfaces that we spend the majority of our time with: forms and lists. The form and the list are the two major views of data within ServiceNow. You can, of course, create custom user interfaces, and we'll cover those in Chapter 10, Making ServiceNow Beautiful with CMS and Jelly.

Lists

Lists show several records in a table, line by line. Pretty obvious, but let's break out of the frames and examine some URLs again. Remember how we navigated to a specific record in a field? Let's instead show a list of Guest records. The table name is suffixed with _list, with the usual .do. Here's an example: http://<instance>.service-now.com/u_guest_list.do

You might be wondering what the .do is all about. This is the typical suffix that is used by Apache Struts, which has become the go-to framework for developing Java web applications like ServiceNow. This gives a hint as to the technologies used within the ServiceNow platform.

We've already seen that the sys_id can be used as a parameter to immediately jump to a record. There are other parameters that are useful too. Here's an example that shows how you can specify a database query through a URL: http://<instance>.service-now.com/sys_user_list.do?sysparm_query=user_name=fred.luddy.

If you navigate to this URL, you will be presented with a list of the records that match this query. If you remove the _list part from the URL, you will be presented with the first record that matches.
These URLs do not open up the familiar navigation frames, but simply show the content. You may want to have multiple browser tabs open, without the frameset. Edit the URL directly and get to where you want to go, fast. If you must have the frames, try http://<instance>.service-now.com/nav_to.do?uri=u_guest_list.do.

**Choosing the fields to show**

A list in ServiceNow can include any of the fields that are on the table. But a list works best when you show only the most relevant information. Adding in lots of columns takes longer to load (more data to get from the instance, to be sent across the Internet, and parsed by your browser) and often only adds to clutter and complexity.

Typically, this includes something that identifies the individual record (usually a name, number, and maybe a short description) and when the record was last updated. If there is a categorization field, that should be included too. It is very helpful to sort or group the records by these values. The **Go to** quick search option also allows you to search these fields.

The number of records that are shown on a list is configurable by the user. The System Administrator sets the choices they have in **UI Properties**. Keep the maximum number low, again to minimize the amount of data that needs to be worked with. Setting it to **1,000** is useful to be able to do mass deletion, but if everyone has it selected, it will impact performance.

**Having reference fields on lists**

Reference fields are treated slightly differently in a list view. The majority of fields are shown as simple text, but reference fields are always shown as a link to the referenced record. The first column in the list is also converted into a link, this time linking to the record itself.

Never put a reference field as the first column on a list. While the system will understand this, and consequently make the **second** column the link to the record, it is incredibly confusing to the user. People become very used to clicking on the first column and expect to see that record in the list.

You can always get to the record by clicking on the icon to the left of a particular column.
The varied capabilities of lists

Users of ServiceNow often forget about the extra functionality that lists provide. Functionality such as list editing and the powerful context menus (like Show Matching when you right-click on a list) should be thought about carefully and explained to users of the system to ensure they use the interface in an efficient manner.

A hierarchical list is not used that often, but is very powerful. It allows you to display the Related Lists of records on the list view. So even while looking at the Reservations list, the guests can still be inspected. You turn on this functionality in List Control.

Here are some tips to keep in mind when creating lists:

- Try not to include journal, HTML, or other multiline fields on the list. They just get big.
- Think carefully about List Control. Do you want New or Edit buttons? This especially matters on Related Lists.
- When running a query on a list, if you click on the New button, the values you searched for will be copied into the form.

Forms

In contrast to the simple concept of lists, a form generally contains more detailed information. It is where users usually interact with the data.

Try not to break away from the convention of having two columns of fields, with the labels to the left. Although it might be considered plain, it also means the forms are consistent, easy to read, and relatively uncluttered. The emphasis should therefore be on creating logic and process to control the data while keeping the interface simple.

If you want to make things more exciting, CSS can be applied to the main interface using Themes. Chapter 10, Making ServiceNow Beautiful with CMS and Jelly, explores how completely custom interfaces can be made. Check out the wiki for more information: http://wiki.servicenow.com/?title=CSS_Theme_Support.

Annotations allow you to add text and even HTML to forms. They are especially useful to add simple work instructions, but be careful to ensure the forms don't get cluttered.

Finally, formatters allow you to include Jelly on your form. Rather than a sugary treat, Jelly is a scriptable language used to build the ServiceNow interface. Chapter 10, Making ServiceNow Beautiful with CMS and Jelly, discusses custom interfaces in more detail.
Creating useful forms
By following some best practices, you can make the ServiceNow interface a more pleasant place to be!

- Every table needs a form, even if it is basic.
- Forms should read top to bottom, with important fields at the top left.
- The reference name or number of the record is normally at the top left.
- Lay out the fields in the order you’d fill them in. Users can tab between fields.
- Mandatory fields should be obvious, again, usually towards the top.
- Keep to the standard layout for consistency – two columns at the top and full width at the bottom.
- Keep forms as short as possible. Don’t include unnecessary fields. Views can be very useful to provide targeted designs.
- Use annotations to create Section Separators (not Form Sections) to separate out content on the page and provide a logical hierarchy or workflow of the data.
- For larger forms, use Form Sections. These are useful for creating tabs.
- Fields with multiple lines (such as descriptions or comments) should expand across the whole page, not half. This means they go at the bottom of the form.

Adding related and embedded lists
We’ve already seen Related Lists when discussing reference fields. But they come with a few disadvantages. Embedded lists remove some of their constraints:

- Embedded lists can be placed anywhere on the form, rather than just at the bottom.
- Since Related Lists show related records, they will only be displayed on a saved record. If it is unsaved, no records can be linked. Embedded lists will show at all times.

Let’s use an embedded list to create Guest records at the same time as a reservation:

1. On the Reservations form, first remove the Guests Related List by going to Personalize > Related Lists.
2. Under Personalize > Form Layout, find the Guests embedded list. It’ll show up in red in the Available column.
3. Once added and saved, a new **Reservations** record should look like the following screenshot. Guests can easily be added and removed from this interface by using list editing. Try it out!

![Reservation interface screenshot]

Embedded Related Lists are not always appropriate. They are designed to have an interface where you often create new related records with a minimum amount of information. There is no way to disable the creation of new records, for instance.

---

**Defining your own Related Lists**

**Defined Related Lists** gives a list of any records you want at the bottom of the form. For example, a simple Defined Related List may be placed on the **Room** form that lists other rooms on the same floor. This helps you to quickly navigate to them.

In order to create a Defined Related List, we'll need to use a little JavaScript. We'll work through this in more detail in the next chapter.

Navigate to **System Definition > Relationships** and click on **New**. Use the following details:

- **Name**: Rooms on the same floor
- **Applies to table**: Room [u_room]
- **Queries from table**: Room [u_room]
- **Query with**: current.addQuery('u_floor', parent.u_floor);
This code gets records where the `u_floor` field is the same as the record we are viewing. Two JavaScript variables are being used here, `current` is the table you are getting the records from and `parent` is the record that is being displayed in the form.

Navigate to the Room form and add the new Related List into the form. Now you'll see the other rooms listed as well. Useful!

![Room form with Related List](image)

**Creating tags and bookmarks**

Do you lose things? Me too. I lose my hotel key all the time. Perhaps you are writing several difficult scripts that you want only a couple of clicks away, or you need to quickly navigate to a User record. Tags are a way to collect arbitrary records together, making them easy to find, while bookmarks give single-click access.

**Adding a bookmark**

A way to quickly access information is through bookmarks. To the left of the Application Navigator is the Edge toolbar, which lets you control your interface. Try dragging links to the bar, where they will be saved for one-click access.

Since ServiceNow is a real web application, you can obviously add bookmarks through your browser. You might need to break out of the frames first or use the Copy URL option on a link breadcrumb to get a navigable URL.
Defining a tag

Tags collect records together. To create one, go to a record form. Click on the Tags icon next to the table name or right-click on the context menu and go to Assign Tag > New. Type in a label and save.

Then find your records by clicking on Tagged Documents in the Edge toolbar on the left-most part of the user interface:

In older versions of ServiceNow, this functionality was called labels. You can inspect the different user interfaces available on the wiki: http://wiki.servicenow.com/?title=Navigation_and_the_User_Interface.

You can also add tags to records from lists by selecting the items you want to add and then using Assign Tag from the Actions on selected rows choice list.

Tag configuration is possible by going to System Definition > Tags. This includes global tags (which show for every user) and dynamic tags (which try to automate the selection of labels for you; for example, the most recently used).

Enjoying views

If you find that you need to include lots of fields on a form or list, consider using views. They allow you to present a different set of elements that are specific to a situation. For example, in our hotel, a guest may be enrolled in our points program. In that case, we may want two views of user records: a simple uncluttered view for single-time guests (containing the minimum amount of information required to load quickly, and without extraneous data) and a more detailed view for frequent visitors (containing extra information to help serve them better).
Let's create a simple view of the **Guests** form.

1. Navigate to it now and then go to **Personalize > Form Design**.
2. On the **Default view** selection box, choose **New...** and enter **Simple** as the new view name.
3. Then remove all fields other than **First name**, **Last name**, and **Membership Number**, and click on **Save**.

A System Administrator (or a user with the **view_changer** role) can change views by clicking on the name of the table and choosing **View**. Otherwise, the view for the record is set through rules, through the view specified in the module link in the application menu to the left, or it is inherited.

The view of a record is inherited as you navigate through the interface. If you follow reference links, the system will attempt to be consistent and use the same view as before. If there isn't one with the same name, it will show the default one. Be aware of this behavior when you are naming and configuring forms.

### Controlling views

View Rules (available under **System UI > View Rules**) are a great way to force the display of a particular view. They work with a condition that uses information on the record itself. For example, you may decide to create a VIP view that shows extra fields. The VIP view is then only shown when the VIP field is ticked.

If you need more control, then create a script that can use other information to make the correct choice. A great use case for this is selecting a view based on the role of the logged-in user. Learn how to do this by going through the **Special function calls** section in **Chapter 2, Server-side Control**.

Views are often useful, but they can become frustrating. You end up managing several forms; for example, if you create a field and want it on all of them, you must repeat yourself several times and change the forms several times. And since the view you are using is kept as you navigate through the interface, be aware of which view you are editing: you may end up creating new views on forms unintentionally.

**Menus and modules**

To help you navigate the applications in ServiceNow, the interface provides you with the Application Navigator—or, as I like to call it, 'the menu to the left'. At its heart, this is a series of links to either forms or lists of data. They can specify a view name directly, and lists can include a filter, enabling you to decide exactly what the user sees when they click on it. This gives you a great deal of control.

What is shown in the Application Navigator is only natively controlled by roles. However, modules, like all configurations, are stored in a database table – the sys_app_module table to be exact. This gives rise to the possibility of restricting who sees modules in other ways. One example is creating a query Business Rule on this table to filter modules by group. Chapter 7, Securing Applications and Data, explores how that is accomplished.

**Specifying a view**

Let's set the default view for Guests to be our simple one.

1. Navigate to System Definition > Modules > Guests.
2. Change the View name field to Simple and save.

Now, each time you follow this link, you'll see the three-field view.

**Setting a filter**

When providing links to lists, it is a good idea to include a filter. Not only does it let you find the data you are looking more quickly, but it also reduces the need to immediately create a filter yourself. Often, you aren't interested in records that are six months old, for instance, so filter them out of the link. If you always filter the list (such as to find guests who have recently checked in), why not create a new module so you can jump straight to them?
Speak to the users of the system and understand what they are looking for. Not only will that make their interaction slightly easier, but you can also reduce the load on the instance by only displaying the appropriate information. Adding modules is really easy, and it can make a dramatic difference to usability.

Let’s create a new module that shows Reservations for today. Navigate to System Definition > Modules and click on New. Use these details:

- **Title**: Today’s Reservations
- **Application menu**: Hotel
- **Table**: Reservation [u_reservation]
- **Filter**: Arrival – on – Today

### Building the right modules

Menus and modules should be appropriately named. The Navigation Filter at the top is incredibly useful for selecting from the enormous list available to you as an administrator. And it is also helpful to power users. But the filter only matches on the Name parameter. For example, one of the module names that really frustrates me is the name of the link to view all the items in the system error log: All. The text to find this precise entry will therefore be all. Using log or error or other strings will either produce a lot of or no results, which to me is quite unintuitive. If you do type all in, you see lots of completely irrelevant entries. Besides that, All is not very descriptive! Something like All log entries will help in every respect.

### Summary

This chapter explored the key design principles that ServiceNow is built on, ensuring the foundations of the platform are well understood. A ServiceNow instance is a single-tenancy design, giving you a great deal of control and independence over how the platform works for you. The architecture of the system relies upon its database to store all configuration and data, so the hosting of ServiceNow gives a redundant pair for maintenance, disaster recovery, and performance reasons.

Creating tables and fields is a fundamental part of administrating ServiceNow. There are many field types available, from strings to URLs and journals fields. The dictionary stores information about each field and can make the values of the field unique, or act as the record’s display value.
Hierarchical tables give a great deal of benefit. Inheritance allows the Guest table to take advantage of all the functionality provided by the out-of-the-box User table.

In a relational system, linking records together is a key part of data design. Reference fields provide links between two records and give a great deal of capability and configuration choice, such as dot-walking, dynamic creation, and handling deletion. In addition to these, there are other types of relationships, such as many-to-many tables and Glide Lists.

The ServiceNow interface is built on lists and forms. These relate to the database tables they are showing and give a great deal of functionality – from hierarchical lists to tags and from views to filtered modules.

The next chapter will build on this data structure. By adding logic to the ServiceNow platform, we go beyond just storing data and instead get control over it.
Where to buy this book

You can buy Mastering ServiceNow 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.