**Django signals**: provide a flexible and powerful way to create interactions between different parts of your application without tightly coupling them.

| **Signal** | **Description** | **Use Case** |
| --- | --- | --- |
| **Django Signal** | A mechanism provided by Django to allow various parts of an application to communicate and react to certain events that occur within the application. | Used for decoupling components, enabling extensibility, and triggering actions in response to events. |
| **Types of Signals** |  |  |
| **1. Built-in Signals** | These signals are provided by Django itself and are emitted by different parts of the framework. | Example: **pre\_save** and **post\_save** signals for model instances, allowing actions before and after saving. |
| **2. Custom Signals** | These signals are created by developers to suit their application's needs. | Developers define the signals and when they are emitted, allowing other parts of the app to listen and respond. |
| **3. Third-party Signals** | Signals provided by third-party libraries used in Django projects. | Example: Django Allauth provides signals like **user\_signed\_up** that can be used to perform actions after user signup. |
| **Signal Components** |  |  |
| **1. Sender** | The object emitting the signal, often a model instance. | For example, a model's save method emitting **pre\_save** and **post\_save** signals. |
| **2. Signal** | The instance of the **django. dispatch. Signal** class representing the signal. | It acts as a bridge between senders and receivers, allowing connections to be made. |
| **3. Receiver** | A function or method that listens for a specific signal and performs actions when the signal is emitted. | Receivers are registered to specific signals and execute when the signal is sent. |
| **4. Connection** | The link established between the sender and receiver using the **Signal.connect()** method. | It binds the receiver function to the signal, allowing the receiver to be called when the signal is emitted. |
| **5. Dispatching** | The process of emitting a signal from the sender. | When an event (e.g., saving a model instance) occurs, the signal is emitted, and connected receivers are executed. |
| **Benefits of Signals** |  |  |
| **1. Decoupling** | Components can interact without direct dependencies, enhancing modularity and maintainability. | Changes in one part of the app (e.g., a model's behaviour) don't require changes in every connected receiver. |
| **2. Extensibility** | Allows third-party apps to hook into application events without modifying the core code. | Developers can extend the functionality of their apps without needing to modify or override existing code. |
| **3. Flexibility** | Enables dynamic behaviour by adding or removing receivers at runtime. | New functionalities can be added to the application without altering the existing codebase. |

CSRF Token

**What is CSRF?** Cross-Site Request Forgery (CSRF) is a type of attack where an attacker tricks a user into performing actions they didn't intend to perform on a different website. This can lead to unauthorized actions being taken on behalf of the user.

**What is a CSRF Token?** A CSRF token is a security measure used to mitigate CSRF attacks. It's a unique value associated with a user's session and embedded in forms or requests. When the form is submitted, the server checks if the token matches the expected value, ensuring that the request was intentionally made by the user.

**Example Scenario: Transferring Funds**

Let's imagine an online banking application where users can transfer funds between their accounts. We'll go through the process of adding a CSRF token to the transfer form.

**Step 1: Login**

1. Alice logs into her online banking account.
2. The server generates a unique CSRF token and associates it with Alice's session.

**Step 2: Transfer Form**

1. Alice decides to transfer $100 from her savings to checking account.
2. She navigates to the "Transfer Funds" section.
3. The server generates a new CSRF token and embeds it in a hidden field within the transfer form.

**Step 3: Attacker's Attempt**

1. An attacker, Eve, sends a phishing email to Alice.
2. The email contains a link to a malicious website controlled by Eve.
3. Alice, being unaware of the threat, clicks the link.

**Step 4: Malicious Site**

1. The malicious site loads, and Eve's JavaScript code automatically triggers a transfer request on Alice's behalf to the banking website.
2. Since Alice is logged into the banking site in the same browser, the browser automatically sends Alice's cookies, including her session cookies, to the banking site.

**Step 5: CSRF Protection in Action**

1. As the transfer request reaches the banking site, the server checks for the CSRF token embedded in the request.
2. The server verifies the token against the one associated with Alice's session.
3. Since the token doesn't match the expected value (because Eve's malicious site couldn't fetch the actual token from Alice's session), the server rejects the transfer request.
4. Eve's attempt to transfer funds fails.

**Result: Security Maintained**

Thanks to the CSRF token, the malicious transfer request is thwarted. Alice's funds remain safe, and the attack is prevented.

In this example, the CSRF token serves as a "secret handshake" that only the legitimate banking site and Alice's session know about. If a request doesn't include the correct token, the server identifies it as potentially malicious and denies the action.

By incorporating CSRF tokens, websites enhance the security of user actions, protecting them from unauthorized activities initiated by attackers.

**What is the @property Decorator?**

The **@property** decorator in Python is a way to define a method as if it were an attribute. It allows you to access a method like an attribute, providing a more intuitive interface to access class properties. It's often used to create "getter" methods for class attributes.